DTU Library

Formalization of Logic in the Isabelle Proof Assistant

Schlichtkrull, Anders

Publication date:
2018

Document Version
Publisher's PDF, also known as Version of record

Link back to DTU Orbit

Citation (APA):
Schlichtkrull, A. (2018). *Formalization of Logic in the Isabelle Proof Assistant*. DTU Compute. DTU Compute PHD-2018 Vol. 493

PhD thesis

Formalization of Logic in the Isabelle Proof Assistant

Anders Schlichtkrull

Technical University of Denmark

September 14, 2018

PHD-2018-493

Abstract

Isabelle is a proof assistant, i.e. a computer program that helps its user to define concepts in mathematics and computer science as well as to prove properties about them. This process is called formalization. Proof assistants aid their users by ensuring that proofs are constructed correctly and by conducting parts of the proofs automatically.

A logical calculus is a set of rules and axioms that can be applied to construct theorems of the calculus. Logical calculi are employed in e.g. tools for formal verification of computer programs. Two important properties of logical calculi are soundness and completeness, since they state, respectively, that all theorems of a given calculus are valid, and that all valid statements are theorems of the calculus. Validity is defined by a semantics, which gives meaning to formulas.

This thesis describes formalizations in Isabelle of several logics as well as tools built upon these. Specifically this thesis explains and discusses the following contributions of my PhD project:

- A formalization of the resolution calculus for first-order logic, Herbrand's theorem and the soundness and completeness of the calculus.
- A formalization of the ordered resolution calculus for first-order logic, an abstract prover based on it and the prover's soundness and completeness.
- A verified automatic theorem prover for first-order logic. The prover is a refinement of the above formalization of an abstract prover. This explicitly shows that the abstract notion of a prover can describe concrete computer programs.
- The Natural Deduction Assistant (NaDeA), which is a tool for teaching first-order logic that allows users to build proofs in natural deduction. The tool is based on a formalization of natural deduction and its soundness and completeness.
- A verified proof assistant for first-order logic with equality. It is based on an axiomatic system and constitutes a tool for teaching logic and proof assistants.
- A formalization of the propositional fragment of a paraconsistent infinite-valued higher-order logic. Theorems about the necessity of having infinitely many truth values are proved and formalized.

Proof assistants are built to reject proofs that contain gaps or mistakes. Therefore, the formalized results are highly trustworthy. The tools based on formalized calculi consequently have an increased trustworthiness. The above formalizations revealed flaws and mistakes in the literature. In addition to the formalizations and tools themselves, my PhD project contributes solutions that repair these flaws and mistakes.

Resumé

Titel: Formalisering af logik i Isabelle-bevisassistenten

Isabelle er en bevisassistent, dvs. et computerprogram, som kan hjælpe sin bruger med at definere koncepter fra matematik og computer science så vel som med at bevise deres egenskaber. Denne proces kaldes formalisering. Bevisassistenter hjælper deres brugere ved at sikre at beviser bliver konstrueret korrekt og ved at lave dele af beviserne automatisk.

En logisk kalkule er en mængde regler og aksiomer, som kan anvendes til at konstruere kalkulens sætninger. Logiske kalkuler bruges f.eks. i værktøjer til formel verifikation af computerprogrammer. To af kalkulers vigtige egenskaber er korrekthed og fuldstændighed, da de formulerer henholdsvis, at alle sætninger i en given kalkule er gyldige, og at alle gyldige udsagn er sætninger i kalkulen. Gyldighed er defineret af en semantik, som tilskriver formler mening.

Denne afhandling beskriver formaliseringer i Isabelle af adskillige logikker så vel som værktøjer der bygger på dem. Specifikt forklarer og diskuterer denne afhandling de følgende bidrag fra mit ph.d.-projekt:

- En formalisering af resolutionskalkulen for førsteordenslogik, Herbrands sætning og kalkulens korrekthed og fuldstændighed.
- En formalisering af den ordnede resolutionskalkule for førsteordenslogik, en abstrakt bevisfører baseret på den og bevisførerens korrekthed og fuldstændighed.
- En verificeret automatisk bevisfører for førsteordenslogik. Bevisføreren er en forfinelse af den ovenstående formalisering af en abstrakt bevisfører. Dette viser eksplicit, at den abstrakte forståelse af en bevisfører rigtignok kan beskrive et konkret computerprogram.
- Natural Deduction Assistant (NaDeA), som er et værktøj til at undervise i førsteordenslogik, der gør det muligt for sine brugere at bygge beviser i naturlig deduktion. Værktøjet er baseret på en formalisering af naturlig deduktion og dens korrekthed og fuldstændighed.
- En verificeret bevisassistent for førsteordenslogik med lighed. Den er baseret på et aksiomatisk system og udgør et værktøj til at undervise i logik og bevisassistenter.
- En formalisering af det udsagnslogiske fragment af en parakonsistent højereordenslogik med uendeligt mange sandhedsværdier. Sætninger om nødvendigheden af at have uendeligt mange sandhedsværdier bevises og formaliseres.

Bevisassistenter bygges til at afvise beviser, som indeholder huller eller fejl. Derfor er de formaliserede resultater meget pålidelige. Værktøjerne baseret på formaliserede kalkuler har derfor øget pålidelighed. De ovenstående formaliseringer viste mangler og fejl i litteraturen. Ud over formaliseringerne og værktøjerne i sig selv bidrager mit ph.d.-projekt med løsninger, som reparerer disse mangler og fejl.

Preface

The 3 years of PhD studies started 15.09.2015 and ended 14.09.2018. My PhD studies took place at the Department of Applied Mathematics and Computer Science (DTU Compute) of the Technical University of Denmark (DTU) under DTU Compute's PhD school, and were funded by DTU Compute. My main supervisor was Jørgen Villadsen (DTU Compute), and my co-supervisors were Jasmin Christian Blanchette (VU Amsterdam) and Thomas Bolander (DTU Compute).

Acknowledgements

First of all, I would like to thank my three supervisors Jørgen Villadsen, Jasmin Christian Blanchette and Thomas Bolander. Thank you for all the collaboration, help and inspiration, and for teaching me how to be a researcher. I could not have wished for better supervisors. I would also like to thank Christoph Weidenbach for generously hosting me at the Max Planck Institute for Informatics in Saarbrücken. The four months I spent in his group were a fun experience and I thank all the friends I made at the Max Planck Institute and the rest of the Saarland University Campus Saarbrücken. I would like to thank Christian Sternagel for inviting me to Innsbruck University and hosting me there. Working with you and René (Thiemann) on the interfaces between our projects IsaFoL and IsaFoR was a lot of fun. I was very honored to receive the Springer Travel Award at the International Conference on Interactive Theorem Proving (ITP 2016) and the Woody Bledsoe Student Travel Award at the International Joint Conference on Automated Reasoning (IJCAR 2018). I would like to thank Otto Mønsteds Fond for supporting my participation in the Federated Logic Conference (FLoC 2018). I would like to thank my co-authors Jasmin Christian Blanchette, Kasper Fabæch Brandt, Andreas Halkjær From, Alexander Birch Jensen, John Bruntse Larsen, Dmitriy Traytel, Jørgen Villadsen and Uwe Waldman for all our respective collaborations. I would like to thank all my colleagues at the Section for Algorithms, Logic and Graphs as well as my colleagues at the rest of DTU Compute. I would like to thank my friends and family of course!

Contents

Introduction 6

1 Formalization of the Resolution Calculus for First-Order Logic 16

2 Formalizing Bachmair and Ganzinger's Ordered Resolution Prover 48

3 A Verified Automatic Prover Based on Ordered Resolution 73

4 NaDeA: A Natural Deduction Assistant with a Formalization in Isabelle 103

5 Programming and Verifying a Declarative First-Order Prover in Isabelle/HOL 124

6 Formalized Meta-Theory of a Paraconsistent Logic 153

Thesis Appendix: Changes to Published Papers 167

Introduction

This introduction first motivates formalizing logic in the Isabelle proof assistant. It then gives a quick summary of the following chapters and their relations. Thereafter follow more thorough accounts of the chapters. Hereafter is an account of some new developments that build on the chapters, but which are not included in this thesis. Lastly the introduction gives discussions and perspectives on the results of the thesis.

Preliminaries and Motivation

Computer programs are central to many of the technological advances of modern day society. However, enormous amounts of resources are spent fighting problems caused by defects and bugs in computer programs. A way to solve this problem is to apply tools that prove correctness of software and thus avoid defects and bugs in the first place. Such tools can be built on a base of logic. In order to ensure that this base is solid, it should be studied thoroughly.

Logic is the study of reasoning. Of particular interest is the reasoning performed every day by mathematicians and computer scientists. Mathematicians prove properties about mathematical objects such as numbers, vectors, permutations and groups. Computer scientists prove properties about objects such as programs, algorithms, protocols and compilers. To prove something means to argue rigorously that it is true. A proved property is called a *theorem*. Examples of famous theorems are the Pythagorean theorem and the correctness of the quicksort sorting algorithm. The reasoning of mathematicians and computer scientists can be captured using *symbolic logic* which fixes a formal language and defines a logical calculus. A *logical calculus* consists of a set of *axioms*, i.e. theorems of the language that are considered self-evident, and a set of *rules* that defines when a theorem follows from a set of other theorems. Alternatively, or additionally, a *semantics* of the language is defined which assigns meaning to the statements and deems a subset of them valid. For an introductory textbook on the topic, I recommend the one by Ben-Ari [3].

This thesis concerns two applications of symbolic logic:

1. Firstly, symbolic logic allows reasoning about reasoning. It allows the use of mathematical reasoning to study different languages, sets of axioms, sets of rules and semantics as well as their relations.

2. Secondly, symbolic logic enables computers to do mathematical reasoning, since symbolic languages, calculi and semantics can be implemented as data and programs in programming languages. In particular, researchers in *computational logic* study how logical calculi can be implemented as programs called *theorem provers*. *Interactive theorem provers*, or *proof assistants*, involve user interaction to find proofs, while *automatic theorem provers* are fully automatic. See figure 1.

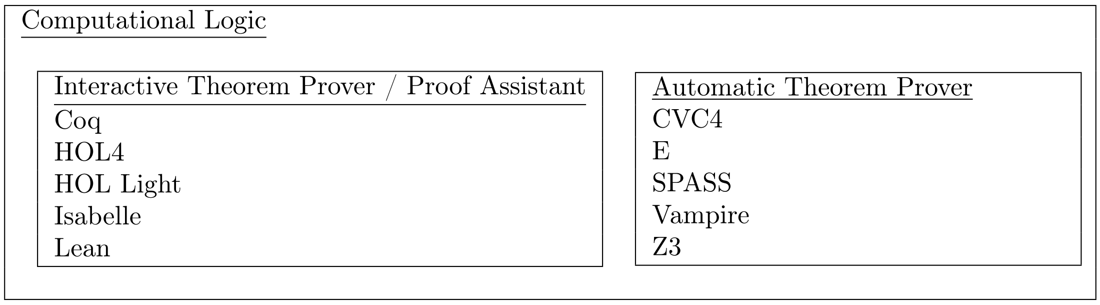


Figure 1: Computational logic can be partitioned into interactive theorem provers and automatic theorem provers. This figure shows a number of popular systems in both categories.

In this thesis, the second application is applied to the first! In this thesis I use higher-order logic, as implemented in the computer program Isabelle [27], to study the language, calculus and semantics of other logics. Of particular interest are the questions of *soundness and completeness*. For a calculus and a semantics, soundness states that any theorem of the calculus is valid according to the semantics. Completeness states, conversely, that any valid formula is a theorem. I also implement a number of tools and provers based on the formalized calculi.

Isabelle is a proof assistant, i.e. a computer program that helps its user to define concepts in mathematics and computer science as well as to prove properties about them. The process of defining concepts and proving properties in a proof assistant is called formalization. Isabelle can do this in many ways, including by ensuring correctness of all proofs written in the system and by helping to do parts of the proofs automatically. Isabelle attains this by implementing logical calculi. In this thesis I use Isabelle/HOL, which implements a higher-order logic. If we can convince ourselves that we trust the calculus and its implementation, then we must also trust the proofs that they accept. This is the foremost motivation for formalization – we obtain results that we can trust.

Another motivation is that of proving computer programs correct, as mentioned in the beginning of this section. This is also Paulson's motivation in his perspective [29] on computational logic. Programs can be modeled as expressions in Isabelle/HOL's language. Isabelle/HOL contains a code generator that can translate a subset of its language to a number of programming languages. This means that one can write programs as expressions in Isabelle/HOL's logical language, prove properties about them and then generate code from them, thus obtaining verified software. The thesis uses this approach to verify an automatic theorem prover and a proof assistant for first-order logic. Another technique to obtain trusted software is that of certification where a (perhaps unverified) program in addition to giving a result also returns a certificate which contains a proof of the correctness of the result. The certificate can then be checked by a (perhaps verified) trusted program.

Synopsis of the Following Chapters

The chapters of this thesis are related by all being within the topics of formalizing logics and basing software tools on such formalizations. Chapters 1, 2 and 3 are on the topic of first-order resolution. Chapter 1 formalizes an unordered resolution calculus and a completeness proof based on the technique of semantic trees which was introduced by Robinson [33]. Chapter 2 formalizes an abstract ordered resolution prover and a completeness proof based on Bachmair and Ganzinger's technique for model generation [1]. Chapters 2 and 3 are strongly related, since the concrete prover built in chapter 3 is based directly on the abstract prover in chapter 2 by providing a strategy to obtain fairness and by providing executable definitions for the needed operations on atoms, literals and clauses. All three chapters on resolution rely on the IsaFoR library [19] to e.g. obtain most general unifiers. Chapters 4 and 5 are on the topic of tools for teaching logic. Chapter 4 is on the Natural Deduction Assistant (NaDeA), which is a web application for teaching natural deduction that is based on a formalization of natural deduction. Chapter 5 is on a verified proof assistant for first-order logic based on an axiomatic system. On top of this proof assistant is implemented a tableau prover which is used as a subcomponent of the NaDeA web application. Chapter 6 formalizes the propositional fragment of a paraconsistent infinite-valued higher-order logic and its meta-theory.

Some of the chapters are based on previously published papers. The appendix of the thesis describes what has been changed from the published papers.

Chapter 1: Formalization of the Resolution Calculus for First-Order Logic

Chapter 1 describes how I formalized the resolution calculus for first-order logic in Isabelle. The chapter was published as a paper by me [34] in a special issue of "Journal of Automated Reasoning" on "Milestones in Interactive Theorem Proving". The audience of the journal includes researchers in both automatic and interactive theorem proving, and therefore I do not assume the reader is an expert on both of these two subjects at the same time. The resolution calculus is important because superposition, an extension to first-order logic with equality, is implemented in many of today's most efficient automatic theorem provers. The formalization includes formalized proofs of Herbrand's theorem, soundness and completeness. The formalization is, to the best of my knowledge, the first formalization of first-order resolution, its soundness and its completeness. Most of the formalization is based on the books by Ben-Ari [3] and Chang and Lee [13], however, both books have flawed proofs of a lemma used to prove completeness which is called the lifting lemma. For the proof of that lemma, I therefore followed the book by Leitsch [23]. In the completeness proof, the assumption is that the considered formula is unsatisfied by all interpretations with the Herbrand terms as their universe. The chapter formalizes that, in the assumption, the Herbrand terms can be replaced by any countably infinite universe. The formalization also elaborates on parts of the theory that were glossed over in the paper proofs – for instance the step from satisfiability by a path in a semantic tree to satisfiability by an interpretation. The chapter contains a thorough overview of formalizations of proof systems for first-order logic.

Chapter 2: Formalizing Bachmair and Ganzinger's Ordered Resolution Prover

Chapter 2 describes a formalization of the ordered resolution prover by Bachmair and Ganzinger [1]. The chapter is the technical report, extending the paper by Blanchette, Traytel, Waldmann and myself [37] which was published in the Proceedings of the 9th International Joint Conference on Automated Reasoning, IJCAR 2018. Ordered resolution is a restriction of resolution which enriches the resolution rule with a number of side-conditions that are not meant to ensure soundness, but instead to rule out certain inferences based on an order on terms. The overall idea is that proof search becomes more efficient when there are fewer inferences to choose between. Furthermore, the chapter shows the step from proof calculus to prover. The prover has a notion of redundancy deletion, i.e. in order to prevent the clause database from growing too large, the prover is allowed to delete clauses, but only in a way that does not compromise completeness.

The technique used to prove completeness is called the model generation technique. The formalization in the chapter is, to the best of my knowledge, the first formalization of the soundness and completeness of ordered resolution. The majority of Bachmair and Ganzinger's theory was easy to formalize – in particular the theory on ground resolution. Formalizing the section on first-order logic, however, we found challenging. We had to change a number of definitions in order to make the lemmas hold. In particular, the prover presented in the chapter turned out to be incomplete, but fortunately this was not difficult to repair. Aside from these and a number of other problems and unclarities, Bachmair and Ganzinger's prover withstood the challenge of formalization.

Chapter 3: A Verified Automatic Prover Based on Ordered Resolution

Chapter 3 describes a verified automatic prover based on ordered resolution. The chapter is a draft paper written by Blanchette, Traytel and myself [36]. It was written for a broad audience interested in the principles of programming languages, and thus does not assume expert knowledge on its topics of automatic and interactive theorem proving. It specifically shows how the abstract prover from chapter 2 can be refined to a verified prover in the Standard ML programming language. The verification is by refinement with the following four refinement layers:

- Layer 1 is the formalization of Bachmair and Ganzinger's prover from chapter 2. Clauses are represented by multisets of literals.
- Layer 2 enriches the prover using a priority queue in the clause database to ensure fairness. Clauses are paired with their timestamps stating when they were generated.
- Layer 3 is a deterministic prover with a specific strategy of when to perform inference and delete redundant clauses. Clauses are represented as finite lists and are again paired with timestamps.
- Layer 4 is an executable prover and so it replaces all the notions that were specified in the above layers with executable functions. From this layer, code is generated in the Standard ML programming language.

The verified prover is, to the best of my knowledge, the first verified sound and complete prover based on an optimized calculus and is thus interesting in itself. Additionally, the prover is interesting because it shows the viability of refining from the abstract theory of resolution down to a concrete prover. Therefore it also shows that the theory set up by Bachmair and Ganzinger indeed describes real provers.

Chapter 4: NaDeA: A Natural Deduction Assistant with a Formalization in Isabelle

Chapter 4 describes NaDeA which is a tool for teaching logic – specifically the natural deduction system. The chapter was published as a paper by Jensen, Villadsen and myself [48] in a special issue of the "IfCoLog Journal of Logics and their Applications" on "Tools for Teaching Logic". The paper is written for a broad audience with knowledge on logic, but not necessarily expertise on interactive theorem proving. The tool is motivated by what we consider three key ideals for a natural deduction assistant:

- It should be easy to use.
- It should make all the details of proofs clear and explicit.

- It should be based on a formalization that can be proved at least sound, but preferably also complete.

One of the reasons for teaching logic to computer science students is that it has applications in software verification. By verifying the soundness of the natural deduction system we practice what we preach. The tool is a web application written in TypeScript in which students can perform proofs in natural deduction. The chosen natural deduction system and the formalization is inspired by the work of Berghofer [6] but has a different syntax for first-order logic, a different set of rules, differently defined auxiliary notions and consequently a different soundness proof.

Chapter 5: Programming and Verifying a Declarative First-Order Prover in Isabelle/HOL

Chapter 5 describes a verified declarative first-order prover. The chapter was published as a paper by Jensen, Larsen, Villadsen and myself [20] in a special issue of "AI Communications" on "Automated Reasoning". Since it is written for the artificial intelligence community, we do not assume expert knowledge on its topics of automatic and interactive theorem proving. The prover is built on the LCF-principle in which only a small, trusted kernel can build theorems by exposing a number of functions that return theorems. The only way to make new functions that return theorems is to let them do function calls into the kernel. Therefore, any tool built in this way is at least as sound as the kernel. The specific prover we build is a Standard ML (SML) translation of John Harrison's declarative proof assistant for first-order logic from his "Handbook of Practical Logic and Automated Reasoning" [17]. We turn it into a verified prover by first writing a new kernel as definitions in Isabelle/HOL, then proving that it is sound and finally exporting the kernel to SML. Then, we replace the kernel of an SML translation of Harrison's prover with our verified kernel. The prover can be run inside the Isabelle/ML environment and thus both the prover and its verification can run in Isabelle.

Chapter 6: Formalized Meta-Theory of a Paraconsistent Logic

Chapter 6 presents the formalization by Villadsen and me of the propositional fragment of a paraconsistent infinite-valued higher-order logic by Villadsen [42, 43, 44, 45] and more recently Jensen and Villadsen [21]. The chapter is a draft paper written by me [35]. That the logic is paraconsistent means that it does not have the classical property that everything follows from a contradiction. In this specific logic, this is achieved by having more than the two classical truth values. Non-classical logics also deserve to be formalized, and this is the motivation for this work, as well as for exploring what the infinitely many truth values mean for the notion of validity. The chapter combines results from papers by Villadsen and myself [49, 50] with new results. Both the results and their formalization have been developed as part of my PhD project. One result is that it is only necessary to consider a finite subset of the infinitely many interpretations in order to find out whether a formula in the logic is valid. Another result is that, if we make a restricted version of the logic with only finitely many truth values, then we get another logic. This is proved by demonstrating a formula that is valid in the finite-valued logic, but not the infinite-valued logic.

Other Developments

During my PhD studies I contributed to a number of other developments. Villadsen, From and I [51] worked on a simple prover for first-order logic with the goal of being easy to understand, easy to verify and easy to modify. Its starting point was the work by Ridge and Margetson [31, 32]. Villadsen, From and I [46] also presented the extension of NaDeA with the ability to be able to export proofs to Isabelle/HOL. This connects the NaDeA implementation with its formalized logic, and we can think of the exported proof as a certificate and Isabelle/HOL as the program that can check these certificates. Furthermore, we presented a formalized completeness result for NaDeA. In contrast to Berghofer's formalization of natural deduction, as originally published in 2007, there is no requirement for the considered formula to be closed. In 2018, From also removed this requirement from Berghofer's formalization [6]. Villadsen, From and I presented NaDeA and the Students' Proof Assistant (SPA) – a new version of the prover from chapter 5 – in papers for the "Theorem proving components for Educational software (ThEdu)" community at their 2018 workshop in Oxford [38, 47].

Discussions and Perspectives

The formalizations and tools presented are of course contributions in themselves. This means increased trust in the results that have been formalized and the tools. Furthermore, formalization requires theorem statements and definitions to be made precise, which is valuable in itself as it can clear up some of the unclarities that occur in natural language and means that there are no forgotten corner cases. The formalization allows Isabelle to keep track of exactly where assumptions and lemmas are used. If an assumption or lemma seems not to be needed, the user can try to delete it and Isabelle will report where the proof breaks.

A valuable by-product of the formalizations is that they revealed new facets of old results. Chapters 1 and 2 showed flaws and mistakes in published results and showed how to solve these problems. But even setting these aside, formalizing the theory revealed how to deal with some of the details that are glossed over. For example, chapter 1 elaborated on the conversions from paths in trees to Herbrand interpretations, chapter 2 showed the details of the ordered resolution calculus's lifting lemma and both chapters showed explicitly how to incorporate renaming.

Another by-product is that the libraries of Isabelle/HOL grow. Isabelle/HOL includes definitions and theorems itself, and in addition to these exists the *Archive of Formal Proofs (AFP)*, which is a library that contains more than 100,000 lemmas [10, 14]. This means that researchers who want to formalize their favorite results do not have to start from scratch, but can build on the available definitions and lemmas. The formalizations in this project are part of IsaFoL, which is a project that unites researchers in the area of formalizing logics. As the developments there mature, they are moved to the AFP.

Proof assistants benefit from automatic theorem provers by using them to find proofs. For example, Isabelle/HOL includes the tools *Metis* [18, 30], Isabelle's *smt* command [11] and *Sledgehammer* [8], which are based on automatic theorem provers. These tools are used in the chapters of this thesis. All three tools follow the idea of having an automatic theorem prover find a proof which is then reconstructed in Isabelle's trusted kernel. As demonstrated in chapters 1, 2 and 3, automatic theorem provers can also benefit from proof assistants, in that proof assistants can be used to study the theoretical properties of automatic theorem provers and their underlying calculi. This shows that the community around proof assistants, also called interactive theorem provers (ITPs), and the community around automatic theorem provers (ATPs) benefit from each other – there is a lot of cooperation across the border seen in figure 1. I see more opportunities for the research communities around these two topics to work together.

One difference between the two communities is their focus on different logics. The ATP community has a fondness for first-order logic, while the ITP community has one for higher-order logics and other type theories. This is also the case in figure 1, where all the ITPs are for higher-order logics or other type theories and the ATPs are for first-order logic – some including various theories. The split is not clear cut though: ITPs do incorporate first-order provers, and on the ATP front there are a number of provers for higher-order logic including Leo-II [5], Leo-III [40] and Satallax [12]. Furthermore, there is work on extending Zipperposition [4] and Vampire [7] to deal with the higher-order case.

Another difference between the two communities is in their focus. In the ATP community, systems are compared on performance in the annual CASC [41] and SMT-COMP [2] competitions. ATPs obtain competitiveness using advanced data structures and complex optimized algorithms and heuristics. ITPs, on the other hand, typically have architectures which emphasize trust in their results, e.g. using the LCF-principle in which a small kernel implements a logic in a way that is optimized for clarity. This shows that the ATP community has focus on efficiency, while the ITP community has focus on trust.

If we consider the verified ATP in chapter 3, we notice that its definition and soundness proof are rather involved, while the soundness theorem's statement itself is simple. The result is a tool with a high degree of trust despite its relatively complex code. A similar conclusion can be made concerning the works on verifying SAT solvers [25, 26, 9, 28, 24, 39, 15]. Likewise, if we took the verified ITP kernel from chapter 5 and changed its data structures and function definitions to be more complex then it would make the soundness theorem's statement no more complicated. In principle, we could go so far as to include a whole ATP in the kernel. The ITP kernel in chapter 5 is for first-order logic, but the same could be done for higher-order logic, e.g. based on the work of Harrison [16] and that of Kumar, Arthan, Myreen and Owens [22] on verifying the HOL Light ITP. What we see is a way to obtain systems with high degrees of both efficiency and trust. This is of course not the only way to combine the efficiency of an ATP with the trust of an ITP – the approach of the aforementioned *smt* command, Metis and Sledgehammer obtains the same advantages using certification of their results. The approach of these tools is perhaps more practical, since it has the advantage of not requiring verification of the code that does the search for a proof. On the other hand, verifying ATP and ITP tools gives us an opportunity to tie the theory behind our tools together with their implementations and to study their completeness. The two approaches complement each other and both show the close relationship between ITP and ATP.

References

[1] L. Bachmair and H. Ganzinger. Resolution theorem proving. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, pages 19–99. Elsevier and MIT Press, 2001.

[2] C. Barrett, L. de Moura, and A. Stump. SMT-COMP: Satisfiability Modulo Theories Competition. In K. Etessami and S. K. Rajamani, editors, *Computer Aided Verification (CAV)*, pages 20–23. Springer, 2005.

[3] M. Ben-Ari. *Mathematical Logic for Computer Science.* Springer, 3rd edition, 2012.

[4] A. Bentkamp, J. C. Blanchette, S. Cruanes, and U. Waldmann. Superposition for lambda-free higher-order logic. In D. Galmiche, S. Schulz, and R. Sebastiani, editors, *International Joint Conference on Automated Reasoning (IJCAR)*, pages 28–46. Springer, 2018.

[5] C. Benzmüller, N. Sultana, L. C. Paulson, and F. Theiß. The higher-order prover Leo-II. *Journal of Automated Reasoning*, 2015.

[6] S. Berghofer. First-order logic according to Fitting. *Archive of Formal Proofs*, Aug. 2007. `http://isa-afp.org/entries/FOL-Fitting.shtml`, Formal proof development.

[7] A. Bhayat and G. Reger. Set of support for higher-order reasoning. In *6th Workshop on Practical Aspects of Automated Reasoning (PAAR)*, pages 2–16, 2018. `http://ceur-ws.org/Vol-2162/`.

[8] J. C. Blanchette, S. Böhme, and L. C. Paulson. Extending Sledgehammer with SMT solvers. *Journal of Automated Reasoning*, 51(1):109–128, 2013.

[9] J. C. Blanchette, M. Fleury, P. Lammich, and C. Weidenbach. A verified SAT solver framework with learn, forget, restart, and incrementality. *Journal of Automated Reasoning*, 61(1–4):333–365, 2018.

[10] J. C. Blanchette, M. Haslbeck, D. Matichuk, and T. Nipkow. Mining the Archive of Formal Proofs. In M. Kerber, J. Carette, C. Kaliszyk, F. Rabe, and V. Sorge, editors, *Conference on Intelligent Computer Mathematics (CICM)*, pages 3–17. Springer, 2015.

[11] S. Böhme and T. Weber. Fast LCF-style proof reconstruction for Z3. In M. Kaufmann and L. C. Paulson, editors, *Interactive Theorem Proving (ITP)*, pages 179–194. Springer, 2010.

[12] C. E. Brown. Satallax: An automatic higher-order prover. In B. Gramlich, D. Miller, and U. Sattler, editors, *International Joint Conference on Automated Reasoning (IJCAR)*, pages 111–117. Springer, 2012.

[13] C.-L. Chang and R. C.-T. Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, Inc., 1st edition, 1973.

[14] M. Eberl, G. Klein, T. Nipkow, L. Paulson, and R. Thiemann. Archive of Formal Proofs – statistics. `https://www.isa-afp.org/statistics.html`.

[15] M. Fleury, J. C. Blanchette, and P. Lammich. A verified SAT solver with watched literals using imperative HOL. In J. Andronick and A. P. Felty, editors, *Certified Programs and Proofs (CPP)*, pages 158–171. ACM, 2018.

[16] J. Harrison. Towards self-verification of HOL Light. In U. Furbach and N. Shankar, editors, *International Joint Conference on Automated Reasoning (IJCAR)*, volume 4130 of *LNCS*, pages 177–191. Springer, 2006.

[17] J. Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, 2009.

[18] J. Hurd. First-order proof tactics in higher-order logic theorem provers. In M. Archer, B. D. Vito, and C. Muñoz, editors, *Design and Application of Strategies/Tactics in Higher Order Logics (STRATA)*, NASA Technical Reports, pages 56–68, 2003.

[19] IsaFoR developers. An Isabelle/HOL formalization of rewriting for certified termination analysis. `http://cl-informatik.uibk.ac.at/software/ceta/`.

[20] A. B. Jensen, J. B. Larsen, A. Schlichtkrull, and J. Villadsen. Programming and verifying a declarative first-order prover in Isabelle/HOL. *AI Communications*, 31(3):281–299, 2018.

[21] A. S. Jensen and J. Villadsen. Paraconsistent computational logic. In P. Blackburn, K. F. Jørgensen, N. Jones, and E. Palmgren, editors, *8th Scandinavian Logic Symposium: Abstracts*, pages 59–61. Roskilde University, 2012.

[22] R. Kumar, R. Arthan, M. O. Myreen, and S. Owens. Self-formalisation of higher-order logic. *Journal of Automated Reasoning*, 56(3):221–259, 2016.

[23] A. Leitsch. *The Resolution Calculus*. Springer, 1997.

[24] S. Lescuyer. *Formalizing and Implementing a Reflexive Tactic for Automated Deduction in Coq*. PhD thesis, Université Paris-Sud, 2011.

[25] F. Marić. Formal verification of modern SAT solvers. *Archive of Formal Proofs*, July 2008. Formal Proof Development. `http://isa-afp.org/entries/SATSolverVerification.html`.

[26] F. Marić. Formal verification of a modern SAT solver by shallow embedding into Isabelle/HOL. *Theoretical Computer Science*, 411(50):4333–4356, 2010.

[27] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[28] D. Oe, A. Stump, C. Oliver, and K. Clancy. `versat`: A verified modern SAT solver. In V. Kuncak and A. Rybalchenko, editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 7148 of *LNCS*, pages 363–378. Springer, 2012.

[29] L. C. Paulson. Computational logic: its origins and applications. *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 474(2210), 2018.

[30] L. C. Paulson and K. W. Susanto. Source-level proof reconstruction for interactive theorem proving. In K. Schneider and J. Brandt, editors, *Theorem Proving in Higher Order Logics (TPHOLs)*, pages 232–245. Springer, 2007.

[31] T. Ridge. A mechanically verified, efficient, sound and complete theorem prover for first order logic. *Archive of Formal Proofs*, Sept. 2004. `http://isa-afp.org/entries/Verified-Prover.shtml`, Formal proof development.

[32] T. Ridge and J. Margetson. A mechanically verified, sound and complete theorem prover for first order logic. In *Theorem Proving in Higher Order Logics (TPHOLs)*, pages 294–309, 2005.

[33] J. A. Robinson. The generalized resolution principle. *Machine Intelligence*, 3:77–93, 1968.

[34] A. Schlichtkrull. Formalization of the resolution calculus for first-order logic. *Journal of Automated Reasoning*, 61(4):455–484, 2018.

[35] A. Schlichtkrull. Formalized meta-theory of a paraconsistent logic. 2018. Submitted.

[36] A. Schlichtkrull, J. C. Blanchette, and D. Traytel. A verified automatic prover based on ordered resolution. 2018. Submitted.

[37] A. Schlichtkrull, J. C. Blanchette, D. Traytel, and U. Waldmann. Formalizing Bachmair and Ganzinger's ordered resolution prover. In D. Galmiche, S. Schulz, and R. Sebastiani, editors, *International Joint Conference on Automated Reasoning (IJCAR)*, pages 89–107. Springer, 2018. Extended in technical report: `http://matryoshka.gforge.inria.fr/pubs/rp_report.pdf`.

[38] A. Schlichtkrull, J. Villadsen, and A. H. From. Students' Proof Assistant (SPA). In *7th International Workshop on Theorem proving components for Educational software (ThEdu)*, 2018.

[39] N. Shankar and M. Vaucher. The mechanical verification of a DPLL-based satisfiability solver. *Electronic Notes in Theoretical Computer Science*, 269:3–17, 2011.

[40] A. Steen and C. Benzmüller. The higher-order prover Leo-III. In D. Galmiche, S. Schulz, and R. Sebastiani, editors, *International Joint Conference on Automated Reasoning (IJCAR)*, pages 108–116. Springer, 2018.

[41] G. Sutcliffe. The CADE ATP System Competition - CASC. *AI Magazine*, 37(2):99–101, 2016.

[42] J. Villadsen. Combinators for paraconsistent attitudes. In P. de Groote, G. Morrill, and C. Retoré, editors, *Logical Aspects of Computational Linguistics (LACL)*, volume 2099 of *LNCS*, pages 261–278. Springer, 2001.

[43] J. Villadsen. Paraconsistent assertions. In G. Lindemann, J. Denzinger, I. J. Timm, and R. Unland, editors, *Multi-Agent System Technologies*, volume 3187 of *LNCS*, pages 99–113, 2004.

[44] J. Villadsen. A paraconsistent higher order logic. In B. Buchberger and J. A. Campbell, editors, *Artificial Intelligence and Symbolic Computation*, volume 3249 of *LNCS*, pages 38–51. Springer, 2004.

[45] J. Villadsen. Supra-logic: Using transfinite type theory with type variables for paraconsistency. *Logical Approaches to Paraconsistency, Journal of Applied Non-Classical Logics*, 15(1):45–58, 2005.

[46] J. Villadsen, A. H. From, and A. Schlichtkrull. Natural deduction and the Isabelle proof assistant. In P. Quaresma and W. Neuper, editors, *Proceedings 6th International Workshop on Theorem proving components for Educational software (ThEdu)*, volume 267 of *Electronic Proceedings in Theoretical Computer Science*, pages 140–155. Open Publishing Association, 2018.

[47] J. Villadsen, A. H. From, and A. Schlichtkrull. Natural Deduction Assistant (NaDeA). In *7th International Workshop on Theorem proving components for Educational software (ThEdu)*, 2018.

[48] J. Villadsen, A. B. Jensen, and A. Schlichtkrull. NaDeA: A natural deduction assistant with a formalization in Isabelle. *IfCoLog Journal of Logics and their Applications*, 4(1):55–82, 2017.

[49] J. Villadsen and A. Schlichtkrull. Formalization of Many-Valued Logics. In H. Christiansen, M. Jiménez-López, R. Loukanova, and L. Moss, editors, *Partiality and Underspecification in Information, Languages, and Knowledge*, chapter 7. Cambridge Scholars Publishing, 2017.

[50] J. Villadsen and A. Schlichtkrull. Formalizing a paraconsistent logic in the Isabelle proof assistant. In A. Hameurlain, J. Küng, R. Wagner, and H. Decker, editors, *Transactions on Large-Scale Data- and Knowledge-Centered Systems (TLDKS)*, volume 10620 of *LNCS*, pages 92–122. Springer, 2017.

[51] J. Villadsen, A. Schlichtkrull, and A. H. From. A verified simple prover for first-order logic. In *6th Workshop on Practical Aspects of Automated Reasoning (PAAR)*, pages 88–104, 2018.

Formalization of the Resolution Calculus for First-Order Logic

Anders Schlichtkrull

DTU Compute, Technical University of Denmark, Kongens Lyngby, Denmark

Abstract

I present a formalization in Isabelle/HOL of the resolution calculus for first-order logic with formal soundness and completeness proofs. To prove the calculus sound, I use the substitution lemma, and to prove it complete, I use Herbrand interpretations and semantic trees. The correspondence between unsatisfiable sets of clauses and finite semantic trees is formalized in Herbrand's theorem. I discuss the difficulties that I had formalizing proofs of the lifting lemma found in the literature, and I formalize a correct proof. The completeness proof is by induction on the size of a finite semantic tree. Throughout the paper I emphasize details that are often glossed over in paper proofs. I give a thorough overview of formalizations of first-order logic found in the literature. The formalization of resolution is part of the IsaFoL project, which is an effort to formalize logics in Isabelle/HOL.

Keywords First-order logic, Resolution, Isabelle/HOL, Herbrand's theorem, Soundness, Completeness, Semantic trees

1 Introduction

The resolution calculus plays an important role in automatic theorem proving for first-order logic as many of the most efficient automatic theorem provers, e.g. E [68], SPASS [74], and Vampire [56], are based on superposition, an extension of resolution. Studying the resolution calculus is furthermore an integral part of many university courses on logic in computer science. The resolution calculus was introduced by Robinson in his ground-breaking paper [59] which also introduced most general unifiers (MGUs).

The resolution calculus reasons about first-order literals, i.e. atoms and their negations. Since the literals are first-order, they may contain full first-order terms. Literals are collected in clauses, i.e. disjunctions of literals. The calculus is refutationally complete, which means that if a set of clauses is unsatisfiable, then the resolution calculus can derive a contradiction (the empty clause) from it. One can also use the calculus to prove any valid sentence by first negating it, then transforming it to an equisatisfiable set of clauses, and lastly refuting this set with the resolution calculus. Resolution is a calculus for first-order logic, but it does not have any machinery to handle equality or any other theories.

There are several techniques for proving the completeness of resolution calculi. In this work I use the one of semantic trees, which was introduced by Robinson [60]. Semantic trees are binary trees that represent interpretations. I mostly follow textbooks by Ben-Ari [4], Chang and Lee [19], and Leitsch [43]. The idea of Chang and Lee's completeness proof is that a semantic tree is cut smaller and smaller, and for each cut, a derivation is done towards the empty clause.

I also formalize Herbrand's theorem, which cuts the tree down to finite size. I prove a stronger version of the usual refutational completeness theorem by weakening its assumption to require unsatisfiability in only a single countably infinite universe instead of in all universes. The usual theorem follows directly from this, which is proven, e.g. by Chang and Lee as Theorem 4.2. I discuss why this usual theorem is not formalized.

The formalization is included in the IsaFoL project [33] and the Archive of Formal Proofs [64] where it is available for download. The IsaFoL project formalizes several logics in Isabelle/HOL [47]. IsaFoL is part of a larger effort of research in this area. This also includes formalizations of ground resolution, which is propositional by nature. The formalization in this paper stands out from these by formalizing resolution for first-order logic. The theory needed to do this is very different from that of ground resolution since first-order logic involves a richer syntax and semantics. To the best of my knowledge, I present the first formalized completeness proof of the resolution calculus for first-order logic.

Harrison [28] formalized Herbrand's theorem, also known as uniformity, in a model theoretic formulation. It says that if a purely existential formula is valid, then some disjunction of instances of the body is propositionally valid. In automatic theorem proving, the theorem is viewed in a different, equivalent way: A set of clauses is unsatisfiable only if some finite set of ground, i.e. variable free, instances of its clauses is as well. This can be used to build a first-order refutation prover from a propositional SAT solver. Such a prover enumerates ground instances, which it tries to refute with the SAT solver. I formalize a third equivalent view stating exactly what the completeness proof needs: If a set of clauses is unsatisfiable, then there is a finite semantic tree whose branches falsify the set. This bridges first-order unsatisfiability with decisions made in a semantic tree.

Understanding proofs of logical systems can be challenging since one must keep separate the parts of the proofs that are about the syntactic level, and the parts that are about the semantic level. It can be tempting to mix intuition about syntax and semantics. Fortunately, a formalization makes the distinction very clear, and ideally this can aid in understanding the proofs.

This paper extends my previous paper [63] which I presented at ITP 2016. It is extended with more thorough explanations and now contains illustrative examples of structured Isar proofs. Furthermore, the discussion of the tools used in the formalization has been expanded, and the related-works section now contains a much more thorough overview of the formalizations of first-order logic found in the literature. Additionally, the formalization now contains three new versions of the soundness theorem and two new illustrative versions of the completeness theorem, which are explained.

2 Overview

This section introduces the terminology of clausal first-order logic and the resolution rule. It gives a brief explanation of semantic trees and gives the big picture of the proofs of Herbrand's theorem, the lifting lemma, and completeness.

A *literal* l is either an atom or its negation. The *sign* of an atom is *True*, while that of its negation is *False*. The *complement* p^{c} of an atom p is $\neg p$, and the complement $(\neg p)^{\mathrm{c}}$ of its negation is p. The complement L^{C} of a set of literals L is $\{l^{\mathrm{c}} \mid l \in L\}$. The set of variables in a set of literals L is $vars_{\mathrm{ls}}\ L$. A *clause* is a set of literals representing the universal closure of the disjunction of the literals in the clause. The empty clause represents a contradiction since it is an empty disjunction. A clause with an empty set of variables is called *ground*. A *substitution* σ is a function from variables to terms, and is applied to a clause C by applying it to all variables in C.

The result is written $C \cdot_{\mathrm{ls}} \sigma$ and is called an instance of C. We can likewise apply a substitution to a single literal $l \cdot_{\mathrm{l}} \sigma$ or term $t \cdot_{\mathrm{t}} \sigma$. The composition $\sigma_1 \cdot \sigma_2$ of two substitutions is the substitution that maps any variable x to $(\sigma_1\ x) \cdot_{\mathrm{t}} \sigma_2$. A *unifier* σ for a set of literals L is a substitution such that applying it to L makes all the literals therein equal. A *most general unifier (MGU)* for a set of literals L is a unifier σ for L such that any other unifier for L can be expressed as $\sigma \cdot \tau$ for some substitution τ.

We will consider the following formulation of the resolution rule:

$$\frac{C_1 \qquad\qquad C_2}{((C_1 - L_1) \cup (C_2 - L_2)) \cdot_{\mathrm{ls}} \sigma} \quad \begin{array}{l} \mathit{vars}_{\mathrm{ls}}\ C_1 \cap \mathit{vars}_{\mathrm{ls}}\ C_2 = \{\} \\ L_1 \subseteq C_1,\ L_2 \subseteq C_2 \\ \sigma \text{ is a substitution and an MGU of } L_1 \cup {L_2}^{\mathrm{C}} \end{array}$$

The conclusion of the rule is called a *resolvent* of C_1 and C_2. L_1 and L_2 are called *clashing* sets of literals. Additionally, the calculus allows us to apply variable renaming to clauses before we apply the resolution rule. Renaming variables in two clauses C_1 and C_2 such that $\mathit{vars}_{\mathrm{ls}}\ C_1 \cap \mathit{vars}_{\mathrm{ls}}\ C_2 = \{\}$ is called *standardizing apart*. Notice that L_1 and L_2 are sets of literals. Some other resolution calculi instead let L_1 and L_2 be single literals. These calculi then have an additional rule called factoring, which allows unification of subsets of clauses. The completeness of the above rule implies the completeness of resolution on single literals with factoring, as explained by e.g. Fitting [25], but I have not formalized this result. The idea is that the above rule can be simulated by applications of resolution on single literals and factoring.

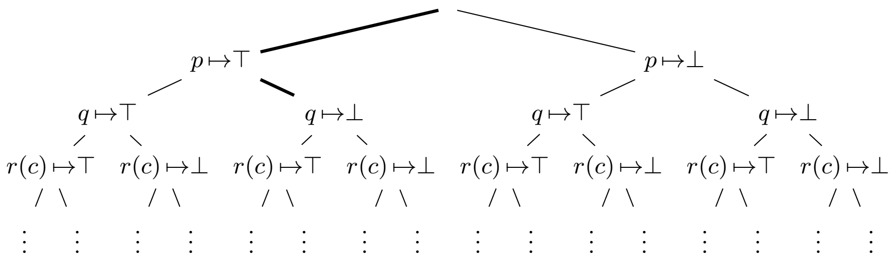
I now give an overview of the completeness proof. The completeness proof is very much inspired by that of Chang and Lee [19], while the proof of the lifting lemma is inspired by that of Leitsch [43].

Semantic trees are defined from an enumeration of Herbrand, i.e. ground, atoms. A semantic tree is essentially a binary decision tree in which the decision of going left in a node on level i corresponds to mapping the ith atom of the enumeration to *True*, and in which going right corresponds to mapping it to *False*. See Fig. 1. Therefore, a finite path in a semantic tree can be seen as a *partial interpretation*. This differs from the usual interpretations in first-order logic in two ways. Firstly, it does not consist of a function denotation and a predicate denotation, but instead assigns *True* and *False* to ground atoms directly. Secondly, it is finite, which means that some ground literals are assigned neither *True* nor *False*. A partial interpretation is said to *falsify a ground clause* if it, to all literals in the clause, assigns the opposite of their signs. A *branch* is a path from the root of a tree to one of its leaves. An *internal path* is a path from the root of a tree to some node that is not a leaf. A *closed path* is a path whose corresponding partial interpretation falsifies some ground instance of a clause in the set of clauses. A *closed semantic tree* for a set of clauses is a tree that has two properties: Firstly, each of its branches is closed. Secondly, the internal paths in the tree are not closed. The second property expresses minimality of the first property, because it ensures that no proper subtree of a closed semantic tree can have the first property.

Note that Chang and Lee's notion of semantic trees is more general than mine since it allows each decision to assign truth values to several atoms. This generality is not needed in the completeness proof, and therefore I prefer a simpler definition in order to ease formalization.

Herbrand's theorem is proven in the following formulation: If a set of clauses is unsatisfiable, then there is a finite and closed semantic tree for that set. I prove it in its contrapositive formulation and therefore assume that all finite semantic trees of a set of clauses have an open (non-closed) branch. By obtaining longer and longer branches of larger and larger finite semantic trees, we can, using König's lemma, obtain an infinite path, all of whose prefixes are open branches of finite semantic trees. Thus these branches satisfy, that is, do not falsify, the set of clauses. We can then prove that this infinite path, when seen as an Herbrand interpretation, also

Figure 1: Semantic tree with partial interpretation $[p \mapsto \mathit{True}, q \mapsto \mathit{False}]$.



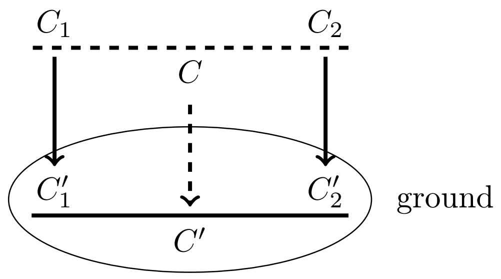


Figure 2: The lifting lemma. An arrow from C to C' indicates that C' is an instance of C. The bars are derivations. Full bars or arrows are relations we know, and the dashed ones are established by the lemma.

satisfies the set of clauses, and this concludes the proof. Converting the infinite path to a full interpretation can be seen as the step that goes from syntax to semantics.

The lifting lemma lifts resolution derivation steps done on the ground level up to the first-order world. The lemma considers two instances, C'_1 and C'_2, of two first-order clauses, C_1 and C_2. It states that if C'_1 and C'_2 can be resolved to a clause C' then also C_1 and C_2 can be resolved to a clause C. And not only that, it can even be done in such a way that C' is an instance of this C. See Fig. 2. To prove the theorem, we look at the clashing sets of literals $L'_1 \subseteq C'_1$ and $L'_2 \subseteq C'_2$. We partition C'_1 in L'_1 and the rest, $R'_1 = C'_1 - L'_1$. Then we lift this up to C_1 by partitioning it in L_1, the part that instantiates to L'_1, and the rest R_1, which instantiates to R'_1. We do the same for C_2. Since L'_1 and ${L'_2}^{\mathrm{C}}$ can be unified, so can L_1 and ${L_2}^{\mathrm{C}}$, and therefore they have an MGU. Thus C_1 and C_2 can be resolved to a resolvent C. With some bookkeeping of the substitutions and unifiers, we can also show that C has the ground resolvent C' as an instance.

Lastly, *completeness* itself is proven. It states that the empty clause can be derived from any unsatisfiable set of clauses. We start by obtaining a finite closed semantic tree for the set of clauses. Then we cut off two sibling leaves. The branches ending in these leaves agree on all atoms except for the one, a, in their leaves. Additionally they falsify a ground clause each, but, by minimality of closed trees, their prefixes do not. Therefore, setting a to *True* in a sibling, must have falsified a clause, and thus the literal $\neg a$ must be in a clause. Likewise, setting a to *False* in a sibling, must have falsified a clause, and thus the literal a must be in a clause. These clauses can be resolved. We lift this up to the first-order world by the lifting lemma and resolve the first-order clauses. Repeating this procedure, we obtain a derivation that ends when we have cut the tree down to the root. Only the empty clause can be falsified here, so we have a derivation of the empty clause.

3 Isabelle

This section explains the logic of Isabelle/HOL and the Isar language [75] for writing structured proofs. Isar is illustrated with some simple examples.

Isabelle is a generic proof assistant that implements several logics, and Isabelle/HOL is its implementation of a higher-order logic (HOL). HOL can be seen as a combination of typed functional programming and logic. This gives, among other things, access to the usual logical operators and quantifiers such as $\longrightarrow$, $\wedge$, $\vee$, $\neg$, $\forall$ and $\exists$. The long arrow ($\Longrightarrow$) is Isabelle's meta-implication, which for the purpose of this paper can be thought of as a normal implication ($\longrightarrow$), and likewise the big wedge ($\bigwedge$) can be thought of as universal quantification ($\forall$).

In Isabelle's Isar language one can write structured proofs that both humans can read and Isabelle/HOL can check. I present a subset here, which is large enough for the reader to understand this paper. Let us consider a template Isar proof:

theorem *L:*
  assumes a_1*:* A_1
  $\vdots$
  assumes a_n*:* A_n
  shows B
proof R
  C_1
  $\vdots$
  C_m
qed

Here L is the theorem's name, $A_1, \ldots, A_n$ are optional assumptions of the theorem, $a_1, \ldots, a_n$ are optional names of the assumption, and B is the theorem's conclusion. If there are no assumptions the keyword **shows** may be omitted. R instructs Isabelle on how to start the proof. For instance, if nothing is written, it applies a well-suited rule, and if a dash ($-$) is written, then no rule is applied. $C_1, \ldots, C_m$ is a list of statements, similar to the sentences of a paper proof, which is to prove the theorem. Let us look at three kinds of statements. First, we have the **have** goal:

from F_1 **have** s*:* S **using** F_2 **by** M

Here S is a proposition which is proven by proof method M. Proof method M could be one of Isabelle/HOL's proof methods that implement automatic theorem provers. s is an optional name of S. F_1 and F_2 are lists of names of facts that M is allowed to use. They could be names of previously proven theorems, assumptions or of a proposition of one of the preceding statements. Both **from** F_1 and **using** F_2 can be omitted. Additionally **from** F_1 can be replaced with **then**, which refers to the fact that was most recently established, i.e. the proposition in the previous statement.

Second, we have the **obtain** goal:

from F_1 **obtain** t **where** s*:* S **using** F_2 **by** M

Here t is a new constant that is introduced in the proof. S is a proposition that characterizes t. It is named s. F_1 and F_2 are lists of facts that the proof method M is instructed to use to prove the existence of t.

Third, we have the **show** goal:

from F_1 **show** s*:* S **using** F_2 **by** M

This is similar to the **have** goal except that it requires S to be one of the propositions that R instructs us to prove. Sometimes S will be $?thesis$, which refers to B. When we have shown all statements required by R we can end the proof with **qed**.

Let us look at a variation of a simple proof of Cantor's theorem from an introduction to Isabelle/HOL by Nipkow and Klein [46] that illustrates the language. The theorem states that a function from a set to its powerset cannot be surjective. Here the set is formalized as a type $'a$ and its powerset as the type $'a\ set$.

```
theorem cantor: ¬ surj(f :: 'a ⇒ 'a set)
proof
  assume surj f
  then have ∀ A. ∃ a. A = f a using surj-def by metis
  then have ∃ a. {x. x ∉ f x} = f a by blast
  then obtain a where {x. x ∉ f x} = f a by blast
  then show False by blast
qed
```

A list of statements can also form a calculation. In the example below the horizontal ellipses (. . .) are part of the concrete Isabelle syntax while the vertical ellipsis (⋮) indicates that some intermediate steps were omitted.

```
have s₁: S₀ = S₁ using F₁ by M₁
also have s₂: . . . = S₂ using F₂ by M₂
  ⋮
also have sₙ: . . . = Sₙ using Fₙ by Mₙ
finally have sₙ₊₁: S₀ = Sₙ using Fₙ₊₁ by Mₙ₊₁
```

This list of statements proves $S_0 = S_n$ by proving $S_0 = S_1 = S_2 = \cdots = S_n$ where the first equality $S_0 = S_1$ is proven by the first **have** goal and each subsequent equality $S_i = S_{i+1}$ is proven by the **also have** goal with name s_i.

For example we can prove a simple lemma about the identity function:

```
lemma identities:
  assumes ∀ y. identity y = y
  shows identity (identity (identity x)) = x
proof −
  have identity (identity (identity x)) = identity (identity x) using assms by auto
  also have ... = identity x using assms by auto
  also have ... = x using assms by auto
  finally show identity (identity (identity x)) = x by −
qed
```

Isar allows many more kinds of constructions of proofs, for instance nesting proofs, combining proof methods and more.

4 Clausal First-Order Logic

This section explains the formalization of the syntax and semantics of first-order clausal logic.

First, a signature is fixed where variable symbols, function symbols, and predicate symbols are represented by the type *string*. The type *string* consists of strings over a finite alphabet, and is thus a countably infinite type.

type-synonym *var-sym* = *string*
type-synonym *fun-sym* = *string*
type-synonym *pred-sym* = *string*

Similar to, e.g. Berghofer's formalization of first-order logic [5], the predicate and function symbols do not have fixed arities.

A first-order term is either a variable consisting of a variable symbol or it is a function application consisting of a function symbol and a list of subterms:

datatype *fterm* = *Var var-sym* | *Fun fun-sym* (*fterm list*)

A literal is either positive or negative, and it contains a predicate symbol (a string) and a list of terms. The datatype is parametrized with the type of terms *'t* since it will both represent first-order literals (*fterm literal*) and Herbrand literals. A clause is a set of literals.

datatype *'t literal* = *Pos pred-sym* (*'t list*) | *Neg pred-sym* (*'t list*)

type-synonym *'t clause* = *'t literal set*

Ground *fterm literal*s are formalized using a predicate $ground_l$ which holds for l if it contains no variables. Ground *fterm clause*s are similarly formalized using a predicate $ground_{ls}$.

A semantics of terms and literals is also formalized. A variable denotation, *var-denot*, maps variable symbols to values of the domain. The universe is represented by the type variable *'u*.

type-synonym *'u var-denot* = *var-sym* $\Rightarrow$ *'u*

Interpretations consist of denotations of functions and predicates. A function denotation maps function symbols and lists of values to values:

type-synonym *'u fun-denot* = *fun-sym* $\Rightarrow$ *'u list* $\Rightarrow$ *'u*

Likewise, a predicate denotation maps predicate symbols and lists of values to the two boolean values:

type-synonym *'u pred-denot* = *pred-sym* $\Rightarrow$ *'u list* $\Rightarrow$ *bool*.

The semantics of a term is defined by the recursive function $eval_t$:

fun $eval_t$ :: *'u var-denot* $\Rightarrow$ *'u fun-denot* $\Rightarrow$ *fterm* $\Rightarrow$ *'u* **where**
 $eval_t\ E\ F\ (Var\ x) = E\ x$
 $|eval_t\ E\ F\ (Fun\ f\ ts) = F\ f\ (map\ (eval_t\ E\ F)\ ts)$

Here, $map\ (eval_t\ E\ F)\ [e_1, \ldots, e_n] = [eval_t\ E\ F\ e_1, \ldots, eval_t\ E\ F\ e_n]$, and from now on $map\ (eval_t\ E\ F)\ ts$ is abbreviated as $eval_{ts}\ E\ F\ ts$.

If an expression evaluates to *True* in an interpretation, we say that it is satisfied by the interpretation. If it evaluates to *False*, we say that it is falsified. The semantics of literals is a function $eval_l$ that evaluates literals:

fun $eval_l$ :: *'u var-denot* $\Rightarrow$ *'u fun-denot* $\Rightarrow$ *'u pred-denot* $\Rightarrow$ *fterm literal* $\Rightarrow$ *bool*
 where
 $eval_l\ E\ F\ G\ (Pos\ p\ ts) \longleftrightarrow\ \ G\ p\ (eval_{ts}\ E\ F\ ts)$
 $|eval_l\ E\ F\ G\ (Neg\ p\ ts) \longleftrightarrow \neg G\ p\ (eval_{ts}\ E\ F\ ts)$

The semantics is extended to clauses:

```
definition evalc :: 'u fun-denot ⇒ 'u pred-denot ⇒ fterm clause ⇒ bool where
  evalc F G C ⟷ (∀E. ∃l ∈ C. evall E F G l)
```

It is important that the ranges of all the environments that *eval*$_c$ quantifies over are actually subsets of the considered universe. The type system of Isabelle/HOL ensures this, as we can inspect that the type of E indeed is *'u var-denot*. Had I instead chosen to represent the universe as a set, I would have to pass it as an argument to *eval*$_c$ and have a predicate ensure that all the environments considered did not go outside this universe. Likewise, I would also have to make a decision of what to do if the range of F was not a subset of the universe.

A set of clauses *Cs* is satisfied, written *eval*$_{cs}$ *F G Cs*, if all its clauses are satisfied:

```
definition evalcs :: 'u fun-denot ⇒ 'u pred-denot ⇒ fterm clause set ⇒ bool where
  evalcs F G Cs ⟷ (∀C ∈ Cs. evalc F G C)
```

The semantics can be illustrated with the universe *nat* of natural numbers, a function denotation that maps *add*, *mul*, *one*, and *zero* to their usual meanings, a predicate denotation that maps *less*, *greater*, and *equals* to their usual meanings, as well as a variable denotation that maps x to 26 and y to 5:

```
fun Fnat :: nat fun-denot where
  Fnat f [n,m] =
     (if f = ''add'' then n + m else
      if f = ''mul'' then n * m else 0)
| Fnat f [] =
     (if f = ''one''  then 1 else
      if f = ''zero'' then 0 else 0)
| Fnat f us = 0

fun Gnat :: nat pred-denot where
  Gnat p [x,y] =
     (if p = ''less'' ∧ x < y then True else
      if p = ''greater'' ∧ x > y then True else
      if p = ''equals'' ∧ x = y then True else False)
| Gnat p us = False

fun Enat :: nat var-denot where
  Enat x =
     (if x = ''x'' then 26 else
      if x = ''y'' then 5 else 0)
```

It is also illustrative to evaluate the literal $equals(add(mul(y, y), one), x)$ with the above denotations:

```
lemma evall Enat Fnat Gnat
       (Pos ''equals''
         [Fun ''add'' [Fun ''mul'' [Var ''y'',Var ''y''],Fun ''one'' []]
         ,Var ''x'']
       ) = True
  by auto
```

5 Substitutions and Unifiers

This section formalizes substitutions, unifiers, MGUs, and the unification theorem, which states the existence of MGUs.

A substitution is a function from variable symbols into terms:

type-synonym *substitution* = *var-sym* $\Rightarrow$ *fterm*

This is very different from Chang and Lee where they are represented by finite sets [19]. The advantage of functions is that they make it much easier to apply and compose substitutions. If C' is an instance of C we write *instance-of*$_{\text{ls}}$ C' C. The composition of two substitutions, σ_1 and σ_2, is also defined, and written $\sigma_1 \cdot \sigma_2$. We also define unifiers and MGUs of literals (and similarly of terms):

definition $\textit{unifier}_{\text{ls}}\ \sigma\ L \longleftrightarrow (\exists l'.\ \forall l \in L.\ l \cdot_{\text{l}} \sigma = l')$

definition $\textit{mgu}_{\text{ls}}\ \sigma\ L \longleftrightarrow \textit{unifier}_{\text{ls}}\ \sigma\ L \wedge (\forall u.\ \textit{unifier}_{\text{ls}}\ u\ L \longrightarrow \exists i.\ u = \sigma \cdot i)$

One important theorem is the unification theorem, which states that if a finite set of literals has a unifier, then it also has an MGU. This is usually proven by defining a unification algorithm and proving it correct. This has been formalized several times. An early formalization is by Paulson [48] in LCF of an algorithm by Manna and Waldinger [44]. Coen [21] used this as basis for a formalization of the algorithm in Isabelle, and his formalization was improved first by Slind [72] and later Krauss [38]. Their formalization [20] is now part of the Isabelle distribution. There, terms are formalized as binary tree structures and substitutions as association lists. Sternagel and Thiemann [73] formalize in the IsaFoR project [34] an algorithm presented by Baader and Nipkow [2]. They formalize terms, unifiers and MGUs in a similar way to me. Therefore it is relatively easy to obtain the unification theorem by proving my terms, unifiers, and MGUs equivalent to the ones in IsaFoR.

theorem *unification:*
  assumes *finite* L
  assumes $\textit{unifier}_{\text{ls}}\ \sigma\ L$
  shows $\exists \theta.\ \textit{mgu}_{\text{ls}}\ \theta\ L$

For the purpose of formalizing the resolution calculus the choice of unification algorithm is irrelevant since we only need one to prove the existence of MGUs. If one wants to formalize a resolution prover the choice is important especially with respect to runtime. The two presented algorithms seem to be efficient in practice, but have an exponential worst-case runtime. Ruiz-Reina, Martín-Mateos, and Hidalgo [61], however, formalize, in ACL2, an algorithm by Corbin and Bidoit [22] as presented by Baader and Nipkow [2], which has a quadratic worst-case runtime. Some automatic theorem provers, e.g. SPASS, use the technique of term indexing to compute MGUs – see, e.g. Sekar, Ramakrishnan, and Voronkov's chapter on the topic [69]. I do not know of any formalization of this technique in a proof assistant.

6 The Resolution Calculus

This section formalizes the resolution calculus and its soundness proof. It also formalizes steps and derivations in the resolution calculus.

First, resolvents are formalized, i.e. the conclusions of the resolution rule:

definition $\textit{resolution}\ C_1\ C_2\ L_1\ L_2\ \sigma = ((C_1 - L_1) \cup (C_2 - L_2)) \cdot_{\text{ls}} \sigma$

In Sect. 2 we saw that the resolution rule had three side-conditions. The rule is additionally restricted to require that L_1 and L_2 are non-empty. When these side-conditions are fulfilled, the rule is applicable.

definition *applicable* C_1 C_2 L_1 L_2 $\sigma \longleftrightarrow$
$C_1 \neq \{\} \wedge C_2 \neq \{\} \wedge L_1 \neq \{\} \wedge L_2 \neq \{\}$
$\wedge$ $vars_{\text{ls}}$ $C_1 \cap vars_{\text{ls}}$ $C_2 = \{\}$
$\wedge L_1 \subseteq C_1 \wedge L_2 \subseteq C_2$
$\wedge$ mgu_{ls} σ $(L_1 \cup L_2{}^{\text{C}})$

A step in the resolution calculus either inserts a resolvent of two clauses in a set of clauses, or it inserts a variable renaming of one of the clauses. Two clauses are variable renamings of each other if they can be instantiated to each other. Alternatively, we could say that we apply a substitution which is a bijection between the variables in the clause and another set of variables.

definition *var-renaming-of* :: *fterm clause* $\Rightarrow$ *fterm clause* $\Rightarrow$ *bool* **where**
var-renaming-of C_1 C_2 $\longleftrightarrow$ *instance-of*$_{\text{ls}}$ C_1 C_2 $\wedge$ *instance-of*$_{\text{ls}}$ C_2 C_1

A step in the resolution calculus is formalized as an inductive predicate named *resolution-step*. In Isabelle/HOL this is done by specifying a number of rules characterizing the predicate. Specifically there are two rules. One resolution-rule allows us to apply the resolution rule, and the other standardize-apart allows us to rename clauses such that we can standardize them apart.

inductive *resolution-step* :: *fterm clause set* $\Rightarrow$ *fterm clause set* $\Rightarrow$ *bool* **where**
resolution-rule*:*
$C_1 \in Cs \Longrightarrow C_2 \in Cs \Longrightarrow$ *applicable* C_1 C_2 L_1 L_2 $\sigma \Longrightarrow$
resolution-step Cs $(Cs \cup \{$*resolution* C_1 C_2 L_1 L_2 $\sigma\})$
| standardize-apart*:*
$C \in Cs \Longrightarrow$ *var-renaming-of* C $C' \Longrightarrow$ *resolution-step* Cs $(Cs \cup \{C'\})$

Derivation steps are extended to derivations by taking the reflexive transitive closure of *resolution-step*, which is given by *rtranclp*:

definition *resolution-deriv* = *rtranclp resolution-step*

The soundness proofs in the three books were not immediately ready to be formalized. The proof by Ben-Ari uses Herbrand interpretations, but this machinery is actually not necessary to prove soundness and does not seem to give a simpler proof. Chang and Lee prove soundness for first-order logic by referring to the soundness proof for the propositional case, but they do not make it clear how variables should be handled. Leitsch's soundness proof refers to the substitution principle, but neither states nor proves it. It can be found elsewhere, e.g. in the textbook by Ebbinghaus, Flum, and Thomas [24] in the form of the substitution lemma. The formalized soundness proof also uses the substitution lemma.

I prove the resolution rule sound by combining three simpler rules:

1. A substitution rule that allows us to infer instances.

2. A special, simpler, resolution rule.

3. A superset rule that allows us to infer supersets.

Rule 1, the substitution rule, states that we can do substitution:

$$\frac{C}{C \cdot_{\text{ls}} \sigma}$$

Informally this seems obvious. C is satisfied and is a first-order clause, i.e. it represents a universal quantification. $C \cdot_{\text{ls}} \sigma$ then instantiates its variables, which are bound and universally quantified, and must therefore also be satisfied. Formally, however, this is not precise enough since C being satisfied is a statement about variable denotations, i.e. a semantic form of instantiation, while a substitution is a syntactic form of instantiation. This problem is overcome by the substitution principle. The needed insight is that given a function denotation and a variable denotation, any substitution can be converted to a variable denotation by evaluating the terms of its domain. In the formalization this is done using Isabelle/HOL's function composition operator which is written as $\circ$ in infix notation.

definition $\mathit{evalsub}\ E\ F\ \sigma = (\mathit{eval}_{\mathrm{t}}\ E\ F) \circ \sigma$

The substitution lemma then states that applying a substitution to a literal is semantically the same as instead turning the substitution into a variable denotation:

lemma *substitution:* $\mathit{eval}_{\mathrm{l}}\ E\ F\ G\ (l \cdot_{\mathrm{l}} \sigma) \longleftrightarrow \mathit{eval}_{\mathrm{l}}\ (\mathit{evalsub}\ E\ F\ \sigma)\ F\ G\ l$

Let us now look at the soundness proof of substitution. The proof is written in Isar and uses *evalsub* and the substitution lemma:

```
lemma subst-sound:
 assumes asm: $eval_{c}\ F\ G\ C$
 shows $eval_{c}\ F\ G\ (C \cdot_{ls} \sigma)$
unfolding $eval_{c}$-def proof
  fix $E$
  from asm have $\forall E'.\ \exists l \in C.\ eval_{l}\ E'\ F\ G\ l$ using $eval_{c}$-def by blast
  then have $\exists l \in C.\ eval_{l}\ (evalsub\ E\ F\ \sigma)\ F\ G\ l$ by auto
  then show $\exists l \in C \cdot_{ls} \sigma.\ eval_{l}\ E\ F\ G\ l$ using substitution by blast
qed
```

Notice that I am **unfolding** the definition of $\mathit{eval}_{\mathrm{c}}$ before the **proof** begins. The definition says that $\mathit{eval}_{\mathrm{c}}$ is a universal quantification over the variable denotations. Therefore Isabelle now requires us to **fix** an arbitrary variable denotation and find a satisfied literal in $C \cdot \sigma$. By the assumption C has such a literal for any variable denotation E' and in particular for σ transformed to a variable denotation $\mathit{evalsub}\ E\ F\ \sigma$. The substitution lemma allows the substitution to be applied instead of transformed and this concludes the proof.

Rule 2, the special substitution rule, is a special, ground-like, version of the resolution rule. The rule is special since it is only allowed to remove two literals l_1 and l_2 instead of two sets of literals and because it requires l_1 and l_2^{c} to be equal instead of unifiable:

$$\frac{C_1 \qquad\qquad C_2}{(C_1 - \{l_1\}) \cup (C_2 - \{l_2\})} \begin{array}{l} l_1 \in C_1 \\ l_2 \in C_2 \\ l_1 = l_2^{\mathrm{c}} \end{array}$$

Rule 3, the superset rule, states that from a clause follows any superset of the clause:

$$\frac{C_1}{C_1 \cup C_2}$$

The proofs of all three rules are made as short structured Isar proofs.

These four sound rules are combined to give the resolution rule, which must consequently be sound. We are of course allowed to use the assumptions of the resolution rule, so we know that when σ is applied to L_1 and L_2, they turn into a complementary pair of literals, which we denote $l_1 \cdot_{\mathrm{ls}} \sigma$ and $l_2 \cdot_{\mathrm{ls}} \sigma$. This justifies the bookkeeping inference below. It also means that we can apply the special resolution rule. The bottom-most rule application uses the superset rule.

$$\dfrac{\dfrac{\dfrac{\dfrac{C_1}{C_1 \cdot_{\text{ls}} \sigma} \qquad \dfrac{C_2}{C_2 \cdot_{\text{ls}} \sigma}\ \text{substitution rule}}{(C_1 \cdot_{\text{ls}} \sigma - \{l_1 \cdot_{\text{ls}} \sigma\}) \cup (C_2 \cdot_{\text{ls}} \sigma - \{l_2 \cdot_{\text{ls}} \sigma\})}\ \text{special resolution}}{(C_1 \cdot_{\text{ls}} \sigma - L_1 \cdot_{\text{ls}} \sigma) \cup (C_2 \cdot_{\text{ls}} \sigma - L_2 \cdot_{\text{ls}} \sigma)}\ \text{book keeping}}{((C_1 - L_1) \cup (C_2 - L_2)) \cdot_{\text{ls}} \sigma}\ \text{superset rule}$$

All this reasoning is made as structured Isar proofs. The soundness theorem is stated as follows:

theorem *resolution-sound:*
  assumes $eval_{\text{c}}\ F\ G\ C_1 \wedge eval_{\text{c}}\ F\ G\ C_2$
  assumes $applicable\ C_1\ C_2\ L_1\ L_2\ \sigma$
  shows $eval_{\text{c}}\ F\ G\ (resolution\ C_1\ C_2\ L_1\ L_2\ \sigma)$

From this it follows that resolution steps are sound:

theorem *step-sound:*
  assumes $resolution\text{-}step\ Cs\ Cs'$
  assumes $eval_{\text{cs}}\ F\ G\ Cs$
  shows $eval_{\text{cs}}\ F\ G\ Cs'$

And then it follows that resolution derivations are sound:

theorem *derivation-sound:*
  assumes $resolution\text{-}deriv\ Cs\ Cs'$
  assumes $eval_{\text{cs}}\ F\ G\ Cs$
  shows $eval_{\text{cs}}\ F\ G\ Cs'$

The soundness theorem is also formalized in the refutational style:

theorem *derivation-sound-refute:*
  assumes $resolution\text{-}deriv\ Cs\ Cs' \wedge \{\} \in Cs'$
  shows $\neg eval_{\text{cs}}\ F\ G\ Cs$

To summarize, I have defined a function *resolution* giving the conclusion of the resolution rule, as well as a predicate *applicable* which formalizes its side conditions. I have combined these to form the predicates *resolution-step* and *resolution-derivation* which formalize when a set of clauses follows from another, respectively by a step or derivation of the resolution calculus. The resolution rule and its steps and derivations were proven sound.

7 Herbrand Interpretations and Semantic Trees

Now that soundness is proven, it is time to take the first steps towards proving completeness. Therefore this section formalizes Herbrand interpretations and semantic trees. It also formalizes Herbrand's theorem and emphasizes how an infinite path in a semantic tree is transformed to an interpretation.

Herbrand interpretations are a special kind of interpretation characterized by two properties. The first is that their universe is the set of all Herbrand terms. I chose that universes should be represented by types and this is of course also the case for the universe of Herbrand terms. Therefore, a new type *hterm* is introduced which is similar to *fterm*, but does not have a constructor for variables:

datatype *hterm* = *HFun fun-sym* (*hterm list*)

This is the same datatype as in Berghofer's formalization of natural deduction [5]. Had I chosen to represent the universes by sets like Ridge and Margetson [58], then I could instead have represented the Herbrand universe by the set of ground *fterm*s.

Two functions called *fterm-of-hterm* and *hterm-of-fterm* are introduced that convert between *hterm*s and ground *fterm*s. Note that some authors require the terms in the Herbrand universe to be built from the function symbols in a considered set of clauses. I choose to use all function symbols because it allows the Herbrand universe to be represented by the above datatype.

The second characteristic property is that the function denotation of an Herbrand interpretation is *HFun*, and thus, evaluating a ground term under such an interpretation corresponds to replacing all applications of *Fun* with *HFun*, that is, the ground term is interpreted as itself.

As we saw in Sect. 2, an enumeration of Herbrand atoms is needed, such that we can construct our semantic trees. Therefore, the type of atoms is defined:

type-synonym *'t atom* = *pred-sym* ∗ *'t list*

Again the symbols are not restricted to those occurring in a considered set of clauses. Isabelle/HOL provides the proof method *countable-datatype* that can automatically prove that a given datatype, in our case *hterm*, is countable. Since also the predicate symbols are countable, then so must *hterm atom* be. Furthermore, it is easy to prove that there are infinitely many *hterm atom*s. Using these facts and Hilbert's choice operator, I specify a bijection *hatom-of-nat* between the natural numbers and the *hterm atom*s. Its inverse is called *nat-of-hatom*. Additionally, the functions *nat-of-fatom* and *fatom-of-nat* enumerate the ground *fterm* atoms in the same order. A function *get-atom* returns the atom corresponding to a literal. The enumeration will be used to define which levels of the semantic trees correspond to which atoms.

7.1 Semantic Trees

In paper-proofs semantic trees are often labeled with the atoms that their nodes set to *True* or *False*. In this formalization the trees are unlabeled, because for a given level, the corresponding atom can always be calculated using the enumeration:

datatype *tree* = *Leaf* | *Branching tree tree*

The formalization contains a quite substantial, approximately 700-lines, theory on these unlabeled binary trees, paths within them, and their branches. The details are not particularly interesting, but a theory of binary trees is necessary.

In the formalization, *bool list*s represent both paths in trees and partial interpretations, denoted by the type *partial-pred-denot*. E.g. if we consider the path [*True*, *True*, *False*], then it is the path from the root of a semantic tree that goes first left, then left again, and lastly right. On the other hand, it is also the partial interpretation that considers *hatom-of-nat 0* to be *True*, *hatom-of-nat 1* to be *True* and *hatom-of-nat 2* to be *False*. Our formalization illustrates the correspondence between partial interpretations and paths clearly by identifying their types. Therefore, synonym *dir* is introduced for *bool* as well as the abbreviations *Left* for *True* and *Right* for *False*.

The above datatype cannot represent infinite trees. Thus, infinite trees are modeled as sets of paths with a wellformedness property:

abbreviation *wf-tree* :: *dir list set* ⇒ *bool* **where**
  wf-tree T ≡ (∀*ds d*. (*ds* @ *d*) ∈ *T* ⟶ *ds* ∈ *T*)

Alternatively I could have used Isabelle's codatatype package [8, 10], since codatatypes can represent infinite-depth trees in a very natural way.

Infinite paths are modeled as functions from natural numbers into finite paths. Applying the function to number i gives us the prefix of length i. From here on such functions are called infinite paths, and their characteristic property is

> **abbreviation** $\mathit{wf\text{-}infpath} :: (\mathit{nat} \Rightarrow\ '\mathit{a\ list}) \Rightarrow \mathit{bool}$ **where**
> $\quad \mathit{wf\text{-}infpath}\ f \equiv (f\ 0 = []) \wedge (\forall n.\ \exists a.\ f\ (\mathit{Suc}\ n) = (f\ n)\ @\ [a])$

It must be made formal, what it means for a partial interpretation, i.e. a path, to falsify an expression. A partial interpretation G falsifies, written $\mathit{falsifies}_l\ G\ l$, a ground literal l, if the opposite of its sign occurs on index $\mathit{nat\text{-}of\text{-}fatom}\ (\mathit{get\text{-}atom}\ l)$ of the interpretation. The exclamation mark (!) is Isabelle/HOL's *nth* operator, i.e. $G\ !\ i$ gives the ith element of G.

> **definition** $\mathit{falsifies}_l :: \mathit{partial\text{-}pred\text{-}denot} \Rightarrow \mathit{fterm\ literal} \Rightarrow \mathit{bool}$ **where**
> $\quad \mathit{falsifies}_l\ G\ l \longleftrightarrow \mathit{ground}_l\ l$
> $\quad\quad \wedge\ (\mathit{let}\ i = \mathit{nat\text{-}of\text{-}fatom}\ (\mathit{get\text{-}atom}\ l)\ \mathit{in}$
> $\quad\quad\quad i < \mathit{length}\ G \wedge G\ !\ i = (\neg \mathit{sign}\ l))$

A ground clause C is falsified, written $\mathit{falsifies}_g\ G\ C$, if all its literals are falsified. A first-order clause C is falsified, written $\mathit{falsifies}_c\ G\ C$, if it has a falsified ground instance. A partial interpretation satisfies an expression if the partial interpretation does not falsify it. A set Cs of first-order clauses is falsified by a partial interpretation if it falsifies some clause in Cs. A set Cs of first-order clauses is falsified by a tree if each of the tree's branches falsifies some clause in Cs. Lastly, a semantic tree T is closed, written $\mathit{closed\text{-}tree}\ T\ Cs$, for a set of clauses Cs if it is a tree that falsifies Cs, but whose internal paths do not. Notice that a closed tree is minimal with respect to having falsifying branches, since any proper subtree has a branch that does not falsify anything in the set.

7.2 Herbrand's Theorem

The formalization of Herbrand's theorem is mostly straightforward and is done as an Isar proof that follows the sketch from Sect. 2. The challenging part is to take an infinite path, all of whose prefixes satisfy a set of clauses Cs and then prove that its translation to an interpretation also satisfies Cs. Chang and Lee [19] do not elaborate much on this, but it takes up a large part of the formalization and illustrates the interplay of syntax and semantics.

The first step is to define how to transform the infinite path to an Herbrand interpretation. The function denotation has to be *HFun*, and the infinite path needs to be converted to a predicate denotation. This can be done as follows:

> **abbreviation** $\mathit{extend} :: (\mathit{nat} \Rightarrow \mathit{partial\text{-}pred\text{-}denot}) \Rightarrow \mathit{hterm\ pred\text{-}denot}$ **where**
> $\quad \mathit{extend}\ f\ P\ \mathit{ts} \equiv$
> $\quad\quad \mathit{let}\ n = \mathit{nat\text{-}of\text{-}hatom}\ (P, \mathit{ts})\ \mathit{in}$
> $\quad\quad\quad f\ (\mathit{Suc}\ n)\ !\ n$

Because of currying, P and ts can be thought of as the predicate symbol and list of values that we wish to evaluate in our semantics. It is done by collecting them to an Herbrand atom, and finding its index. Thereafter a prefix of our infinite path is found that is long enough to have decided whether the atom is considered *True* or *False*.

I now prove that if the prefixes collected in the infinite path f satisfy a set of clauses Cs, then so does its extension to a full predicate denotation $\mathit{extend}\ f$.

Since I want to prove that the clauses in Cs are satisfied, I fix one C and prove that it has the same property:

```
lemma extend-infpath:
  assumes wf-infpath (f :: nat ⇒ partial-pred-denot)
  assumes ∀n. ¬falsifies_c (f n) C
  assumes finite C
  shows eval_c HFun (extend f) C
```

There are four ways in which clauses can be satisfied:

1. A *first-order clause* can be satisfied by a *partial interpretation.*

2. A *ground clause* can be satisfied by a *partial interpretation.*

3. A *ground clause* can be satisfied by an *Herbrand interpretation.*

4. A *first-order clause* can be satisfied by an *Herbrand interpretation.*

The four ways are illustrated as the nodes in Fig. 3. The *extend-infpath* lemma relates 1 and 4 using lemmas that relate 1 to 2 to 3 to 4. The four ways seem similar, but they are in fact very different. For instance, a ground clause being satisfied is very different from a first-order clause being satisfied, since there are no ground instances or variables to worry about. Likewise, a ground clause being satisfied by a partial interpretation is clearly different from being satisfied by an Herbrand interpretation, since the two types are vastly different: A partial interpretation is a *bool list* while an Herbrand interpretation consists of a *fun-sym* $\Rightarrow$ *hterm list* $\Rightarrow$ *hterm* and a *pred-sym* $\Rightarrow$ *hterm list* $\Rightarrow$ *bool*.

1 and 2 are related: If a first-order clause is satisfied by all prefixes of an infinite path, then so is any, in particular ground, instance. This follows from the definition of being satisfied by a partial interpretation.

2 and 3 are related: If a ground clause is satisfied by all prefixes of an infinite path f, then it is also satisfied by *extend* f. This follows almost directly from the definition of *extend*.

3 and 4 are related: Ideally one would prove that if a ground clause is satisfied by an Herbrand interpretation, then so is a first-order clause of which it is an instance. That is, however, too general. Fortunately, there is a similarity that ties first-order clauses and ground clauses together. Consider a variable denotation in the Herbrand universe, i.e. of type *var-sym* $\Rightarrow$ *hterm*. There is a function that converts its domain to *fterm*s, and thus turns it in to a substitution:

```
fun sub-of-denot :: hterm var-denot ⇒ substitution
  sub-of-denot E = fterm-of-hterm ∘ E
```

This is the machinery necessary to state the needed lemma: If the ground clause $C \cdot_{ls}$ *sub-of-denot* E is satisfied by an Herbrand interpretation under E, then so is the first-order clause C. The reason is simply that any variable in C is replaced by some ground term in the domain of *sub-of-denot* E. This term evaluates to the same as the Herbrand term that it is interpreted as in E.

The final step is to chain 1, 2, 3, and 4 together to relate 1 and 4. The steps are shown as the arrows in Fig. 3.

1. Assume that C is satisfied by all prefixes of f.

2. Then the ground instance $C \cdot_{ls}$ *sub-of-denot* E is satisfied by all f's prefixes.

3. Then the ground instance $C \cdot_{ls}$ *sub-of-denot* E is satisfied by *extend* f under E in particular.

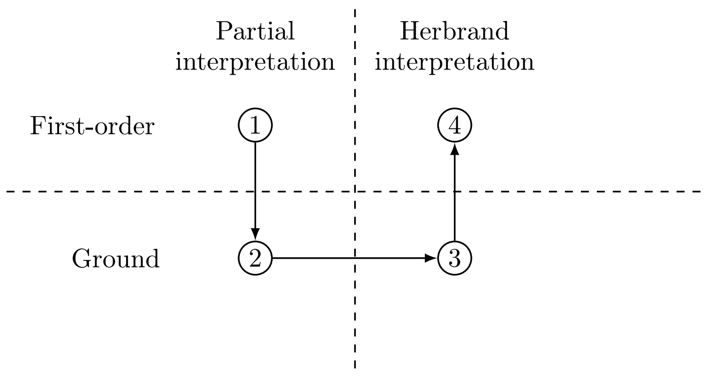


Figure 3: Illustration of how to go from satisfiability of first-order clauses in partial interpretations to their satisfiability in Herbrand interpretations. As shown, it can be done by going down to the ground level and up again.

4. Then C is satisfied by *extend* f under E.

With this, Herbrand's theorem is formalized:

theorem *herbrand:*
  assumes $\forall G.\ \neg \mathit{eval}_{\mathrm{cs}}\ \mathit{HFun}\ G\ \mathit{Cs}$
  assumes $\mathit{finite}\ \mathit{Cs} \land (\forall C \in \mathit{Cs}.\ \mathit{finite}\ C)$
  shows $\exists T.\ \mathit{closed\text{-}tree}\ T\ \mathit{Cs}$

The proof, as said, follows the sketch from Sect. 2.

8 Completeness

The completeness proof combines Herbrand's theorem, the lifting lemma, and reasoning about semantic trees and derivations. The purpose of this section is to take a look at the most challenging parts of the formalization of the proof. These are the lifting lemma, standardizing clauses apart, and some finer details of reasoning about branches in semantic trees. Furthermore, the section illustrates the derivation of the empty clause and shows a number of formal completeness theorems.

8.1 Lifting Lemma

Let us first take a look at the formalization of the lifting lemma. More precisely I will explain a flaw in proofs from the literature and present the formalization of a correct proof.

Let us look at the flawed proofs. The formalization of the resolution rule removes literals from clauses before it applies the MGU. This is similar to several presentations from the literature including those of Robinson [59] and Leitsch [43]. Another approach, which the formalization used in an earlier version, is to apply the MGU before the literals are removed:

$$\frac{C_1 \qquad\qquad C_2}{(C_1 \cdot_{\mathrm{ls}} \sigma - L_1 \cdot_{\mathrm{ls}} \sigma) \cup (C_2 \cdot_{\mathrm{ls}} \sigma - L_2 \cdot_{\mathrm{ls}} \sigma)} \quad \begin{array}{l} \mathit{vars}_{\mathrm{ls}}\ C_1 \cap \mathit{vars}_{\mathrm{ls}}\ C_2 = \{\} \\ L_1 \subseteq C_1,\ L_2 \subseteq C_2 \\ \sigma \text{ is an MGU of } L_1 \cup {L_2}^{\mathrm{C}} \end{array}$$

$$\begin{tikzcd}
L_1 \arrow[dr, dashed, "\tau"] \arrow[dd, "\eta"'] & & L_2^{\mathrm{C}} \arrow[dl, dashed, "\tau"'] \arrow[dd, "\eta"] \\
& L_1 \cdot_{\mathrm{ls}} \tau \arrow[dd, dashed, "\varphi"] & \\
L_1' \arrow[dr, "\sigma"] & & L_2'^{\mathrm{C}} \arrow[dl, "\sigma"'] \\
& L_1' \cdot_{\mathrm{ls}} \sigma &
\end{tikzcd}$$

Figure 4: The substitutions of the lifting lemma. An arrow from L to L' labeled with η indicates that $L \cdot_{\mathrm{ls}} \eta = L'$. Full arrows are relations we know. The dashed ones are established in the proof of the lemma by noticing that $\eta \cdot \sigma$ is a unifier of L_1 and L_2^C, which means we can obtain the MGU τ and by the definition of MGUs also φ.

This is exactly the rule used by Ben-Ari [4]. Chang and Lee [19] use a similar approach with more possibilities for factoring. However, I was not able to formalize their proofs of the lifting lemma because they had some flaws. The flaws are described in my master's thesis [62]. The most critical flaw is that the proofs seem to use that $B \subseteq A$ implies $(A - B) \cdot_{\mathrm{ls}} \sigma = A \cdot_{\mathrm{ls}} \sigma - B \cdot_{\mathrm{ls}} \sigma$, which does not hold in general. Leitsch [42, Proposition 4.1] noticed flaws in Chang and Lee's proof already, and presented a counter-example to it.

Let us now look at a formalization of a correct proof. With the current approach the lifting lemma is straightforward to formalize as an Isar proof following the proof by Leitsch [43]. The Isar proof is presented below. It consists of four parts. First we obtain the subsets L_1 and L_2 of C_1 and C_2 that we want to resolve upon. Next we obtain substitutions τ and φ as illustrated in Fig. 4. This is where we use the *unification* theorem to obtain τ. We can then construct the desired resolvent, and show that resolution is applicable.

To illustrate the correspondence between informal proofs and Isar proofs I present the whole Isar proof below, interleaved with an informal proof that expands on the sketch from Sect. 2. The reader should notice the similarities between formal and informal proof, but is not expected to understand all details of the formal proof. Notice also that we do not need to assume groundness of C_1' and C_2'.

Lemma: *Assume we have two finite clauses C_1 and C_2 that share no variables. Assume also that C_1' is an instance of C_1 and that C_2' is an instance of C_2. Furthermore, assume that the resolution rule is applicable to C_1' and C_2' on clashing sets of literals L_1' and L_2' with MGU σ. Then there exist sets of literals L_1 and L_2 and substitution τ such that the resolution rule is applicable to C_1 and C_2 on clashing sets of literals L_1 and L_2 with MGU τ and that their conclusion has the conclusion from C_1' and C_2' as an instance.*

lemma *lifting*:
  assumes *fin*: *finite* $C_1 \wedge$ *finite* C_2
  assumes *apart*: *vars*$_{\mathrm{ls}}$ $C_1 \cap$ *vars*$_{\mathrm{ls}}$ $C_2 = \{\}$
  assumes *inst*: *instance-of*$_{\mathrm{ls}}$ C_1' C_1 $\wedge$ *instance-of*$_{\mathrm{ls}}$ C_2' C_2
  assumes *appl*: *applicable* C_1' C_2' L_1' L_2' σ
  shows $\exists L_1\ L_2\ \tau.$ *applicable* $C_1\ C_2\ L_1\ L_2\ \tau \wedge$
        instance-of$_{\mathrm{ls}}$ (*resolution* $C_1'\ C_2'\ L_1'\ L_2'\ \sigma$) (*resolution* $C_1\ C_2\ L_1\ L_2\ \tau$)

proof –

– First we obtain the subsets to resolve upon:
Look at the clashing sets of literals L_1' and L_2'. We partition C_1' in L_1' and the rest, $R_1' = C_1' - L_1'$. Likewise, we partition C_2' in L_2' and the rest, $R_2' = C_2' - L_2'$. Since C_1' is an instance of C_1 there must be a substitution γ such that $C_1 \cdot_{\mathrm{ls}} \gamma = C_1'$. Likewise there must be a μ such that $C_2 \cdot_{\mathrm{ls}} \mu = C_2'$. Since the C_1 and C_2 share no variables, we can replace this with a single substitution η. We now partition C_1 in a part, L_1, that η instantiates to L_1' and a rest $C_1 - L_1$ that η instantiates to R_1'. We call the rest R_1. Likewise we obtain an L_2 which η instantiates to L_2' and an R_2 that η instantiates to R_2'.

define R_1' **where** $R_1' = C_1' - L_1'$
define R_2' **where** $R_2' = C_2' - L_2'$

from *inst* **obtain** γ μ **where** $C_1 \cdot_{\mathrm{ls}} \gamma = C_1' \wedge C_2 \cdot_{\mathrm{ls}} \mu = C_2'$
  unfolding *instance-of*$_{\mathrm{ls}}$*-def* **by** *auto*
then obtain η **where** *η-p*: $C_1 \cdot_{\mathrm{ls}} \eta = C_1' \wedge C_2 \cdot_{\mathrm{ls}} \eta = C_2'$
  using *apart merge-sub* **by** *force*

from *η-p* **obtain** L_1 **where** *L_1-p*: $L_1 \subseteq C_1 \wedge L_1 \cdot_{\mathrm{ls}} \eta = L_1' \wedge (C_1 - L_1) \cdot_{\mathrm{ls}} \eta = R_1'$
  using *appl project-sub* **using** *applicable-def R_1'-def* **by** *metis*
define R_1 **where** $R_1 = C_1 - L_1$
from *η-p* **obtain** L_2 **where** *L_2-p*: $L_2 \subseteq C_2 \wedge L_2 \cdot_{\mathrm{ls}} \eta = L_2' \wedge (C_2 - L_2) \cdot_{\mathrm{ls}} \eta = R_2'$
  using *appl project-sub* **using** *applicable-def R_2'-def* **by** *metis*
define R_2 **where** $R_2 = C_2 - L_2$

– Then we obtain their MGU:
We assumed that resolution is applicable on clashing sets of literals L_1' and L_2' with MGU σ. Therefore σ is an MGU of $L_1' \cup L_2'^{\mathrm{C}}$ which is the same as it being an MGU of $(L_1 \cdot_{\mathrm{ls}} \eta) \cup (L_2 \cdot_{\mathrm{ls}} \eta)^{\mathrm{C}}$ which again is the same as it being an MGU of $(L_1 \cup L_2{}^{\mathrm{C}}) \cdot_{\mathrm{ls}} \eta$. Thus σ is a unifier of $(L_1 \cup L_2{}^{\mathrm{C}}) \cdot_{\mathrm{ls}} \eta$, and therefore $\eta \cdot \sigma$ is a unifier of $L_1 \cup L_2{}^{\mathrm{C}}$. By the unification theorem there must also be an MGU τ of $L_1 \cup L_2{}^{\mathrm{C}}$, and by the definition of it being an MGU there must also be a substitution φ such that $\tau \cdot \varphi = \eta \cdot \sigma$.

from *appl* **have** *mgu*$_{\mathrm{ls}}$ σ $(L_1' \cup L_2'^{C})$
  using *applicable-def* **by** *auto*
then have *mgu*$_{\mathrm{ls}}$ σ $((L_1 \cdot_{\mathrm{ls}} \eta) \cup (L_2 \cdot_{\mathrm{ls}} \eta)^{C})$
  using *L_1-p L_2-p* **by** *auto*
then have *mgu*$_{\mathrm{ls}}$ σ $((L_1 \cup L_2{}^{C}) \cdot_{\mathrm{ls}} \eta)$
  using *compls-subls subls-union* **by** *auto*
then have *unifier*$_{\mathrm{ls}}$ σ $((L_1 \cup L_2{}^{C}) \cdot_{\mathrm{ls}} \eta)$
  using *mgu*$_{\mathrm{ls}}$*-def* **by** *auto*
then have *$\eta\sigma$uni*: *unifier*$_{\mathrm{ls}}$ $(\eta \cdot \sigma)$ $(L_1 \cup L_2{}^{C})$
  using *unifier*$_{\mathrm{ls}}$*-def composition-conseq2l* **by** *auto*
then obtain τ **where** *τ-p*: *mgu*$_{\mathrm{ls}}$ τ $(L_1 \cup L_2{}^{C})$
  using *unification fin L_1-p L_2-p* **by** (*meson finite-UnI finite-imageI rev-finite-subset*)
then obtain φ **where** *φ-p*: $\tau \cdot \varphi = \eta \cdot \sigma$
  using *$\eta\sigma$uni mgu*$_{\mathrm{ls}}$*-def* **by** *auto*

– We show that we have the desired conclusion:
Define C as $((C_1 - L_1) \cup (C_2 - L_2)) \cdot_{\mathrm{ls}} \tau$, i.e. the resolvent of C_1 and C_2 on clashing sets of literals L_1 and L_2 with MGU τ. Let us see what φ instantiates it to:
$C \cdot_{\mathrm{ls}} \varphi = (R_1 \cup R_2) \cdot_{\mathrm{ls}} (\tau \cdot \varphi)$ – by the definitions of C, R_1, and R_2.
$= (R_1 \cup R_2) \cdot_{\mathrm{ls}} (\eta \cdot \sigma)$ – since these two composed substitutions were equal.
$= ((R_1 \cdot_{\mathrm{ls}} \eta) \cup (R_2 \cdot_{\mathrm{ls}} \eta)) \cdot_{\mathrm{ls}} \sigma$

$= (R_1' \cup R_2') \cdot_{\mathrm{ls}} \sigma$ – by the definitions of R_1' and R_2'.
In conclusion $C \cdot_{\mathrm{ls}} \varphi = ((C_1' - L_1') \cup (C_2' - L_2')) \cdot_{\mathrm{ls}} \sigma$, i.e. the conclusion from C_1 and C_2 has the one from C_1' and C_2' as an instance.

define C **where** $C = ((C_1 - L_1) \cup (C_2 - L_2)) \cdot_{\mathrm{ls}} \tau$
have $C \cdot_{\mathrm{ls}} \varphi = (R_1 \cup R_2) \cdot_{\mathrm{ls}} (\tau \cdot \varphi)$
  using *subls-union composition-conseq2ls* **using** *C-def* R_1*-def* R_2*-def* **by** *auto*
also have ... $= (R_1 \cup R_2) \cdot_{\mathrm{ls}} (\eta \cdot \sigma)$
  using φ*-p* **by** *auto*
also have ... $= ((R_1 \cdot_{\mathrm{ls}} \eta) \cup (R_2 \cdot_{\mathrm{ls}} \eta)) \cdot_{\mathrm{ls}} \sigma$
  using *subls-union composition-conseq2ls* **by** *auto*
also have ... $= (R_1' \cup R_2') \cdot_{\mathrm{ls}} \sigma$
  using η*-p* L_1*-p* L_2*-p* **using** R_1*-def* R_2*-def* **by** *auto*
finally have $C \cdot_{\mathrm{ls}} \varphi = ((C_1' - L_1') \cup (C_2' - L_2')) \cdot_{\mathrm{ls}} \sigma$
  unfolding R_1'*-def* R_2'*-def* **by** *auto*
then have *ins*: *instance-of*$_{\mathrm{ls}}$ (*resolution* C_1' C_2' L_1' L_2' σ) (*resolution* C_1 C_2 L_1 L_2 τ)
  using *resolution-def instance-of*$_{\mathrm{ls}}$*-def C-def* **by** *metis*

– We show that the resolution rule is applicable:
We know that the resolution rule was applicable on C_1' and C_2' with clashing sets of literals L_1' and L_2'. Therefore these sets must be non-empty. Since they are instances of C_1', C_2', L_1', and L_2', these must also be non-empty. We have already established all other conditions of resolution being applicable.
This concludes the proof.

have $C_1' \neq \{\} \wedge C_2' \neq \{\} \wedge L_1' \neq \{\} \wedge L_2' \neq \{\}$
  using *appl applicable-def* **by** *auto*
then have $C_1 \neq \{\} \wedge C_2 \neq \{\} \wedge L_1 \neq \{\} \wedge L_2 \neq \{\}$
  using η*-p* L_1*-p* L_2*-p* **by** *auto*
then have *appli*: *applicable* C_1 C_2 L_1 L_2 τ
  using *apart* L_1*-p* L_2*-p* τ*-p* *applicable-def* **by** *auto*

from *ins appli* **show** *?thesis*
  by *auto*
qed

8.2 The Formal Completeness Proof

Like Herbrand's theorem, I formalize completeness as an Isar proof following Chang and Lee [19]. This time, however, the proof is much longer than its informal counterpart. The paper proof is about 30 lines, while the formal proof is approximately 150 lines. There are several reasons for this:

1. Clauses have to be explicitly standardized apart.
2. The clauses falsified by branches ending in two sibling leaves must be resolved and the sibling leaves must be cut off.
3. Even more of the tree must be cut off to minimize it.
4. The derivation-steps must be tied together.

We need to prove that the cut tree is closed. Furthermore, cutting the tree requires very precise reasoning about the numbers of the ground atoms. In the following subsection I tackle 1, 2 and 4 which are particularly interesting.

Let us now look at the completeness proof from a high level to choose an appropriate induction principle. The completeness proof consists of two steps. First Herbrand's theorem is applied to obtain a finite tree. Next the finite tree is cut smaller and a derivation step is made. Then the process is repeated on the smaller tree. To prove that this works, I formalize the process using induction on the size of the tree. The formalization uses the induction rule *measure_induct_rule* instantiated with the size of a tree. This gives the following induction principle:

lemma
  assumes $\bigwedge x.\ (\bigwedge y.\ \mathit{treesize}\ y < \mathit{treesize}\ x \Longrightarrow P\ y) \Longrightarrow P\ x$
  shows $P\ a$

Here, the induction hypothesis holds for any tree of a smaller size, and this is needed since several nodes are cut off in each step.

In order for the completeness theorem to fit with the above induction principle, it is first formulated in an appropriate way, assuming the existence of a closed semantic tree. I show this formulation along with a sketch of the inductive Isar proof:

theorem *completeness'*:
  assumes *closed-tree T Cs*
  assumes $\forall C{\in}Cs.\ \mathit{finite}\ C$
  shows $\exists Cs'.\ \mathit{resolution\text{-}deriv}\ Cs\ Cs' \wedge \{\} \in Cs'$
using *assms* **proof** (*induction T arbitrary*: *Cs rule*: *measure-induct-rule*[*of treesize*])
  fix T :: *tree*
  fix Cs :: *fterm clause set*
  assume *ih*: $\bigwedge T'\ Cs.\ \mathit{treesize}\ T' < \mathit{treesize}\ T \Longrightarrow \mathit{closed\text{-}tree}\ T'\ Cs \Longrightarrow$
      $\forall C{\in}Cs.\ \mathit{finite}\ C \Longrightarrow \exists Cs'.\ \mathit{resolution\text{-}deriv}\ Cs\ Cs' \wedge \{\} \in Cs'$
  assume *clo*: *closed-tree T Cs*
  assume *finite-Cs*: $\forall C{\in}Cs.\ \mathit{finite}\ C$
  ⋮
  ultimately show $\exists Cs'.\ \mathit{resolution\text{-}deriv}\ Cs\ Cs' \wedge \{\} \in Cs'$ **by** *auto*
qed

An alternative approach would have been to use an induction on the subtree relationship.

8.3 Standardizing Apart

In each step the resolved clauses must be standardized apart. Two functions can do this:

abbreviation $\mathit{std}_1\ C \equiv C \cdot_{\mathrm{ls}} (\lambda x.\ \mathit{Var}\ (''1''\ @\ x))$

abbreviation $\mathit{std}_2\ C \equiv C \cdot_{\mathrm{ls}} (\lambda x.\ \mathit{Var}\ (''2''\ @\ x))$

They take clauses C_1 and C_2 and create the clauses $\mathit{std}_1\ C_1$ and $\mathit{std}_2\ C_2$ which have added respectively 1 and 2 to the beginning of all variables. The most important property is that the clauses actually have distinct variables after the functions are applied. We need this such that we can apply the resolution rule, and so we can use the lifting lemma.

lemma *std-apart-apart:* $\mathit{vars}_{\mathrm{ls}}\ (\mathit{std}_1\ C_1) \cap \mathit{vars}_{\mathrm{ls}}\ (\mathit{std}_2\ C_2) = \{\}$

I prove that the functions actually rename the variables. This was a prerequisite for the standardize apart rule of the calculus.

lemma std_1*-renames:* $\mathit{var\text{-}renaming\text{-}of}\ C_1\ (\mathit{std}_1\ C_1)$

In the completeness proof I need that C_1 and $\mathit{std}_1\ C_1$ are falsified by the same partial interpretations:

lemma std_1*-falsifies:* $\mathit{falsifies}_{\mathrm{c}}\ G\ C_1 \longleftrightarrow \mathit{falsifies}_{\mathrm{c}}\ G\ (\mathit{std}_1\ C_1)$

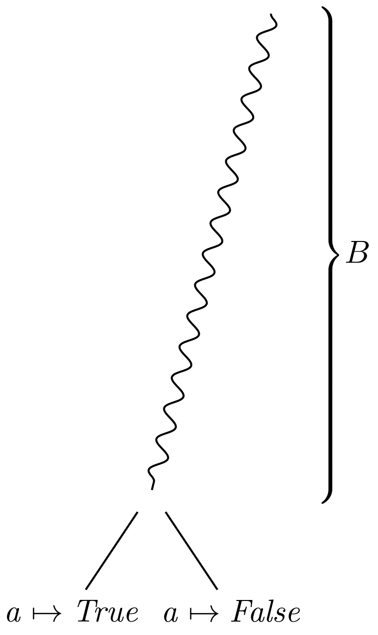


Figure 5: B is a path from the root of a semantic tree to a parent of two sibling nodes. B_1 extends B by going left and B_2 by going right. B falsifies no clause in our set of clauses, but B_1 falsifies C_1, and B_2 falsifies C_2.

8.4 Resolving Falsified Clauses

Let us now look at how to formalize the removal of two sibling leaves, and why the clauses that their branches falsified can be resolved. In each step, the completeness proof removes two sibling leaves. The branches, $B_1 = B \mathbin{@} [\mathit{True}]$ and $B_2 = B \mathbin{@} [\mathit{False}]$, ending in these sibling leaves falsified a first-order clause each, C_1 and C_2. By the definition of falsification of first-order clauses, B_1 and B_2 falsified ground instances C_1' and C_2' of C_1 and C_2 respectively. These ground clauses are then resolved, and the resolvent is falsified by B. This is then lifted to the first-order level using the lifting lemma. See the situation in Fig. 5.

Thus, on the ground level, two properties must be established:

1. The two ground clauses C_1' and C_2' falsified by B_1 and B_2 can be resolved.
2. Their ground resolvent C' is falsified by B. This ensures that the tree is closed when we cut off B_1 and B_2 and minimize it.

Let us prove 1 first. This is done by proving that C_1' contains the negative literal $l = \mathit{Neg}\ a$ of number *length* B in the enumeration, and that C_2' contains its complement. Here, the case for C_1' is presented. C_1' is falsified by B_1, but not B, because the closed semantic tree is minimal. Thus, it must be the decision of going left that was necessary to falsify C_1'. Going left falsified the negative literal l with number *length* B in the enumeration, and hence it must be in C_1'.

Let us prove 2 next. To prove it we must show that the ground resolvent $C' = (C_1' - \{l\}) \cup (C_2' - \{l^c\})$ is falsified by B. We do it by proving that the literals in both $C_1' - \{l\}$ and $C_2' - \{l^c\}$ are falsified. The case for $C_1' - \{l\}$ is presented here. The overall idea is that l is falsified by B_1, but not by B. The decision of going left falsified l, and then all of C_1' was falsified. Therefore, the other literals must have been falsified before we made the decision, in other words, they must have been falsified already by B.

To formalize this we must prove that all the literals in $C_1' - \{l\}$ are indeed falsified by B. We do it by a lemma showing that any other literal $lo \in C_1'$ than l is falsified by B. Its proof first shows that lo has another number than l has, i.e. other than *length* B. It seems obvious since $lo \neq l$, but we also need to ensure that $lo \neq l^c$. We do this by proving another lemma, which says that a clause only can be falsified by a partial interpretation if it does not contain two complementary literals. Then we show that lo has a number smaller than *length* (B @ [*True*]), since lo is falsified by B @ [*True*]. This concludes the proof.

I abstract from *True* to d such that the lemma also will work for the path B @ [*False*] that goes left:

lemma *other-falsified:*
  assumes $\mathit{ground}_{\mathrm{ls}}\ C_1' \wedge \mathit{falsifies}_{\mathrm{g}}\ (B\ @\ [d])\ C_1'$
  assumes $l \in C_1' \wedge \mathit{nat\text{-}of\text{-}fatom}\ (\mathit{get\text{-}atom}\ l) = \mathit{length}\ B$
  assumes $lo \in C_1' \wedge lo \neq l$
  shows $\mathit{falsifies}_{\mathrm{l}}\ B\ lo$

8.5 The Derivation

At the end of the proof the derivations are tied together:

$$\dfrac{\dfrac{\dfrac{C_1}{\mathit{std}_1\ C_1} \qquad \dfrac{C_2}{\mathit{std}_2\ C_2}}{\mathit{resolution}\ C_1\ C_2\ L_1\ L_2\ \sigma}}{\begin{array}{c}\vdots\\ \hline \{\}\end{array}}$$

The dots represent the derivation we obtain from the induction hypothesis. It is done using the definitions of *resolution-step* and *resolution-deriv*. From *herbrand* and *completeness′* follows the completeness theorem:

theorem *completeness:*
  assumes $\mathit{finite}\ \mathit{Cs} \wedge (\forall C \in \mathit{Cs}.\ \mathit{finite}\ C)$
  assumes $\forall (F :: \mathit{hterm\ fun\text{-}denot})\ (G :: \mathit{hterm\ pred\text{-}denot}).\ \neg \mathit{eval}_{\mathrm{cs}}\ F\ G\ \mathit{Cs}$
  shows $\exists \mathit{Cs}'.\ \mathit{resolution\text{-}deriv}\ \mathit{Cs}\ \mathit{Cs}' \wedge \{\} \in \mathit{Cs}'$

8.6 Further Completeness Theorems

Let us now look at the strength of the above completeness proof and consider several other variants.

Notice that the above completeness theorem is actually stronger than the usual one. Usually, the assumption would consider all interpretations of all universes. Here, however, the assumption is weakened to consider only all interpretations of the Herbrand universe.

It could be illustrative to also formalize the usual formulation, but unfortunately, because of my choice of representing universes by types this is not possible. The reason is that although all statements in HOL are implicitly universally quantified over all types at the top, we are not allowed to do type quantification explicitly inside HOL formulas.

I instead prove some other instructive formulations of the theorem. For the completeness proof it was central that we considered the Herbrand universe, but for the theorem it is actually not important. The Herbrand universe can be replaced by any countably infinite universe. To prove this we fix an arbitrary countably infinite universe and obtain a bijection between it and the Herbrand terms. Three functions are defined that can apply the bijection to respectively variable denotations, function denotations, and predicate denotations:

> **definition** $\textit{E-conv} :: ('a \Rightarrow 'b) \Rightarrow 'a\ \textit{var-denot} \Rightarrow 'b\ \textit{var-denot}$ **where**
> $\textit{E-conv b-of-a E} \equiv \lambda x.\ (\textit{b-of-a}\ (E\ x))$

> **definition** $\textit{F-conv} :: ('a \Rightarrow 'b) \Rightarrow 'a\ \textit{fun-denot} \Rightarrow 'b\ \textit{fun-denot}$ **where**
> $\textit{F-conv b-of-a F} \equiv \lambda f\ bs.\ \textit{b-of-a}\ (F\ f\ (\textit{map}\ (\textit{inv b-of-a})\ bs))$

> **definition** $\textit{G-conv} :: ('a \Rightarrow 'b) \Rightarrow 'a\ \textit{pred-denot} \Rightarrow 'b\ \textit{pred-denot}$ **where**
> $\textit{G-conv b-of-a G} \equiv \lambda p\ bs.\ (G\ p\ (\textit{map}\ (\textit{inv b-of-a})\ bs))$

Proving some appropriate lemmas about these functions I arrive at the following completeness theorem:

> **theorem** *completeness-countable:*
> **assumes** $\textit{infinite}\ (\textit{UNIV} :: ('u :: \textit{countable})\ \textit{set})$
> **assumes** $\textit{finite}\ Cs \wedge (\forall C \in Cs.\ \textit{finite}\ C)$
> **assumes** $\forall (F :: 'u\ \textit{fun-denot})\ (G :: 'u\ \textit{pred-denot}).\ \neg \textit{eval}_{\text{cs}}\ F\ G\ Cs$
> **shows** $\exists Cs'.\ \textit{resolution-deriv}\ Cs\ Cs' \wedge \{\} \in Cs'$

In particular, I use it to replace the Herbrand universe with the universe of the natural numbers:

> **theorem** *completeness-nat:*
> **assumes** $\textit{finite}\ Cs \wedge (\forall C \in Cs.\ \textit{finite}\ C)$
> **assumes** $\forall (F :: \textit{nat fun-denot})\ (G :: \textit{nat pred-denot}).\ \neg \textit{eval}_{\text{cs}}\ F\ G\ Cs$
> **shows** $\exists Cs'.\ \textit{resolution-deriv}\ Cs\ Cs' \wedge \{\} \in Cs'$

9 Discussion

Since this paper is a case study in formalizing mathematics, it is worthwhile considering which tools were helpful in this regard. This section discusses each of these tools. The section also gives an idea of how much work went in to making the formalization. It discusses the consequences of the choice of representing universes as types. Lastly, it discusses the applicability of the formalization to implemented automated theorem provers.

Integrated Development Environments (IDEs) help their users do software development. Isabelle includes the Isabelle/jEdit Prover IDE, which has many useful features for navigating, reading, and writing proof documents. For instance it reveals type information of constants when the user hovers the mouse cursor over them. The user can click on any constant or type to jump to its definition and with another click she can jump back again. These features were especially advantageous when the theory grew larger. This is not only useful when writing proofs but also when reading them. In a formalization, every word is formally tied to its definition, so if at some point I forget the meaning of some expression the definitions are available by the click of a button. In my opinion this corroborates the claim that formal companions to paper proofs are highly useful.

The structured proof language Isar was beneficial because it allows formal proofs to be written as sequences of claims that follow from the previous claims. This clearly mirrors mathematical paper proof, which is what I am formalizing. Furthermore, it makes the proofs easy to read, and this is important when a formalization is to help in the understanding of a theory.

Isabelle/HOL includes several generic proof methods or tactics that can discharge proof goals. Writing a proof in Isabelle/HOL is a process of stating the formula you think holds and showing this from the previous statements with the right proof method. The *simp* and *auto* methods do rewriting and more while, e.g. blast and metis are first-order automatic theorem provers. Knowing which one to use in a given situation is a matter of knowledge of how the prover works, of experience, and of trial and error.

The Sledgehammer tool [6] finds proofs by picking important facts from the theory and then employing top-of-the-line automatic theorem provers and satisfiability modulo theory solvers. It often helps proving claims that we know are true, but where finding the necessary facts from the theory and libraries as well as choosing and instructing a proof method would be tedious.

It is also worthwhile considering how much work went into the formalization. The whole development is about 3300 lines of code. A preliminary version of the theory was developed during 5 months as part of my master's thesis [62] including formalizations of clausal logic, its semantics, unifiers, a resolution calculus, its soundness, Herbrand interpretations, semantic trees and Herbrand's theorem. I developed the rest of the theory during the first 5 months of my PhD studies concurrently with my other duties. The completeness theorems in Subsect. 8.6 were formalized with little work while writing this extended paper. The lifting lemma was the greatest challenge because of the flaws in Chang and Lee's proof. As soon as I looked at the proof by Leitsch, it was straightforward to finish.

In Sect. 4, I chose to represent universes as sets. The advantage of this was that I did not need additional predicates to restrict the ranges of variable and function denotations to stay within the fixed universe, since this was captured in the types. This is rather convenient, since then the proofs are not cluttered with reasoning about these predicates. On the other hand, in Sect. 7, I needed to introduce a type for the Herbrand universe, where it could otherwise have been captured directly as the set of ground terms. In Sect. 8, we also saw that a consequence was that we could not express completeness in its usual formulation, but had to go with a stronger formulation. To sum up, by formalizing universes as types rather than sets, I gained convenience, but lost some expressibility.

Finally, it is worthwhile considering the applicability of the formalization to implementations of automated theorem provers such as E, SPASS, and Vampire. Such automatic theorem provers consist of a calculus and a function to construct proofs in the calculus. The present formalization is purely of a calculus. Furthermore, the mentioned provers use the superposition calculus, which is an extension of resolution to first-order logic with equality. Resolution and superposition coincide for first-order logic without equality. The rules of superposition have several side conditions which only serve to rule out unnecessary inferences while the resolution rule I formalize has no such side conditions.

10 Related Work

The literature describes several formalizations of logic. This section takes a look at formalizations of both intuitionistic and classical first-order logic. Furthermore, it looks at two results that go beyond first-order logic. Lastly, it gives an overview of the IsaFoL project, which is an effort to bring together researchers of formalizations of logic.

10.1 Formalizations of Proof Systems for First-Order Logic

The completeness of first-order logic is a landmark of logic and thus formalizing this theorem is interesting in itself. Natural deduction calculi and sequent calculi are very suited for this purpose because of their simplicity.

Persson [54] formalized, in ALF, intuitionistic first-order logic. He formalized an intuitionistic natural deduction system and an intuitionistic sequent calculus. The semantics are defined using topology, which is a generalization of the semantics for classical first-order logic. He formalized both natural deduction, sequent calculi, and an axiomatic system. He proved the natural deduction systems and the sequent calculus sound, and proved the natural deduction with named variables complete. Ilik [31] also formalized, in Coq, natural deduction for intuitionistic first-order logic. He proved it complete with respect to a Kripke semantics. He also studied properties of intuitionistic logic extended with delimited control operators known from programming languages.

Harrison [28] formalized, in HOL Light, model theoretic results about classical first-order logic, including the compactness theorem, the Löwenheim-Skolem theorem, and Herbrand's theorem.

Moreover, there are several formalized completeness proofs for classical first-order natural deduction. Berghofer [5] formalized natural deduction, in Isabelle/HOL, and proved it sound and complete. He also proved the Löwenheim-Skolem theorem. Raffalli [55] proved, in Phox, natural deduction complete for first-order logic. His semantics is that of minimal models. These are similar to the sets of formulas true in the usual semantics, but behave differently with respect to negation. The completeness statement is equivalent to the one with respect to the usual semantics, but this is not formalized. Ilik [31] formalized, essentially, the same result in Coq, although less abstractly.

Other authors formalized completeness proofs for classical first-order sequent calculi. Margetson and Ridge [45] formalized, in Isabelle/HOL, a sequent calculus. Their syntax is that of formulas on negation normal form without first-order functions. They proved the calculus sound and complete with respect to a semantics on this syntax. Braselmann and Koepke [14,15] proved, in Mizar, a sequent calculus for first-order logic sound and complete. Schlöder and Koepke [67] proved it complete even for uncountable languages. Ilik [31] also proved a sequent calculus complete with respect to a Kripke-style semantics that he, Lee, and Herbelin [32] introduced for classical first-order logic.

Many completeness proofs follow similar recipes. Blanchette, Popescu, and Traytel [9, 12] formalized one such recipe, in Isabelle/HOL, as an abstract completeness proof for first-order logic that is independent of syntax and proof system. An interesting aspect of the proof is that it uses codatatypes to define and reason about infinite derivation trees. Their abstract completeness theorem states that if a proof system has a number of fairness properties, then it is complete in the following abstract sense: Any formula can either be proved or there exists some infinite path in a fair derivation tree of the formula. This means that the user of their formalization has three things to do in order to get a concrete completeness proof. First she needs to define a syntax, second she needs to define a fair proof system and third, she has to interpret the infinite paths as countermodels. The authors performed this step for a sequent calculus for first-order logic with equality and sorts. My formalization does not follow this recipe, opting instead for formalizing semantic trees. Blanchette, Popescu, and Traytel [11, 12] also formalized abstract soundness results and used them to prove a certain kind of infinite proofs correct.

Breitner and Lohner [17] defined natural deduction in an abstract way that is independent of syntax and the concrete rules of the system. They then used the abstract completeness proof by Blanchette, Popescu, and Traytel to prove it complete in the abstract sense. They also defined a novel graph representation of proofs, which is also independent of syntax and rules. They proved that any natural deduction system and the corresponding graph representation can prove the same theorems. Thus the graph representation is as sound and complete as the corresponding natural deduction system. They concretely instantiated it with a propositional logic with only conjunction and implication as well as a first-order logic with only universal quantification and implication. Breitner [16] used their graph representation to implement a tool for teaching logic.

For automatic theorem provers, it is not only important that the calculus is complete, but also that it can be implemented as a program. Ridge and Margetson [57, 58] verified a prover based on their formalized sequent calculus. Since their calculus does not contain full first-order terms, it means that they do not need any machinery such as MGUs to handle them. They also implement the prover as an OCaml program.

In his master's thesis, Gebhard [26] formalized several ground resolution calculi in the ΩMEGA proof assistant. A version of this development is available online in The Theorem Prover Museum [37]. Gebhard proved completeness using induction on the excess literal number. The excess literal number is the number of occurrences of literals in a set of clauses minus the number of clauses. The technique was introduced by Anderson and Bledsoe [1] who used it to prove a linear format for resolution complete. Arguably, semantic trees are a more pedagogical construction since they so naturally express interpretations, and therefore I prefer them. Furthermore, Goubault-Larrecq and Jouannaud [27] showed that semantic trees can actually be used to prove many of the refinements of resolution complete – including linear resolution. Another difference from my formalization is that Gebhard uses proof planning. Proof plans were introduced by Bundy [18] as formal specifications of LCF-style tactics, which are functions that can replace a goal in a proof with zero or more new subgoals. I instead made structured proofs in the declarative Isar language, which allowed me to write humanly readable proofs that can be checked by the Isabelle/HOL proof assistant.

Concurrent with this Isabelle/HOL formalization of resolution, an important step in formalizing automatic theorem provers for first-order logic was taken. Peltier proved propositional resolution [52] and a variant of the superposition calculus for first-order logic [53] sound and complete. The superposition calculus can be seen as a highly efficient generalization of resolution for first-order logic to first-order logic with equality. Therefore his formalization is representative of the state of the art in formalizing the theory of automatic theorem proving.

10.2 Beyond Completeness of First-Order Logic

There are also results that go beyond completeness of first-order logic. An early such result is Shankar's formalization [70, 71], in Nqthm, of Gödel's first incompleteness theorem. Raffalli [55] proved, in Phox, parts of the second incompleteness theorem. His proof is very abstract and thus relies on strong assumptions about codings of formulas. He does not provide an explicit coding and thus does not prove these assumptions. Paulson [49–51] did not take any shortcuts and managed to formalize the entirety of both Gödel's incompleteness theorems with a concrete coding of formulas based on hereditarily finite set theory.

Harrison [29] proved the soundness and consistency of HOL Light. He did this in two ways. First, he added an extra axiom to HOL Light that assumes the existence of a very large cardinal, and with this he was able to prove the unaltered HOL Light sound and consistent. Secondly, in unaltered HOL Light he proved the soundness and consistency of HOL Light altered by removing its axiom of infinity. Kumar, Arthan, Myreen, and Owens [39] extended Harrison's result by proving, in HOL4, soundness and consistency of HOL Light with definitions. Their approach is a bit different from Harrison's. Instead of adding an axiom describing a large cardinal to HOL4, their soundness proof assumes a specification of set theory. Additionally, they synthesize a verified implementation of the inference rules of their definition of HOL Light. A similar result is Davis and Myreen's soundness proof [23], in HOL4, of the Milawa theorem prover.

10.3 IsaFoL

This formalization is part of IsaFoL [33], the Isabelle Formalization of Logic, which is a project that brings together researchers of logic from many institutions. In the project we aim to develop libraries of lemmas and methods for formalizing research on logic. In addition to this formalization of logic, several other results have emerged from the project.

This paper is an example of IsaFoL catching up with classical unformalized results in Isabelle/HOL. Likewise, Jensen, Villadsen, and I [35, 36] formalized an axiomatic system that forms the kernel of a proof assistant for first-order logic with equality by Harrison [30]. In addition to the advantages of having a formal companion to Harrison's chapter on the proof assistant, the formalization also enabled us to build a certified prover based on the calculus. Other efforts in this direction are the work due to Blanchette, Traytel, Waldmann, and me [65] on a formalization of a resolution prover for first-order logic, as well as the formalizations of many ground calculi including SAT solvers and propositional resolution due to Blanchette, Fleury, and Weidenbach [7].

Additionally we develop new results in conjunction with formalizing them. Several term-orders have been formalized by Becker, Blanchette, Waldmann, and Wand [3] as well as Blanchette, Waldmann, and Wand [13]. These could serve as a basis for a higher-order superposition calculus. Villadsen and I [66] formalized a propositional paraconsistent logic with infinitely many truth values. Lammich wrote and verified a program that checks the certificates of satisfiability and unsatisfiability that SAT solvers can generate [40, 41].

11 Conclusion

This paper describes a formalization of the resolution calculus for first-order logic as well as its soundness and completeness. This includes formalizations of the substitution lemma, Herbrand's theorem, and the lifting lemma. As far as I know, this is the first formalized soundness and completeness proof of the resolution calculus for first-order logic.

The paper emphasizes how the formalization illustrates details glossed over in the paper proofs. Such details are necessary in a formalization. For instance it shows the jump from satisfiability by an infinite path in a semantic tree to satisfiability by an interpretation. It likewise illustrates how and when to standardize clauses apart in the completeness proof, and the lemmas necessary to allow this. Furthermore, the formalization combines theory from different sources. The proofs of Herbrand's theorem and completeness are based mainly on those by Chang and Lee [19], while the proof of the lifting lemma is based on that by Leitsch [43]. The existence proof of MGUs for unifiable clauses comes from IsaFoR [34].

The formalization is part of the IsaFoL project [33] on formalizing logics. When the project was started in 2015, we hoped it would attract other researchers to join and formalize their results by using and extending the library. It seems that we have had success with this since the number of authors in the project has more than tripled since then.

Proof assistants take advantage of automatic theorem provers by using them to prove subgoals. This formalization is a step towards mutual benefit between the two areas of research. Formalizations in proof assistants can help automatic theorem provers by contributing a highly rigorous understanding of their meta-theory.

Acknowledgements

I would like to thank Jørgen Villadsen, Jasmin Christian Blanchette, and Dmitriy Traytel who supervised me in making the formalization. I would also like to thank Jørgen, Jasmin, John

Bruntse Larsen, Andreas Halkjær From, and the anonymous referees for their valuable feedback on the paper.

References

[1] R. Anderson and W. W. Bledsoe. A linear format for resolution with merging and a new technique for establishing completeness. *Journal of the ACM*, 17(3):525–534, 1970.

[2] F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

[3] H. Becker, J. C. Blanchette, U. Waldmann, and D. Wand. Formalization of Knuth–Bendix orders for lambda-free higher-order terms. *Archive of Formal Proofs*, Nov. 2016. `http://isa-afp.org/entries/Lambda_Free_KBOs.shtml`, Formal proof development.

[4] M. Ben-Ari. *Mathematical Logic for Computer Science.* Springer, 3rd edition, 2012.

[5] S. Berghofer. First-order logic according to Fitting. *Archive of Formal Proofs*, Aug. 2007. `http://isa-afp.org/entries/FOL-Fitting.shtml`, Formal proof development.

[6] J. C. Blanchette, S. Böhme, and L. C. Paulson. Extending Sledgehammer with SMT solvers. *Journal of Automated Reasoning*, 51(1):109–128, 2013.

[7] J. C. Blanchette, M. Fleury, and C. Weidenbach. A verified SAT solver framework with learn, forget, restart, and incrementality. In N. Olivetti and A. Tiwari, editors, *IJCAR 2016*, volume 9706 of *LNCS*, pages 25–44. Springer, 2016.

[8] J. C. Blanchette, J. Hölzl, A. Lochbihler, L. Panny, A. Popescu, and D. Traytel. Truly modular (co)datatypes for Isabelle/HOL. In G. Klein and R. Gamboa, editors, *ITP 2014*, volume 8558 of *LNCS*, pages 93–110. Springer, 2014.

[9] J. C. Blanchette, A. Popescu, and D. Traytel. Abstract completeness. *Archive of Formal Proofs*, Apr. 2014. `http://isa-afp.org/entries/Abstract_Completeness.shtml`, Formal proof development.

[10] J. C. Blanchette, A. Popescu, and D. Traytel. Foundational extensible corecursion: A proof assistant perspective. In K. Fisher and J. Reppy, editors, *ICFP'15*, pages 192–204. ACM, 2015.

[11] J. C. Blanchette, A. Popescu, and D. Traytel. Abstract soundness. *Archive of Formal Proofs*, Feb. 2017. `http://isa-afp.org/entries/Abstract_Soundness.shtml`, Formal proof development.

[12] J. C. Blanchette, A. Popescu, and D. Traytel. Soundness and completeness proofs by coinductive methods. *Journal of Automated Reasoning*, 58(1):149–179, 2017.

[13] J. C. Blanchette, U. Waldmann, and D. Wand. Formalization of recursive path orders for lambda-free higher-order terms. *Archive of Formal Proofs*, Sept. 2016. `http://isa-afp.org/entries/Lambda_Free_RPOs.shtml`, Formal proof development.

[14] P. Braselmann and P. Koepke. Gödel's completeness theorem. *Formalized Mathematics*, 13(1):49–53, 2005.

[15] P. Braselmann and P. Koepke. A sequent calculus for first-order logic. *Formalized Mathematics*, 13(1):33–39, 2005.

[16] J. Breitner. Visual theorem proving with the Incredible Proof Machine. In J. C. Blanchette and S. Merz, editors, *ITP 2016*, volume 9807 of *LNCS*, pages 123–139. Springer, 2016.

[17] J. Breitner and D. Lohner. The meta theory of the Incredible Proof Machine. *Archive of Formal Proofs*, May 2016. `http://isa-afp.org/entries/Incredible_Proof_Machine.shtml`, Formal proof development.

[18] A. Bundy. The use of explicit plans to guide inductive proofs. In E. Lusk and R. Overbeek, editors, *CADE-9*, volume 310 of *LNCS*, pages 111–120. Springer, 1988.

[19] C.-L. Chang and R. C.-T. Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, Inc., 1st edition, 1973.

[20] M. Coen, K. Slind, and A. Krauss. Theory unification. Isabelle. `http://isabelle.in.tum.de/library/HOL/HOL-ex/Unification.html`. Accessed 13 December 2017.

[21] M. D. Coen. *Interactive Program Derivation*. PhD thesis, University of Cambridge, 1992. `http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-272.html`.

[22] J. Corbin and M. Bidoit. A rehabilitation of Robinson's unification algorithm. In *IFIP Congress*, pages 909–914, 1983.

[23] J. Davis and M. O. Myreen. The reflective Milawa theorem prover is sound (down to the machine code that runs it). *Journal of Automated Reasoning*, 2015.

[24] H. Ebbinghaus, J. Flum, and W. Thomas. *Mathematical Logic*. Springer, 2nd edition, 1994.

[25] M. Fitting. *First-Order Logic and Automated Theorem Proving*. Springer, 2nd edition, 1996.

[26] H. Gebhard. Beweisplanung für die Beweise der Vollständigkeit verschiedener Resolutionskalküle in ΩMEGA. Master's thesis, Saarland University, 1999.

[27] J. Goubault-Larrecq and J.-P. Jouannaud. The blossom of finite semantic trees. In A. Voronkov and C. Weidenbach, editors, *Programming Logics: Essays in Memory of Harald Ganzinger*, LNCS, pages 90–122. Springer, 2013.

[28] J. Harrison. Formalizing basic first order model theory. In J. Grundy and M. Newey, editors, *TPHOL's 1998*, volume 1497 of *LNCS*, pages 153–170. Springer, 1998.

[29] J. Harrison. Towards self-verification of HOL Light. In U. Furbach and N. Shankar, editors, *IJCAR 2006*, volume 4130 of *LNCS*, pages 177–191. Springer, 2006.

[30] J. Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, 2009.

[31] D. Ilik. *Constructive Completeness Proofs and Delimited Control*. PhD thesis, École Polytechnique, 2010. `https://tel.archives-ouvertes.fr/tel-00529021/document`.

[32] D. Ilik, G. Lee, and H. Herbelin. Kripke models for classical logic. *Annals of Pure and Applied Logic*, 161(11):1367–1378, 2010.

[33] IsaFoL authors. IsaFoL: Isabelle Formalization of Logic. `https://bitbucket.org/isafol/isafol`. Accessed 13 December 2017.

[34] IsaFoR developers. An Isabelle/HOL formalization of rewriting for certified termination analysis. `http://cl-informatik.uibk.ac.at/software/ceta/`. Accessed 13 December 2017.

[35] A. B. Jensen, A. Schlichtkrull, and J. Villadsen. Verification of an LCF-style first-order prover with equality. In *Isabelle Workshop 2016 Associated with ITP 2016*, 2016.

[36] A. B. Jensen, A. Schlichtkrull, and J. Villadsen. First-order logic according to Harrison. *Archive of Formal Proofs*, Jan. 2017. `http://isa-afp.org/entries/FOL_Harrison.shtml`, Formal proof development.

[37] M. Kohlhase. Theorem prover museum – OMEGA theories – folders: propositional-logic, resolution, proof-theory, prop-res. `https://github.com/theoremprover-museum/OMEGA/tree/master/theories`. Accessed 13 December 2017.

[38] A. Krauss. Partial and nested recursive function definitions in higher-order logic. *Journal of Automated Reasoning*, 44(4):303–336, 2010.

[39] R. Kumar, R. Arthan, M. O. Myreen, and S. Owens. Self-formalisation of higher-order logic: Semantics, soundness, and a verified implementation. *Journal of Automated Reasoning*, 56(3):221–259, 2016.

[40] P. Lammich. Efficient verified (UN)SAT certificate checking. In L. de Moura, editor, *CADE-26*, volume 10395 of *LNCS*, pages 237–254. Springer, 2017.

[41] P. Lammich. The GRAT tool chain. In S. Gaspers and T. Walsh, editors, *SAT 2017*, volume 10491 of *LNCS*, pages 457–463. Springer, 2017.

[42] A. Leitsch. On different concepts of resolution. *Mathematical Logic Quarterly*, 35(1):71–77, 1989.

[43] A. Leitsch. *The Resolution Calculus*. Springer, 1997.

[44] Z. Manna and R. Waldinger. Deductive synthesis of the unification algorithm. *Science of Computer Programming*, 1(1):5–48, 1981.

[45] J. Margetson and T. Ridge. Completeness theorem. *Archive of Formal Proofs*, Sept. 2004. `http://isa-afp.org/entries/Completeness.shtml`, Formal proof development.

[46] T. Nipkow and G. Klein. *Concrete Semantics: With Isabelle/HOL*. Springer, 2014.

[47] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Springer, 2002.

[48] L. C. Paulson. Verifying the unification algorithm in LCF. *Science of Computer Programming*, 5(2):143–169, June 1985.

[49] L. C. Paulson. Gödel's incompleteness theorems. *Archive of Formal Proofs*, Nov. 2013. `http://isa-afp.org/entries/Incompleteness.shtml`, Formal proof development.

[50] L. C. Paulson. A machine-assisted proof of Gödel's incompleteness theorems for the theory of hereditarily finite sets. *Review of Symbolic Logic*, 7(03):484–498, 2014.

[51] L. C. Paulson. A mechanised proof of Gödel's incompleteness theorems using Nominal Isabelle. *Journal of Automated Reasoning*, 55(1):1–37, 2015.

[52] N. Peltier. Propositional resolution and prime implicates generation. *Archive of Formal Proofs*, Mar. 2016. `http://isa-afp.org/entries/PropResPI.shtml`, Formal proof development.

[53] N. Peltier. A variant of the superposition calculus. *Archive of Formal Proofs*, Sept. 2016. `http://isa-afp.org/entries/SuperCalc.shtml`, Formal proof development.

[54] H. Persson. *Constructive completeness of intuitionistic predicate logic*. PhD thesis, Chalmers University of Technology, 1996. `http://web.archive.org/web/19970715002824/http://www.cs.chalmers.se/~henrikp/Lic/`.

[55] C. Raffalli. Krivine's abstract completeness proof for classical predicate logic. `https://github.com/craff/phox/blob/master/examples/complete.phx`, 2005, possibly earlier. Accessed 13 December 2017.

[56] A. Riazanov and A. Voronkov. Vampire. In H. Ganzinger, editor, *CADE-16*, volume 1632 of *LNCS*, pages 292–296. Springer, 1999.

[57] T. Ridge. A mechanically verified, efficient, sound and complete theorem prover for first order logic. *Archive of Formal Proofs*, Sept. 2004. `http://isa-afp.org/entries/Verified-Prover.shtml`, Formal proof development.

[58] T. Ridge and J. Margetson. A mechanically verified, sound and complete theorem prover for first order logic. In J. Hurd and T. Melham, editors, *TPHOL's 2005*, volume 3603 of *LNCS*, pages 294–309. Springer, 2005.

[59] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[60] J. A. Robinson. The generalized resolution principle. *Machine Intelligence*, 3:77–93, 1968.

[61] J.-L. Ruiz-Reina, F.-J. Martín-Mateos, J.-A. Alonso, and M.-J. Hidalgo. Formal correctness of a quadratic unification algorithm. *Journal of Automated Reasoning*, 37(1):67–92, 2006.

[62] A. Schlichtkrull. Formalization of resolution calculus in Isabelle. Master's thesis, Technical University of Denmark, 2015. `https://people.compute.dtu.dk/andschl/Thesis.pdf`.

[63] A. Schlichtkrull. Formalization of the resolution calculus for first-order logic. In J. Blanchette and S. Merz, editors, *ITP 2016*, volume 9807 of *LNCS*, pages 341–357. Springer, 2016.

[64] A. Schlichtkrull. The resolution calculus for first-order logic. *Archive of Formal Proofs*, June 2016. `http://isa-afp.org/entries/Resolution_FOL.shtml`, Formal proof development.

[65] A. Schlichtkrull, J. C. Blanchette, D. Traytel, and U. Waldmann. Formalization of Bachmair and Ganzinger's simple ordered resolution prover. `https://bitbucket.org/isafol/isafol/src/master/Ordered_Resolution_Prover/`. Accessed 13 December 2017.

[66] A. Schlichtkrull and J. Villadsen. Paraconsistency. *Archive of Formal Proofs*, Dec. 2016. `http://isa-afp.org/entries/Paraconsistency.shtml`, Formal proof development.

[67] J. J. Schlöder and P. Koepke. The Gödel completeness theorem for uncountable languages. *Formalized Mathematics*, 20(3):199–203, 2012.

[68] S. Schulz. System description: E 1.8. In K. McMillan, A. Middeldorp, and A. Voronkov, editors, *LPAR-19*, volume 8312 of *LNCS*, pages 735–743. Springer, 2013.

[69] R. Sekar, I. V. Ramakrishnan, and A. Voronkov. Term indexing. In *Handbook of Automated Reasoning*, volume 2, pages 1853–1964. 2001.

[70] N. Shankar. *Proof-checking Metamathematics*. PhD thesis, University of Texas, 1986.

[71] N. Shankar. *Metamathematics, Machines, and Gödel's Proof*. Cambridge University Press, 1994.

[72] K. Slind. *Reasoning about Terminating Functional Programs*. PhD thesis, Technical University of Munich, 1999. `https://mediatum.ub.tum.de/?id=601660`.

[73] C. Sternagel and R. Thiemann. Formalizing Knuth-Bendix orders and Knuth-Bendix completion. In F. van Raamsdonk, editor, *RTA '13*, volume 21 of *LIPIcs*, pages 287–302. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2013.

[74] C. Weidenbach, D. Dimova, A. Fietzke, R. Kumar, M. Suda, and P. Wischnewski. SPASS version 3.5. In R. A. Schmidt, editor, *CADE-22*, volume 5663 of *LNCS*, pages 140–145. Springer, 2009.

[75] M. Wenzel. Isar — a generic interpretative approach to readable formal proof documents. In Y. Bertot, G. Dowek, L. Théry, A. Hirschowitz, and C. Paulin, editors, *TPHOL's 1999*, volume 1690 of *LNCS*, pages 167–183. Springer, 1999.

Formalizing Bachmair and Ganzinger's Ordered Resolution Prover

Anders Schlichtkrull[1], Jasmin Christian Blanchette[2,3], Dmitriy Traytel[4], and Uwe Waldmann[3]

[1] *DTU Compute, Technical University of Denmark, Kongens Lyngby, Denmark*
[2] *Department of Computer Science, Vrije Universiteit Amsterdam, Amsterdam, The Netherlands*
[3] *Max-Planck-Institut für Informatik, Saarland Informatics Campus, Saarbrücken, Germany*
[4] *Institute of Information Security, Department of Computer Science, ETH Zürich, Zürich, Switzerland*

Abstract

We present a formalization of the first half of Bachmair and Ganzinger's chapter on resolution theorem proving in Isabelle/HOL, culminating with a refutationally complete first-order prover based on ordered resolution with literal selection. We develop general infrastructure and methodology that can form the basis of completeness proofs for related calculi, including superposition. Our work clarifies several of the fine points in the chapter's text, emphasizing the value of formal proofs in the field of automated reasoning.

1 Introduction

Much research in automated reasoning amounts to metatheoretical arguments, typically about the soundness and completeness of logical inference systems or the termination of theorem proving processes. Often the proofs contain more insights than the systems or processes themselves. For example, the superposition calculus rules [2], with their many side conditions, look rather arbitrary, whereas in the completeness proof the side conditions emerge naturally from the model construction. And yet, despite being crucial to our field, today such proofs are usually carried out without tool support beyond $\TeX$.

We believe proof assistants are becoming mature enough to help. In this report, we present a formalization, developed using the Isabelle/HOL system [21], of a first-order prover based on ordered resolution with literal selection. We follow Bachmair and Ganzinger's account [3] from Chapter 2 of the *Handbook of Automated Reasoning*, which we will simply refer to as "the chapter." Our formal development covers the refutational completeness of two resolution calculi for ground (i.e., variable-free) clauses and general infrastructure for theorem proving processes and redundancy, culminating with a completeness proof for a first-order prover expressed as transition rules operating on triples of clause sets. This material corresponds to the chapter's first four sections.

From the perspective of automated reasoning, increased trustworthiness of the results is an obvious benefit of formal proofs. But formalizing also helps clarify arguments, by exposing and explaining difficult steps. Making theorem statements (including definitions and hypotheses) precise can be a huge gain for communicating results. Moreover, a formal proof can tell us exactly where hypotheses and lemmas are used. Once we have created a library of basic results and a methodology, we will be in a good position to study extensions and variants. Given that automatic theorem provers are integrated in modern proof assistants, there is also an undeniable thrill in applying these tools to reason about their own metatheory.

From the perspective of interactive theorem proving, formalization work constitutes a case study in the use of a proof assistant. It gives us, as developers and users of such a system, an opportunity to experiment, contribute to lemma libraries, and get inspiration for new features and improvements.

Our motivation for choosing Bachmair and Ganzinger's chapter is manyfold. The text is a standard introduction to superposition-like calculi (together with *Handbook* Chapters 7 [18] and 27 [35]). It offers perhaps the most detailed treatment of the lifting of a resolution-style calculus's static completeness to a saturation prover's dynamic completeness. It introduces a considerable amount of general infrastructure, including different types of inference systems (sound, reductive, counterexample-reducing, etc.), theorem proving processes, and an abstract notion of redundancy. The resolution calculus, extended with a term order and literal selection, captures most of the insights underlying ordered paramodulation and superposition, but with a simple notion of model.

The chapter's level of rigor is uneven, as shown by the errors and imprecisions revealed by our formalization. These are only to be expected in technical material of this kind. Far from diminishing the original work, our corrections should increase its value to the research community. We will see that the main completeness result does not hold, due to the improper treatment of self-inferences. Naturally, our objective is not to diminish Bachmair and Ganzinger's outstanding achievements, which include the development of superposition; rather, it is to demonstrate that even the work of some of the most celebrated researchers in our field can benefit from formalization. Our view is that formal proofs can be used to complement and improve their informal counterparts.

This work is part of the IsaFoL (Isabelle Formalization of Logic) project,[1] which aims at developing a library of results about logical calculi used in automated reasoning. The Isabelle theory files are available in the *Archive of Formal Proofs* (AFP).[2] They amount to about 8000 lines of source text. Below we provide implicit hyperlinks from theory names. A better way to study the theory files, however, is to open them in Isabelle/jEdit [37], an integrated development environment for formal proof. This will ensure that logical and mathematical symbols are rendered properly (e.g., ∀ instead of \\<*forall*>) and let you inspect proof states. We used Isabelle version 2017, but the AFP is continuously updated to track Isabelle's evolution. We assume the reader has some familiarity with the chapter's content.

2 Preliminaries

Ordered resolution depends on little background metatheory that needs to be formalized using Isabelle. Much of it, concerning partial and total orders, well-foundedness, and finite multisets, is provided by standard Isabelle libraries. We also need literals, clauses, models, terms, and substitutions.

Isabelle. Isabelle/HOL [21] is a proof assistant based on classical higher-order logic (HOL) with Hilbert choice, the axiom of infinity, and rank-1 polymorphism. It is the logic of Gordon's original HOL system [14] and of its many successors. HOL notations are similar to those of functional programming languages. Functions are applied without parentheses or commas (e.g., $\mathsf{f}\ x\ y$). Through syntactic abbreviations, many traditional notations from mathematics are provided, notably to denote simply typed sets and multisets. We refer to Nipkow and Klein [20, Part 1] for a modern introduction.

[1]`https://bitbucket.org/isafol/isafol/wiki/Home`

[2]`https://devel.isa-afp.org/entries/Ordered_Resolution_Prover.html`

Clauses and Models. We use the same library of clauses (`Clausal_Logic.thy`) as for the verified SAT solver by Blanchette et al. [6], which is also part of IsaFoL. Atoms are represented by a type variable $'a$, which can be instantiated by arbitrary concrete types—e.g., numbers or first-order terms. A literal, of type $'a$ *literal* (where the type constructor is written in ML-style postfix syntax), can be of the form $\mathsf{Pos}\ A$ or $\mathsf{Neg}\ A$, where $A :: {}'a$ is an atom. The literal order $>$ (written $\succ$ in the chapter) extends a fixed atom order $>$ by comparing polarities to break ties, with $\mathsf{Neg}\ A > \mathsf{Pos}\ A$. Following the chapter, a clause is defined as a finite multiset of literals, $'a$ *clause* $=$ $'a$ *literal multiset*, where *multiset* is the Isabelle type constructor of finite multisets. Thus, the clause $A \vee B$, where A and B are atoms, is identified with the multiset $\{A, B\}$; the clause $C \vee D$, where C and D are clauses, is $C \uplus D$; and the empty clause $\bot$ is $\{\}$. The clause order is the multiset extension of the literal order.

A Herbrand interpretation I is a value of type $'a$ *set*, specifying which ground atoms are true (`Herbrand_Interpretation.thy`). The "models" operator $\vDash$ is defined in the usual way on atoms, literals, clauses, sets, and multisets of clauses; for example, $I \vDash C \Leftrightarrow \exists L \in C.\ I \vDash L$. Satisfiability of a set or multiset of clauses N is defined by $\mathsf{sat}\ N \Leftrightarrow \exists I.\ I \vDash N$.

Multisets are central to our development. Isabelle provides a multiset library, but it is much less developed than those of sets and lists. As part of IsaFoL, we have already extended it considerably and implemented further additions in a separate file (`Multiset_More.thy`). Some of these, notably a plugin for Isabelle's simplifier to apply cancellation laws, are described in a recent paper [7, Sect. 3].

The main hurdle we faced concerned the multiset order. Multisets of clauses have type $'a$ *literal multiset multiset*. The corresponding order is the multiset extension of the clause order. In Isabelle, the multiset order was called $\#{\subset}\#$, and it relied on the element type's $<$ operator, through Isabelle's type class mechanism. Unfortunately, for multisets, $<$ was defined as the subset relation, so when nesting multisets (as $'a$ *multiset multiset*), we obtained the multiset extension of the subset relation. Initially, we worked around the issue by defining an order $\#{\subset}\#\#$ on multisets of multisets, but we also saw potential for improvement. After some discussions on the Isabelle users' mailing list, we decided to let $<$ be the multiset order and introduce the symbol ${\subset}\#$ for the subset relation. To avoid introducing subtle changes in the semantics of existing developments, we first renamed $<$ to ${\subset}\#$, freeing up $<$; then, in the next Isabelle release, we renamed $\#{\subset}\#$ to $<$. In the intermediate state, all occurrences of $<$ and of the lemmas about it were flagged as errors, easing porting. Similar changes affected the nonstrict versions of the orders (e.g., $\leq$) and all the lemmas about them (e.g., *add_mono*: $M \leq M' \Longrightarrow N \leq N' \Longrightarrow M \uplus N \leq M' \uplus N'$).

Terms and Substitutions. The IsaFoR (Isabelle Formalization of Rewriting) library—an inspiration for IsaFoL—contains a definition of first-order terms and results about substitutions and unification [32]. It makes sense to reuse this functionality. A practical issue is that most of IsaFoR is not accessible from the AFP.

Resolution depends only on basic properties of terms and atoms, such as the existence of most general unifiers (MGUs). We exploit this to keep the development parameterized by a type of atoms $'a$ and an abstract type of substitutions $'s$, through Isabelle locales [4] (`Abstract_Substitution.thy`). A locale represents a module parameterized by types and terms that satisfy some assumptions. Inside the locale, we can refer to the parameters and assumptions in definitions, lemmas, and proofs. The basic operations provided by our locale are application ($\cdot :: {}'a \Rightarrow {}'s \Rightarrow {}'a$), identity ($\mathsf{id} :: {}'s$), and composition ($\circ :: {}'s \Rightarrow {}'s \Rightarrow {}'s$), about which some assumptions are made (e.g., $A \cdot \mathsf{id} = A$ for all atoms A). Substitution is lifted to literals, clauses, sets of clauses, and so on. Many other operations can be defined in terms of the primitives—for example:

$$\mathsf{is\_ground}\ A \Leftrightarrow \forall\sigma.\ A = A \cdot \sigma \qquad \mathsf{is\_renaming}\ \sigma \Leftrightarrow \exists\tau.\ \sigma \circ \tau = \mathsf{id}$$
$$\mathsf{is\_ground}\ \sigma \Leftrightarrow \forall A.\ \mathsf{is\_ground}\ (A \cdot \sigma) \qquad \mathsf{instance\_of}\ C\ D \Leftrightarrow \exists\sigma.\ C \cdot \sigma = D$$

MGUs are also taken as a primitive: the $\mathsf{mgu} :: {}'a\ set\ set \Rightarrow {}'s\ option$ operation takes a set of unification constraints, each of the form $A_1 \overset{?}{=} \cdots \overset{?}{=} A_n$, and returns either an MGU or a special value (None).

Perhaps the main reason why multisets are preferable to sets for representing clauses is that they are better behaved with respect to substitution. Using a set representation of clauses, applying $\sigma = \{x \mapsto \mathsf{a},\ y \mapsto \mathsf{a}\}$ to either the unit clause $C = \mathsf{p}(x)$ or the two-literal clause $D = \mathsf{p}(x) \vee \mathsf{p}(y)$ yields a unit clause $\mathsf{p}(\mathsf{a})$. This oddity breaks a property called "stability under substitution"—the requirement that $D > C$ imply $D \cdot \sigma > C \cdot \sigma$.

To complete our formal development and ensure that our assumptions are legitimate, we instantiate the locale's parameters with IsaFoR types and operations and discharge its assumptions (`IsaFoR_Term.thy`). This bridge is currently hosted on the IsaFoL repository, outside the AFP.

3 Refutational Inference Systems

In their Sect. 2.4, Bachmair and Ganzinger introduce basic conventions for refutational inference systems. In Sect. 3, they present two ground resolution calculi and prove them refutationally complete in Theorems 3.9 and 3.16. In Sect. 4.2, they introduce a notion of counterexample-reducing inference system and state Theorem 4.4 as a generalization of Theorems 3.9 and 3.16 to all such systems. For formalization, two courses of actions suggest themselves: follow the book closely and prove the three theorems separately, or focus on the most general result. We choose the latter, as being more consistent with the goal of providing a well-designed, reusable library, at the cost of widening the gap between the text and its formal companion.

We collect the abstract hierarchy of inference systems in a single Isabelle theory file (`Inference_System.thy`). An inference, of type *'a inference*, is a triple $(\mathcal{C}, D, E)$ that consists of a multiset of side premises $\mathcal{C}$, a main premise D, and a conclusion E. An inference system, or calculus, is a possibly infinite set of inferences:

locale *inference_system* =
  fixes $\Gamma :: {}'a\ inference\ set$

We use an Isabelle locale to fix, within a named context (*inference_system*), a set Γ of inferences between clauses over atom type *'a*. Inside the locale, we define a function infers_from that, given a clause set N, returns the subset of Γ inferences whose premises all belong to N.

A satisfiability-preserving (or consistency-preserving) inference system enriches the inference system locale with an assumption, whereas sound systems are characterized by a different assumption:

locale *sat_preserving_inference_system* = *inference_system* +
  assumes $\mathsf{sat}\ N \Longrightarrow \mathsf{sat}\ (N \cup \mathsf{concl\_of}\ `\ \mathsf{infers\_from}\ N)$

locale *sound_inference_system* = *inference_system* +
  assumes $(\mathcal{C}, D, E) \in \Gamma \Longrightarrow I \vDash \mathcal{C} \cup \{D\} \Longrightarrow I \vDash E$

The notation $f\ `\ X$ above stands for the image of the set or multiset X under function f.

Soundness is a stronger requirement than satisfiability preservation. In Isabelle, this can be expressed as a sublocale relation:

sublocale *sound_inference_system* $<$ *sat_preserving_inference_system*

This command emits a proof goal stating that *sound_inference_system*'s assumption implies *sat_preserving_inference_system*'s. Afterwards, all the definitions and lemmas about satisfiability-preserving calculi become available about sound ones.

In reductive inference systems (*reductive_inference_system*), the conclusion of each inference is smaller than the main premise according to the clause order. A related notion, the counterexample-reducing inference systems, is specified as follows:

```
locale counterex_reducing_inference_system = inference_system +
  fixes I_of :: 'a clause set ⇒ 'a set
  assumes {} ∉ N ⟹ D ∈ N ⟹ I_of N ⊭ D ⟹
    (∀C ∈ N. I_of N ⊭ C ⟹ D ≤ C) ⟹
    ∃𝒞 ⊆ N. ∃E. I_of N ⊨ 𝒞 ∧ (𝒞, D, E) ∈ Γ ∧ I_of N ⊭ E ∧ E < D
```

The "model functor" parameter I_of maps clause sets to candidate models. The assumption is that for any clause set N that does not contain $\{\}$ (i.e., $\bot$), if $D \in N$ is the smallest counterexample—the smallest clause in N that is falsified by I_of N—we can derive a smaller counterexample E using an inference from clauses in N. This property is useful because if N is saturated (i.e., closed under Γ inferences), we must have $E \in N$, contradicting D's minimality:

theorem *saturated_model*: saturated $N \Longrightarrow \{\} \notin N \Longrightarrow$ I_of $N \models N$
corollary *saturated_complete*: saturated $N \Longrightarrow \neg$ sat $N \Longrightarrow \{\} \in N$

Bachmair and Ganzinger claim that compactness of clausal logic follows from the refutational completeness of ground resolution (Theorem 3.12), although they give no justification. Our argument relies on an inductive definition of saturation of a set of clauses: saturate :: *'a clause set* $\Rightarrow$ *'a clause set*. Most of the work goes into proving this key lemma, by rule induction on the saturate function:

lemma *saturate_finite*: $C \in$ saturate $N \Longrightarrow \exists M \subseteq N.$ finite $M \wedge C \in$ saturate M

The interesting case is when $C = \bot$. We establish compactness in a locale that combines *counterex_reducing_inference_system* and *sound_inference_system*:

theorem *clausal_logic_compact*: $\neg$ sat $N \Leftrightarrow \exists M \subseteq N.$ finite $M \wedge \neg$ sat M

To give a taste of the formalization, here is the formal proof, expressed using Isabelle's structured Isar format [36]:

```
proof
  assume ¬ sat N
  then have {} ∈ saturate N
    using saturated_complete saturated_saturate saturate.base
    unfolding true_clss_def by meson
  then have ∃M ⊆ N. finite M ∧ {} ∈ saturate M
    using saturate_imp_finite_subset by fastforce
  then show ∃M ⊆ N. finite M ∧ ¬ sat M
    using saturate_sound by auto
next
  assume ∃M ⊆ N. finite M ∧ ¬ sat M
  then show ¬ sat N
    by (blast intro: true_clss_mono)
qed
```

In the "implies" direction, we rely on the calculus's refutation completeness to show that $\bot$ belongs to saturate N, on the above key lemma to obtain a finite subset M from which $\bot$ can be derived, and on the calculus's soundness to conclude that M is unsatisfiable. We believe satisfiability preservation could be used instead of soundness, relying on the property that sat $N \Longrightarrow$ sat (saturate N) for satisfiability-preserving calculi, but we have yet to find a good way to prove this formally.

Our compactness result is meaningful only if the locale assumptions are consistent. In the next section, we will exhibit two sound counterexample-reducing calculi that can be used to instantiate the locale and retrieve an unconditional compactness theorem.

4 Ground Resolution

A useful strategy for establishing properties of first-order calculi is to initially restrict our attention to ground calculi and then to lift the results to first-order formulas containing terms with variables. Accordingly, the chapter's Sect. 3 presents two ground calculi: a simple binary resolution calculus and an ordered resolution calculus with literal selection. Both consist of a single resolution rule, with built-in positive factorization. Most of the explanations and proofs concern the simpler calculus. To avoid duplication, we factor out the candidate model construction (`Ground_Resolution_Model.thy`). We then define the two calculi and prove that they are sound and reduce counterexamples (`Unordered_Ground_Resolution.thy`, `Ordered_Ground_Resolution.thy`).

Candidate Models. Refutational completeness is proved by exhibiting a model for any saturated clause set N that does not contain $\bot$. The model is constructed incrementally, one clause $C \in N$ at a time, starting with an empty Herbrand interpretation. The idea appears to have originated with Brand [10] and Zhang and Kapur [38].

Bachmair and Ganzinger introduce two operators to build the candidate model: I_C denotes the current interpretation before considering C, and ε_C denotes the set of (zero or one) atoms added, or *produced*, to ensure that C is satisfied. The candidate model construction is parameterized by a literal selection function S. It can be ignored by taking $S := \lambda C.\ \{\}$.

```
locale $ground\_resolution\_with\_selection$ =
  fixes $S :: 'a\ clause \Rightarrow 'a\ clause$
  assumes $S\ C \subseteq C$ and $L \in S\ C \Longrightarrow \mathsf{is\_neg}\ L$
```

Inside the locale, we fix a clause set N, for which we try to derive a model. Then we define two operators corresponding to ε_C and I_C:

```
function production :: $'a\ clause \Rightarrow 'a\ set$ where
  production $C = \{A \mid C \in N \wedge C \neq \{\} \wedge \mathsf{Max}\ C = \mathsf{Pos}\ A$
                   $\wedge \big(\bigcup_{D<C} \mathsf{production}\ D\big) \not\models C \wedge S\ C = \{\}\}$
definition interp :: $'a\ clause \Rightarrow 'a\ set$ where
  interp $C = \bigcup_{D<C} \mathsf{production}\ D$
```

To ensure monotonicity of the construction, any produced atom must be maximal in its clause. Moreover, productive clauses may not contain selected literals. In the chapter, ε_C and I_C are expressed in terms of each other. We simplified the definition by inlining I_C in ε_C, so that only ε_C is recursive. Since the recursive calls operate on clauses D that are smaller with respect to a well-founded order, the definition is accepted [16]. Once the operators are defined, we can fold

interp's definition in $\mathsf{production}$'s equation to derive the intended mutually recursive specification as a lemma. Bachmair and Ganzinger's I^C and I_N operators are introduced as abbreviations:

$$\mathsf{Interp}\ C = \mathsf{interp}\ C \cup \mathsf{production}\ C \qquad\qquad \mathsf{INTERP} = \textstyle\bigcup_{C \in N} \mathsf{production}\ C$$

We then prove a host of lemmas about these concepts. Lemma 3.4 amounts to six monotonicity properties:

> **lemma** *Interp_imp_interp*: $C \le D \Longrightarrow D < D' \Longrightarrow \mathsf{Interp}\ D \vDash C \Longrightarrow \mathsf{interp}\ D' \vDash C$
> **lemma** *Interp_imp_Interp*: $C \le D \Longrightarrow D \le D' \Longrightarrow \mathsf{Interp}\ D \vDash C \Longrightarrow \mathsf{Interp}\ D' \vDash C$
> **lemma** *Interp_imp_INTERP*: $C \le D \Longrightarrow \mathsf{Interp}\ D \vDash C \Longrightarrow \mathsf{INTERP} \vDash C$
> **lemma** *interp_imp_interp*: $C \le D \Longrightarrow D \le D' \Longrightarrow \mathsf{interp}\ D \vDash C \Longrightarrow \mathsf{interp}\ D' \vDash C$
> **lemma** *interp_imp_Interp*: $C \le D \Longrightarrow D \le D' \Longrightarrow \mathsf{interp}\ D \vDash C \Longrightarrow \mathsf{Interp}\ D' \vDash C$
> **lemma** *interp_imp_INTERP*: $C \le D \Longrightarrow \mathsf{interp}\ D \vDash C \Longrightarrow \mathsf{INTERP}\ \vDash C$

In the chapter, the first property is wrongly stated with $D \le D'$ instead of $D < D'$, admitting the counterexample $N = \{\{A\}\}$ and $C = D = D' = \{A\}$. Lemma 3.3, whose proof depends on monotonicity, is better proved *after* 3.4:

> **lemma** *productive_imp_INTERP*: $\mathsf{production}\ C \neq \{\} \Longrightarrow \mathsf{INTERP} \vDash C$

A more serious oddity is Lemma 3.7. Using our notations, it can be stated as

$$D \in N \Longrightarrow C \neq D \Longrightarrow \big(\forall D' \le D.\ \mathsf{Interp}\ D' \vDash C\big) \Longrightarrow \mathsf{interp}\ D \vDash D'$$

However, the last occurrence of D' is clearly wrong—the context suggests C instead. Even after this amendment, we have a counterexample, corresponding to a gap in the proof: $D = \{\}$, $C = \{\mathsf{Pos}\ A\}$, and $N = \{D, C\}$. Since this "lemma" is not actually used, we can simply ignore it.

Unordered Resolution. The unordered ground resolution calculus consists of a single binary inference rule, with the side premise $C \vee A \vee \cdots \vee A$, the main premise $\neg A \vee D$, and the conclusion $C \vee D$:

$$\frac{C \vee A \vee \cdots \vee A \quad \neg A \vee D}{C \vee D}$$

Formally, this rule is captured by a predicate:

> **inductive** $\mathsf{unord\_resolve} :: {'a\ clause} \Rightarrow {'a\ clause} \Rightarrow {'a\ clause} \Rightarrow bool$ **where**
> $\mathsf{unord\_resolve}\ (C \uplus \mathsf{replicate}\ (n+1)\ (\mathsf{Pos}\ A))\ (\{\mathsf{Neg}\ A\} \uplus D)\ (C \uplus D)$

Soundness is trivial to prove:

> **lemma** *unord_resolve_sound*: $\mathsf{unord\_resolve}\ C\ D\ E \Longrightarrow I \vDash C \Longrightarrow I \vDash D \Longrightarrow I \vDash E$
> **using** *unord_resolve.cases* **by** *fastforce*

To prove completeness, it suffices to show that the calculus reduces counterexamples. This corresponds to Theorem 3.8, except that the conclusion is strengthened slightly to match *counterex_reducing_inference_system*'s assumption:

> **theorem** *unord_resolve_counterex_reducing*:
> **assumes** $\{\} \notin N$ **and** $C \in N$ **and** $\mathsf{INTERP}\ N \nvDash C$ **and**
> $\forall D \in N.\ \mathsf{INTERP}\ N \nvDash D \Longrightarrow C \le D$
> **obtains** $D\ E$ **where**
> $D \in N$ **and** $\mathsf{INTERP}\ N \vDash D$ **and** $\mathsf{production}\ N\ D \neq \{\}$ **and**
> $\mathsf{unord\_resolve}\ D\ C\ E$ **and** $\mathsf{INTERP}\ N \nvDash E$ **and** $E < C$

The arguments N to INTERP and production are necessary because we are outside the block in which N was fixed. This explicit dependency allows us to instantiate the locale's I_of :: $'a$ *clause set* $\Rightarrow$ $'a$ *set* parameter with INTERP.

By instantiating the *sound_inference_system* and *counterex_reducing_inference_system* locales, we obtain refutational completeness (Theorem 3.9 and Corollary 3.10) and compactness of clausal logic (Theorem 3.12).

Ordered Resolution with Selection. Ordered ground resolution consists of a single rule, ord_resolve. Like unord_resolve, it is sound and counterexample-reducing (Theorem 3.15). Moreover, it is reductive (Lemma 3.13): the conclusion is always smaller than the main premise according to the clause order. The rule is given as

$$\frac{C_1 \vee A_1 \vee \cdots \vee A_1 \quad \cdots \quad C_n \vee A_n \vee \cdots \vee A_n \quad \neg A_1 \vee \cdots \vee \neg A_n \vee D}{C_1 \vee \cdots \vee C_n \vee D}$$

with multiple side conditions whose role is to prune the search space and to make the rule reductive.

The n-ary nature of the rule constitutes a substantial complication. The ellipsis notation hides most of the complexity in the informal proof, but in Isabelle, even stating the rule is tricky, let alone reasoning about it. We represent the n side premises by three parallel lists of length n: CAs gives the entire clauses, whereas Cs and $\mathcal{A}s$ store the C_i and the $\mathcal{A}_i = A_i \vee \cdots \vee A_i$ parts separately. In addition, As is the list $[A_1, \ldots, A_n]$. The following inductive definition captures the rule formally:

inductive ord_resolve :: $'a$ *clause list* $\Rightarrow$ $'a$ *clause* $\Rightarrow$ $'a$ *clause* $\Rightarrow$ *bool* **where**
$|CAs| = n \Longrightarrow |Cs| = n \Longrightarrow |\mathcal{A}s| = n \Longrightarrow |As| = n \Longrightarrow n \neq 0 \Longrightarrow$
$(\forall i < n.\ CAs\,!\,i = Cs\,!\,i \uplus \mathsf{Pos}\text{ ` }\mathcal{A}s\,!\,i) \Longrightarrow (\forall i < n.\ \mathcal{A}s\,!\,i \neq \{\}) \Longrightarrow$
$(\forall i < n.\ \forall A \in \mathcal{A}s\,!\,i.\ A = As\,!\,i) \Longrightarrow \mathsf{eligible}\ As\ (D \uplus \mathsf{Neg}\text{ ` }\mathsf{mset}\ As) \Longrightarrow$
$(\forall i < n.\ \mathsf{strict\_max\_in}\ (As\,!\,i)\ (Cs\,!\,i)) \Longrightarrow (\forall i < n.\ S\ (CAs\,!\,i) = \{\}) \Longrightarrow$
$\mathsf{ord\_resolve}\ CAs\ (D \uplus \mathsf{Neg}\text{ ` }\mathsf{mset}\ As)\ ((\bigcup \mathsf{mset}\ Cs) \uplus D)$

The $xs\,!\,i$ operator returns the $(i + 1)$st element of xs, and mset converts a list to a multiset. Before settling on the above formulation, we tried storing the n premises in a multiset, since their order is irrelevant. However, due to the permutative nature of multisets, there can be no such things as "parallel multisets"; to keep the dependencies between the C_i's and the A_i's, we must keep them in a single multiset of tuples, which is very unwieldy.

A previous version of the formalization represented each $A_i \vee \cdots \vee A_i$ as a value of type $'a \times nat$—the nat representing the number of times A_i is repeated. With this approach, the definition of ord_resolve did not need to state the equality of the atoms in each $As\,!\,i$. Other than that, there was nothing to win, and the approach does not work on the first-order level where atoms should be unifiable instead of equal. To achieve symmetry between the ground and first-order calculi, we went with the current approach.

Formalization revealed an error and a few ambiguities in the rule's statement. References to $S(D)$ in the side conditions should have been to $S(\neg A_1 \vee \cdots \vee \neg A_n \vee D)$. The ambiguities are discussed in Appendix A.

Soundness is a good sanity check for our definition:

lemma *ord_resolve_sound*:
ord_resolve $CAs\ DA\ E \Longrightarrow I \models \mathsf{mset}\ CAs \Longrightarrow I \models DA \Longrightarrow I \models E$

The proof is by case distinction: either the interpretation I contains all atoms A_i, in which case the D subclause of the main premise $\neg A_1 \vee \cdots \vee \neg A_n \vee D$ must be true, or there exists an index i such that $A_i \notin I$, in which case the corresponding C_i must be true. In both cases, the conclusion $C_1 \vee \cdots \vee C_n \vee D$ is true.

5 Theorem Proving Processes

In their Sect. 4, Bachmair and Ganzinger switch from a static to a dynamic view of saturation: from clause sets closed under inferences to theorem proving processes that start with a clause set N_0 and keep deriving new clauses until $\bot$ is generated or no inferences are possible. Proving processes support an important optimization: redundant clauses can be deleted at any point from the clause set, and redundant inferences need not be performed at all.

A derivation performed by a proving process is a possibly infinite sequence $N_0 \rhd N_1 \rhd N_2 \rhd \cdots$, where $\rhd$ relates clause sets (`Proving_Process.thy`). In Isabelle, such sequences are captured by lazy lists, a codatatype [5] generated by $\mathsf{LNil} :: {'a}\ \mathit{llist}$ and $\mathsf{LCons} :: {'a} \Rightarrow {'a}\ \mathit{llist} \Rightarrow {'a}\ \mathit{llist}$, and equipped with lhd ("head") and ltl ("tail") selectors that extract LCons's arguments. Unlike datatypes, codatatypes allow infinite values—e.g., $\mathsf{LCons}\ 0\ (\mathsf{LCons}\ 1\ (\mathsf{LCons}\ 2\ \ldots))$. The coinductive predicate chain checks that its argument is a nonempty lazy list whose elements are consecutively related by a given binary predicate R:

```
coinductive chain :: $({'a} \Rightarrow {'a} \Rightarrow \mathit{bool}) \Rightarrow {'a}\ \mathit{llist} \Rightarrow \mathit{bool}$ where
  $\mathsf{chain}\ R\ (\mathsf{LCons}\ x\ \mathsf{LNil})$
| $\mathsf{chain}\ R\ \mathit{xs} \Longrightarrow R\ x\ (\mathsf{lhd}\ \mathit{xs}) \Longrightarrow \mathsf{chain}\ R\ (\mathsf{LCons}\ x\ \mathit{xs})$
```

A derivation is a lazy list Ns of clause sets satisfying the chain predicate with $R = \rhd$. Derivations depend on a redundancy criterion presented as two functions, $\mathcal{R}_\mathcal{F}$ and $\mathcal{R}_\mathcal{I}$, that specify redundant clauses and redundant inferences, respectively:

```
locale redundancy_criterion = inference_system +
  fixes
    $\mathcal{R}_\mathcal{F} :: {'a}\ \mathit{clause\ set} \Rightarrow {'a}\ \mathit{clause\ set}$ and
    $\mathcal{R}_\mathcal{I} :: {'a}\ \mathit{clause\ set} \Rightarrow {'a}\ \mathit{inference\ set}$
  assumes
    $\mathcal{R}_\mathcal{I}\ N \subseteq \Gamma$ and
    $N \subseteq N' \Longrightarrow \mathcal{R}_\mathcal{F}\ N \subseteq \mathcal{R}_\mathcal{F}\ N'$ and
    $N \subseteq N' \Longrightarrow \mathcal{R}_\mathcal{I}\ N \subseteq \mathcal{R}_\mathcal{I}\ N'$ and
    $N' \subseteq \mathcal{R}_\mathcal{F}\ N \Longrightarrow \mathcal{R}_\mathcal{F}\ N \subseteq \mathcal{R}_\mathcal{F}\ (N \setminus N')$ and
    $N' \subseteq \mathcal{R}_\mathcal{F}\ N \Longrightarrow \mathcal{R}_\mathcal{I}\ N \subseteq \mathcal{R}_\mathcal{I}\ (N \setminus N')$ and
    $\mathsf{sat}\ (N \setminus \mathcal{R}_\mathcal{F}\ N) \Longrightarrow \mathsf{sat}\ N$
```

By definition, a transition from M to N is possible if the only new clauses added are conclusions of inferences from M and any deleted clauses would be redundant in N:

```
inductive $\rhd$ :: ${'a}\ \mathit{clause\ set} \Rightarrow {'a}\ \mathit{clause\ set} \Rightarrow \mathit{bool}$ where
  $N \setminus M \subseteq \mathsf{concl\_of}\ `\ \mathsf{infers\_from}\ M \Longrightarrow M \setminus N \subseteq \mathcal{R}_\mathcal{F}\ N \Longrightarrow M \rhd N$
```

This rule combines deduction (the addition of inferred clauses) and deletion (the removal of redundant clauses) in a single transition. The chapter keeps the two operations separated, but this is problematic, as we will see in Sect. 7.

A key concept to connect static and dynamic completeness is that of the set of persistent clauses, or limit of a sequence of clause sets: $N_\infty = \bigcup_i \bigcap_{j \geq i} N_j$. These are the clauses that belong to all clause sets except for at most a finite prefix of the sequence N_i. We also need the supremum of a sequence, $\bigcup_i N_i$, and of a bounded prefix, $\bigcup_{i=0}^{j} N_i$. We introduce these missing functions (`Lazy_List_Liminf.thy`):

> **definition** $\mathsf{Liminf} :: {'a}\ \mathit{llist} \Rightarrow {'a}$ **where**
> $\quad \mathsf{Liminf}\ \mathit{xs} = \bigcup_{i<|\mathit{xs}|} \bigcap_{j:i\leq j<|\mathit{xs}|} \mathit{xs}\ !\ j$
>
> **definition** $\mathsf{Sup} :: {'a}\ \mathit{llist} \Rightarrow {'a}$ **where**
> $\quad \mathsf{Sup}\ \mathit{xs} = \bigcup_{i<|\mathit{xs}|} \mathit{xs}\ !\ i$
>
> **definition** $\mathsf{Sup\_upto} :: {'a}\ \mathit{llist} \Rightarrow \mathit{nat} \Rightarrow {'a}$ **where**
> $\quad \mathsf{Sup\_upto}\ \mathit{xs}\ j = \bigcup_{i:i<|\mathit{xs}| \wedge i\leq j} \mathit{xs}\ !\ i$

Even though codatatypes open the door to coinductive methods, we follow whenever possible the chapter's index-based approach. When interpreting the notation $\bigcup_i \bigcap_{j\geq i} N_j$ for the case of a finite sequence of length n, it is crucial to use the right upper bounds, namely $i, j < n$. For j, it is clear that '$< n$' is needed to keep N_j's index within bounds. For i, the danger is more subtle: if $i \geq n$, then $\bigcap_{j\,:\,i\leq j<n} N_j$ collapses to the trivial infimum $\bigcap_{j\in\{\}} N_j$, i.e., the set of all clauses.

Lemma 4.2 connects the redundant clauses and inferences at the limit to those of the supremum, and the satisfiability of the limit to that of the initial clause set. Formally:

> **lemma** *Rf_limit_Sup*: $\mathsf{chain}\ (\triangleright)\ \mathit{Ns} \Longrightarrow \mathcal{R}_{\mathcal{F}}\ (\mathsf{Liminf}\ \mathit{Ns}) = \mathcal{R}_{\mathcal{F}}\ (\mathsf{Sup}\ \mathit{Ns})$
> **lemma** *Ri_limit_Sup*: $\mathsf{chain}\ (\triangleright)\ \mathit{Ns} \Longrightarrow \mathcal{R}_{\mathcal{I}}\ (\mathsf{Liminf}\ \mathit{Ns}) = \mathcal{R}_{\mathcal{I}}\ (\mathsf{Sup}\ \mathit{Ns})$
> **lemma** *sat_limit_iff*: $\mathsf{chain}\ (\triangleright)\ \mathit{Ns} \Longrightarrow \big(\mathsf{sat}\ (\mathsf{Liminf}\ \mathit{Ns}) \Leftrightarrow \mathsf{sat}\ (\mathsf{lhd}\ \mathit{Ns})\big)$

The proof of the last lemma relies on

> **lemma** *deriv_sat_preserving*: $\mathsf{chain}\ (\triangleright)\ \mathit{Ns} \Longrightarrow \mathsf{sat}\ (\mathsf{lhd}\ \mathit{Ns}) \Longrightarrow \mathsf{sat}\ (\mathsf{Sup}\ \mathit{Ns})$

In the chapter, this property follows "by the soundness of the inference system Γ and the compactness of clausal logic," contradicting the claim that "we will only consider consistency-preserving inference systems" [3, Sect. 2.4] and not sound ones. Thanks to Isabelle, we now know that soundness is unnecessary. By compactness, it suffices to show that all finite subsets $\mathcal{D}$ of $\bigcup_i N_i$ are satisfiable. By finiteness of $\mathcal{D}$, there must exist an index k such that $\mathcal{D} \subseteq \bigcup_{i=0}^{k} N_i$. We perform an induction on k. The base case is trivial since N_0 is assumed to be satisfiable. For the induction step, if k is beyond the end of the list, then $\bigcup_{i=0}^{k} N_i = \bigcup_{i=0}^{k-1} N_i$ and we can apply the induction hypothesis directly. Otherwise, we have that the set $\mathsf{Sup\_upto}\ \mathit{Ns}\ (k-1) \cup \mathsf{concl\_of}\ `\ \mathsf{infers\_from}\ (\mathsf{Sup\_upto}\ \mathit{Ns}\ (k-1))$ is satisfiable by the induction hypothesis and satisfiability preservation of Γ inferences. Hence, $\mathsf{Sup\_upto}\ \mathit{Ns}\ (k-1) \cup \mathit{Ns}!k$, i.e., $\mathsf{Sup\_upto}\ \mathit{Ns}\ k$, is satisfiable, as desired.

Next, we show that the limit is saturated, under some assumptions and for a relaxed notion of saturation. A clause set N is saturated up to redundancy if all inferences from nonredundant clauses in N are redundant:

> **definition** $\mathsf{saturated\_upto} :: {'a}\ \mathit{clause\ set} \Rightarrow \mathit{bool}$ **where**
> $\quad \mathsf{saturated\_upto}\ N \Longleftrightarrow \mathsf{infers\_from}\ (N \setminus \mathcal{R}_{\mathcal{F}}\ N) \subseteq \mathcal{R}_{\mathcal{I}}\ N$

The limit is saturated for fair derivations—derivations in which no inferences from nonredundant persisting clauses are delayed indefinitely:

> **definition** $\mathsf{fair\_clss\_seq} :: {'a}\ \mathit{clause\ set\ llist} \Rightarrow \mathit{bool}$ **where**
> $\quad \mathsf{fair\_clss\_seq}\ \mathit{Ns} \Longleftrightarrow \mathrm{let}\ N' = \mathsf{Liminf}\ \mathit{Ns} \setminus \mathcal{R}_{\mathcal{F}}\ (\mathsf{Liminf}\ \mathit{Ns})\ \mathrm{in}$
> $\qquad \mathsf{concl\_of}\ `\ \mathsf{infers\_from}\ N' \setminus \mathcal{R}_{\mathcal{I}}\ N' \subseteq \mathsf{Sup}\ \mathit{Ns} \cup \mathcal{R}_{\mathcal{F}}\ (\mathsf{Sup}\ \mathit{Ns})$

The criterion must also be effective, which is expressed by a locale:

locale *effective_redundancy_criterion* = *redundancy_criterion* +
assumes $\gamma \in \Gamma \Longrightarrow \mathsf{concl\_of}\ \gamma \in N \cup \mathcal{R}_{\mathcal{F}}\ N \Longrightarrow \gamma \in \mathcal{R}_{\mathcal{I}}\ N$

In a locale that combines *sat_preserving_inference_system* and *effective_redundancy_criterion*, we have Theorem 4.3:

theorem *fair_derive_saturated_upto*:
$\mathsf{chain}\ (\rhd)\ \mathit{Ns} \Longrightarrow \mathsf{fair\_clss\_seq}\ \mathit{Ns} \Longrightarrow \mathsf{saturated\_upto}\ (\mathsf{Liminf}\ \mathit{Ns})$

It is easy to show that the trivial criterion defined by $\mathcal{R}_{\mathcal{F}}\ N = \{\}$ and $\mathcal{R}_{\mathcal{I}}\ N = \{\gamma \in \Gamma \mid \mathsf{concl\_of}\ \gamma \in N\}$ satisfies the requirements on *effective_redundancy_criterion*. A more useful instance is the standard redundancy criterion, which depends on a counterexample-reducing inference system Γ (`Standard_Redundancy.thy`):

definition $\mathcal{R}_{\mathcal{F}}$ :: *'a clause set* ⇒ *'a clause set* **where**
$\mathcal{R}_{\mathcal{F}}\ N = \{C \mid \exists \mathcal{D} \subseteq N.\ (\forall I.\ I \vDash \mathcal{D} \Longrightarrow I \vDash C) \wedge (\forall D \in \mathcal{D}.\ D < C)\}$

definition $\mathcal{R}_{\mathcal{I}}$ :: *'a clause set* ⇒ *'a inference set* **where**
$\mathcal{R}_{\mathcal{I}}\ N = \{\gamma \in \Gamma \mid \exists \mathcal{D} \subseteq N.\ (\forall I.\ I \vDash \mathcal{D} \uplus \mathsf{side\_prems\_of}\ \gamma \Longrightarrow I \vDash \mathsf{concl\_of}\ \gamma) \wedge$
$(\forall D \in \mathcal{D}.\ D < \mathsf{main\_prem\_of}\ \gamma)\}$

Standard redundancy qualifies as *effective_redundancy_criterion*. In the chapter, this is stated as Theorems 4.7 and 4.8, which depend on two auxiliary properties, Lemmas 4.5 and 4.6. The main result, Theorem 4.9, is that counterexample-reducing calculi are refutationally complete also under the application of standard redundancy:

theorem *saturated_upto_complete*: $\mathsf{saturated\_upto}\ N \Longrightarrow (\neg\ \mathsf{sat}\ N \Longleftrightarrow \{\} \in N)$

The informal proof of Lemma 4.6 applies Lemma 4.5 in a seemingly impossible way, confusing redundant clauses and redundant inferences and exploiting properties that appear only in the first lemma's proof. Our solution is to generalize the core argument into the following lemma and apply it to prove Lemmas 4.5 and 4.6:

lemma *wlog_non_Rf*:
$(\exists \mathcal{D} \subseteq N.\ (\forall I.\ I \vDash \mathcal{D} \uplus \mathcal{C} \Longrightarrow I \vDash E) \wedge (\forall D' \in \mathcal{D}.\ D' < D)) \Longrightarrow$
$\exists \mathcal{D} \subseteq N \setminus \mathcal{R}_{\mathcal{F}}\ N.\ (\forall I.\ I \vDash \mathcal{D} \uplus \mathcal{C} \Longrightarrow I \vDash E) \wedge (\forall D' \in \mathcal{D}.\ D' < D)$

Incidentally, the informal proof of Theorem 4.9 also needlessly invokes Lemma 4.5.

Finally, given a redundancy criterion $(\mathcal{R}_{\mathcal{F}}, \mathcal{R}_{\mathcal{I}})$ for Γ, its standard extension for $\Gamma' \supseteq \Gamma$ is defined as $(\mathcal{R}_{\mathcal{F}}, \mathcal{R}'_{\mathcal{I}})$, where $\mathcal{R}'_{\mathcal{I}}\ N = \mathcal{R}_{\mathcal{I}}\ N \cup (\Gamma' \setminus \Gamma)$ (`Proving_Process.thy`). The standard extension is itself a redundancy criterion and it preserves effectiveness, saturation up to redundancy, and fairness. In Isabelle, this can be expressed by leaving the locales and using the locale predicates—explicit predicates named after the locales and parameterized by the locale arguments:

lemma *standard_redundancy_criterion_extension*:
$\Gamma \subseteq \Gamma' \Longrightarrow \mathsf{redundancy\_criterion}\ \Gamma\ \mathcal{R}_{\mathcal{F}}\ \mathcal{R}_{\mathcal{I}} \Longrightarrow \mathsf{redundancy\_criterion}\ \Gamma'\ \mathcal{R}_{\mathcal{F}}\ \mathcal{R}'_{\mathcal{I}}$

lemma *standard_redundancy_criterion_extension_effective*:
$\Gamma \subseteq \Gamma' \Longrightarrow \mathsf{effective\_redundancy\_criterion}\ \Gamma\ \mathcal{R}_{\mathcal{F}}\ \mathcal{R}_{\mathcal{I}} \Longrightarrow$
$\mathsf{effective\_redundancy\_criterion}\ \Gamma'\ \mathcal{R}_{\mathcal{F}}\ \mathcal{R}'_{\mathcal{I}}$

lemma *standard_redundancy_criterion_extension_saturated_upto_iff*:
$\Gamma \subseteq \Gamma' \Longrightarrow \mathsf{redundancy\_criterion}\ \Gamma\ \mathcal{R}_{\mathcal{F}}\ \mathcal{R}_{\mathcal{I}} \Longrightarrow$
$(\mathsf{redundancy\_criterion.saturated\_upto}\ \Gamma\ \mathcal{R}_{\mathcal{F}}\ \mathcal{R}_{\mathcal{I}}\ N \Longleftrightarrow$
$\mathsf{redundancy\_criterion.saturated\_upto}\ \Gamma'\ \mathcal{R}_{\mathcal{F}}\ \mathcal{R}'_{\mathcal{I}}\ N)$

lemma *standard_redundancy_criterion_extension_fair_iff*:
  $\Gamma \subseteq \Gamma' \Longrightarrow$ effective_redundancy_criterion $\Gamma\ \mathcal{R}_\mathcal{F}\ \mathcal{R}_\mathcal{I} \Longrightarrow$
  (effective_redundancy_criterion.fair_clss_seq $\Gamma'\ \mathcal{R}_\mathcal{F}\ \mathcal{R}'_\mathcal{I}\ Ns \Longleftrightarrow$
   effective_redundancy_criterion.fair_clss_seq $\Gamma\ \mathcal{R}_\mathcal{F}\ \mathcal{R}_\mathcal{I}\ Ns$)

6 First-Order Resolution

The chapter's Sect. 4.3 presents a first-order version of the ordered resolution rule and a first-order prover, RP, based on that rule. The first step towards lifting the completeness of ground resolution is to show that we can lift individual ground resolution inferences (`FO_Ordered_Resolution.thy`).

Inference Rule. First-order ordered resolution consists of a single rule. In the chapter, ground and first-order resolution are both called $O_S^{\succ}$. In the formalization, we also let the rules share the same name, but since they exist in separate locales the system generates qualified names which make this unambiguous: Isabelle generates the name ground_resolution_with_selection.ord_resolve, which refers to ground resolution, and FO_resolution.ordered_resolve, which refers to first-order resolution. If the user is in doubt at any time, the system can always tell which one is meant.

The rule is given as

$$\frac{C_1 \vee A_{11} \vee \cdots \vee A_{1k_1} \quad \cdots \quad C_n \vee A_{n1} \vee \cdots \vee A_{nk_n} \quad \neg A_1 \vee \cdots \vee \neg A_n \vee D}{C_1 \cdot \sigma \vee \cdots \vee C_n \cdot \sigma \vee D \cdot \sigma}$$

where σ is the (canonical) MGU that solves all unification problems $A_{i1} \overset{?}{=} \cdots \overset{?}{=} A_{ik_i} \overset{?}{=} A_i$, for $1 \leq i \leq n$. As expected, the rule has several side conditions. The Isabelle representation of this rule is based on that of its ground counterpart, generalized to apply σ:

inductive ord_resolve :: *'a clause list* $\Rightarrow$ *'a clause* $\Rightarrow$ *'s* $\Rightarrow$ *'a clause* $\Rightarrow$ *bool* **where**
  $|CAs| = n \Longrightarrow |Cs| = n \Longrightarrow |\mathcal{A}s| = n \Longrightarrow |As| = n \Longrightarrow n \neq 0 \Longrightarrow$
  $(\forall i < n.\ CAs\ !\ i = Cs\ !\ i \uplus$ Pos ‘ $\mathcal{A}s\ !\ i) \Longrightarrow (\forall i < n.\ \mathcal{A}s\ !\ i \neq \{\}) \Longrightarrow$
  Some σ = mgu (set_mset ‘ set (map2 add_mset $As\ \mathcal{A}s$)) $\Longrightarrow$
  eligible $\sigma\ As$ ($D \uplus$ Neg ‘ mset As) $\Longrightarrow$
  $(\forall i < n.$ strict_max_in $(As\ !\ i \cdot \sigma)\ (Cs\ !\ i \cdot \sigma)) \Longrightarrow (\forall i < n.\ S\ (CAs\ !\ i) = \{\}) \Longrightarrow$
  ord_resolve CAs ($D \uplus$ Neg ‘ mset As) σ ((($\bigcup$ mset Cs) $\uplus D) \cdot \sigma$)

The rule as stated is incomplete; for example, $\mathsf{p}(x)$ and $\neg\,\mathsf{p}(\mathsf{f}(x))$ cannot be resolved because x and $\mathsf{f}(x)$ are not unifiable. Such issues arise when the same variable names appear in different premises. In the chapter, the authors circumvent this issue by stating, "We also implicitly assume that different premises and the conclusion have no variables in common; variables are renamed if necessary." For the formalization, we first considered enforcing the invariant that all derived clauses use mutually disjoint variables, but this does not help when a clause is repeated in an inference's premises. An example is the inference

$$\frac{\mathsf{p}(x) \quad \mathsf{p}(y) \quad \neg\,\mathsf{p}(\mathsf{a}) \vee \neg\,\mathsf{p}(\mathsf{b})}{\bot}$$

where $\mathsf{p}(x)$ and $\mathsf{p}(y)$ are the same clause up to renaming. Instead, we rely on a predicate ord_resolve_rename, based on ord_resolve, that standardizes the premises apart. The renaming is performed by a function called renamings_apart :: *'a clause list* $\Rightarrow$ *'s list* that, given a list of clauses, returns a list of corresponding substitutions to apply. This function is part of the abstract interface for terms and substitutions (which we presented in Sect. 2) and is implemented using IsaFoR.

Like for the ground case, it is important to establish soundness. We prove that any ground instance of the rule ord_resolve is sound:

> **lemma** *ord_resolve_ground_inst_sound*:
> $\mathsf{ord\_resolve}\ CAs\ DA\ \mathcal{A}s\ As\ \sigma\ E \Longrightarrow I \models \mathsf{mset}\ CAs \cdot \sigma \cdot \eta \Longrightarrow I \models DA \cdot \sigma \cdot \eta \Longrightarrow$
> $\mathsf{is\_ground\_subst}\ \eta \Longrightarrow I \models E \cdot \eta$

Likewise, ground instances of ord_resolve_rename are sound. It then follows that the rules ord_resolve and ord_resolve_rename are sound:

> **lemma** *ord_resolve_rename_sound*:
> $\mathsf{ord\_resolve\_rename}\ CAs\ DA\ \mathcal{A}s\ As\ \sigma\ E \Longrightarrow$
> $(\forall\sigma.\ \mathsf{is\_ground\_subst}\ \sigma \Longrightarrow I \models (\mathsf{mset}\ CAs + \{DA\}) \cdot \sigma) \Longrightarrow$
> $\mathsf{is\_ground\_subst}\ \eta \Longrightarrow I \models E \cdot \eta$

Lifting Lemma. To lift ground inferences to the first-order level, we consider a set of clauses M and introduce an adjusted version S_M of the selection function S.

> **definition** S_M :: *'a literal multiset ⇒ 'a literal multiset* **where**
> $S_M\ C =$
> $(\text{if } C \in \mathsf{grounding\_of\_clss}\ M \text{ then}$
> $\quad (\text{SOME } C'.\ \exists D \in M.\ \exists\sigma.\ C = D \cdot \sigma \wedge C' = S\ D \cdot \sigma \wedge \mathsf{is\_ground\_subst}\ \sigma)$
> else
> $\quad S\ C)$

Here SOME is Hilbert's epsilon operator, which picks an element as described if it exists and an arbitrary one otherwise. In this definition the element does exists, and so we need not worry about it picking an arbitrary one. The new selection function depends on both S and M and works in such a way that any ground instance inherits the selection of at least one of the nonground clauses of which it is an instance. This property is captured formally as

> **lemma** *S_M_grounding_of_clss*:
> $C \in \mathsf{grounding\_of}\ M \Longrightarrow$
> $\exists D \in M.\ \exists\sigma.\ C = D \cdot \sigma \wedge S_M\ C = S\ D \cdot \sigma \wedge \mathsf{is\_ground\_subst}\ \sigma$

where grounding_of M is the set of ground instances of a set of clauses M.

The lifting lemma, Lemma 4.12, states that whenever there exists a ground inference of E from clauses belonging to grounding_of M, there exists a (possibly) more general inference from clauses belonging to M:

> **lemma** *ord_resolve_rename_lifting*:
> $(\forall\rho\ C.\ \mathsf{is\_renaming}\ \rho \Longrightarrow S\ (C \cdot \rho) = S\ C \cdot \rho) \Longrightarrow$
> $\mathsf{ord\_resolve}\ S_M\ CAs\ DA\ \mathcal{A}s\ As\ \sigma\ E \Longrightarrow$
> $\{DA\} \cup \mathsf{set}\ CAs \subseteq \mathsf{grounding\_of}\ M \Longrightarrow$
> $\exists\eta s\ \eta\ \theta\ CAs_0\ DA_0\ \mathcal{A}s_0\ As_0\ E_0\ \tau.$
> $\quad \mathsf{ord\_resolve\_rename}\ S\ CAs_0\ DA_0\ \mathcal{A}s_0\ As_0\ \tau\ E_0 \wedge$
> $\quad CAs_0 \cdot \eta s = CAs \wedge DA_0 \cdot \eta = DA \wedge E_0 \cdot \theta = E \wedge \{DA_0\} \cup \mathsf{set}\ CAs_0 \subseteq M$

The informal proof of this lemma consists of two sentences spanning four lines of text. In Isabelle, these two sentences translate to 250 lines and 400 lines, respectively, excluding auxiliary lemmas. Our proof involves six steps:

1. Obtain a list of first-order clauses CAs_0 and a first-order clause DA_0 that belong to M and that generalize CAs and DA with substitutions ηs and η, respectively.

2. Choose atoms $\mathcal{A}s_0$ and As_0 in the first-order clauses on which to resolve.
3. Standardize CAs_0 and DA_0 apart, yielding CAs'_0 and DA'_0.
4. Obtain the MGU τ of the literals on which to resolve.
5. Show that ordered resolution on CAs'_0 and DA'_0 with τ as MGU is applicable.
6. Show that the resulting resolvent E_0 generalizes E with substitution θ.

In step 1, suitable clauses must be chosen so that $S\ (CAs_0\ !\ i)$ generalizes $S_M\ (CAs\ !\ i)$, for $0 \le i < n$, and $S\ DA_0$ generalizes $S_M\ DA$. By the definition of S_M, this is always possible. In step 2, we choose the literals to resolve upon in the first-order inference depending on the selection on the ground inference. If some literals are selected in DA, we define As_0 as the selected literals in DA_0, such that $(As_0\ !\ i) \cdot \eta = As\ !\ i$ for each i. Otherwise, As must be a singleton list containing some atom A, and we define As_0 as the singleton list consisting of an arbitrary $A_0 \in DA_0$ such that $A_0 \cdot \eta = A$. Step 3 may seem straightforward until one realizes that renaming variables can in principle influence selection. To rule this out, our lemma assumes stability under renaming: $S\ (C \cdot \rho) = S\ C \cdot \rho$ for any renaming substitution ρ and clause C. This requirement seems natural, but it is not mentioned in the chapter.

The above choices allow us to perform steps 4 to 6. In the chapter, the authors assume that the obtained CAs_0 and DA_0 are standardized apart from each other as well as their conclusion E_0. This means that they can obtain a single ground substitution μ that connect CAs_0, DA_0, E_0 to CAs, DA, E. By contrast, we provide separate substitutions ηs, η, θ for the different side premises, the main premise, and the conclusion.

7 A First-Order Prover

Modern resolution provers interleave inference steps with steps that delete or reduce (simplify) clauses. In their Sect. 4.3, Bachmair and Ganzinger introduce the nondeterministic abstract prover RP that works on triples of clause sets and that generalizes the Otter-style and DISCOUNT-style loops [12,17]. RP's core rule, called inference computation, performs first-order ordered resolution as described above; the other rules delete or reduce clauses or move them between clause sets. We formalize RP and prove it complete assuming a fair strategy (`FO_Ordered_Resolution_Prover.thy`).

Abstract First-Order Prover. The RP prover is a relation $\leadsto$ on states of the form $(\mathcal{N}, \mathcal{P}, \mathcal{O})$, where $\mathcal{N}$ is the set of *new clauses*, $\mathcal{P}$ is the set of *processed clauses*, and $\mathcal{O}$ is the set of *old clauses*. RP's formal definition is very close to the original formulation:

inductive $\leadsto$ :: *'a state* $\Rightarrow$ *'a state* $\Rightarrow$ *bool* **where**
  Neg $A \in C \Longrightarrow$ Pos $A \in C \Longrightarrow (\mathcal{N} \cup \{C\}, \mathcal{P}, \mathcal{O}) \leadsto (\mathcal{N}, \mathcal{P}, \mathcal{O})$
| $D \in \mathcal{P} \cup \mathcal{O} \Longrightarrow$ subsumes $D\ C \Longrightarrow (\mathcal{N} \cup \{C\}, \mathcal{P}, \mathcal{O}) \leadsto (\mathcal{N}, \mathcal{P}, \mathcal{O})$
| $D \in \mathcal{N} \Longrightarrow$ strictly_subsumes $D\ C \Longrightarrow (\mathcal{N}, \mathcal{P} \cup \{C\}, \mathcal{O}) \leadsto (\mathcal{N}, \mathcal{P}, \mathcal{O})$
| $D \in \mathcal{N} \Longrightarrow$ strictly_subsumes $D\ C \Longrightarrow (\mathcal{N}, \mathcal{P}, \mathcal{O} \cup \{C\}) \leadsto (\mathcal{N}, \mathcal{P}, \mathcal{O})$
| $D \in \mathcal{P} \cup \mathcal{O} \Longrightarrow$ reduces $D\ C\ L \Longrightarrow (\mathcal{N} \cup \{C \uplus \{L\}\}, \mathcal{P}, \mathcal{O}) \leadsto (\mathcal{N} \cup \{C\}, \mathcal{P}, \mathcal{O})$
| $D \in \mathcal{N} \Longrightarrow$ reduces $D\ C\ L \Longrightarrow (\mathcal{N}, \mathcal{P} \cup \{C \uplus \{L\}\}, \mathcal{O}) \leadsto (\mathcal{N}, \mathcal{P} \cup \{C\}, \mathcal{O})$
| $D \in \mathcal{N} \Longrightarrow$ reduces $D\ C\ L \Longrightarrow (\mathcal{N}, \mathcal{P}, \mathcal{O} \cup \{C \uplus \{L\}\}) \leadsto (\mathcal{N}, \mathcal{P} \cup \{C\}, \mathcal{O})$
| $(\mathcal{N} \cup \{C\}, \mathcal{P}, \mathcal{O}) \leadsto (\mathcal{N}, \mathcal{P} \cup \{C\}, \mathcal{O})$
| $(\{\}, \mathcal{P} \cup \{C\}, \mathcal{O}) \leadsto$ (concl_of ‘ infers_between $\mathcal{O}\ C, \mathcal{P}, \mathcal{O} \cup \{C\}$)

The rules correspond, respectively, to tautology deletion, forward subsumption, backward subsumption in $\mathcal{P}$ and $\mathcal{O}$, forward reduction, backward reduction in $\mathcal{P}$ and $\mathcal{O}$, clause processing, and inference computation.

Initially, $\mathcal{N}$ consists of the problem clauses and the other two sets are empty. Clauses in $\mathcal{N}$ are reduced using $\mathcal{P} \cup \mathcal{O}$, or even deleted if they are tautological or subsumed by $\mathcal{P} \cup \mathcal{O}$; conversely, $\mathcal{N}$ can be used for reducing or subsuming clauses in $\mathcal{P} \cup \mathcal{O}$. Clauses eventually move from $\mathcal{N}$ to $\mathcal{P}$, one at a time. As soon as $\mathcal{N}$ is empty, a clause from $\mathcal{P}$ is selected to move to $\mathcal{O}$. Then all possible resolution inferences between this given clause and the clauses in $\mathcal{O}$ are computed and put in $\mathcal{N}$, closing the loop.

The subsumption and reduction rules depend on the following predicates:

$$\begin{aligned}
\mathsf{subsumes}\ D\ C &\iff \exists\sigma.\ D \cdot \sigma \subseteq C \\
\mathsf{strictly\_subsumes}\ D\ C &\iff \mathsf{subsumes}\ D\ C \land \neg\ \mathsf{subsumes}\ C\ D \\
\mathsf{reduces}\ D\ C\ L &\iff \exists D'\ L'\ \sigma.\ D = D' \uplus \{L'\} \land -L = L' \cdot \sigma \land D' \cdot \sigma \subseteq C
\end{aligned}$$

The definition of the set $\mathsf{infers\_between}\ \mathcal{O}\ C$, on which inference computation depends, is more subtle. In the chapter, the set of inferences between C and $\mathcal{O}$ consists of all inferences from $\mathcal{O} \cup \{C\}$ that have C as *exactly one* of their premises. This, however, leads to an incomplete prover, because it ignores inferences that need multiple copies of C. For example, assuming a maximal selection function (one that always returns all negative literals), the resolution inference

$$\frac{\mathsf{p} \quad \mathsf{p} \quad \neg\,\mathsf{p} \lor \neg\,\mathsf{p}}{\bot}$$

is possible. Yet if the clause $\neg\,\mathsf{p} \lor \neg\,\mathsf{p}$ reaches $\mathcal{O}$ earlier than p, the inference would not be performed. This counterexample requires ternary resolution, but there also exists a more complicated one for binary resolution, where both premises are the same clause. Consider the clause set containing

$$(1)\ \ \mathsf{q}(\mathsf{a},\mathsf{c},\mathsf{b}) \qquad\qquad (2)\ \ \neg\,\mathsf{q}(x,y,z) \lor \mathsf{q}(y,z,x) \qquad\qquad (3)\ \ \neg\,\mathsf{q}(\mathsf{b},\mathsf{a},\mathsf{c})$$

and an order $>$ on atoms such that $\mathsf{q}(\mathsf{c},\mathsf{b},\mathsf{a}) > \mathsf{q}(\mathsf{b},\mathsf{a},\mathsf{c}) > \mathsf{q}(\mathsf{a},\mathsf{c},\mathsf{b})$. Inferences between (1) and (2) or between (2) and (3) are impossible due to order restrictions. The only possible inference involves two copies of (2):

$$\frac{\neg\,\mathsf{q}(x,y,z) \lor \mathsf{q}(y,z,x) \quad \neg\,\mathsf{q}(x',y',z') \lor \mathsf{q}(y',z',x')}{\neg\,\mathsf{q}(x,y,z) \lor \mathsf{q}(z,x,y)}$$

From the conclusion, we derive $\neg\,\mathsf{q}(\mathsf{a},\mathsf{c},\mathsf{b})$ by (3) and $\bot$ by (1). This incompleteness is a severe flaw, although it is probably just an oversight. Fortunately, it can easily be repaired by defining $\mathsf{infers\_between}\ \mathcal{O}\ C$ as $\{(\mathcal{C}, D, E) \in \Gamma \mid \mathcal{C} \cup \{D\} \subseteq \mathcal{O} \cup \{C\} \land C \in \mathcal{C} \cup \{D\}\}$.

Projection to Theorem Proving Process. On the first-order level, a derivation can be expressed as a lazy list $\mathcal{S}s$ of states, or as three parallel lazy lists $\mathcal{N}s$, $\mathcal{P}s$, $\mathcal{O}s$. The limit state of a derivation $\mathcal{S}s$ is defined as $\mathsf{Liminf}\ \mathcal{S}s = (\mathsf{Liminf}\ \mathcal{N}s,\ \mathsf{Liminf}\ \mathcal{P}s,\ \mathsf{Liminf}\ \mathcal{O}s)$, where Liminf on the right-hand side is as in Sect. 5.

Bachmair and Ganzinger use the completeness of ground resolution to prove RP complete. The first step is to show that first-order derivations can be projected down to theorem proving processes on the ground level. This corresponds to Lemma 4.10. Adapted to our conventions, its statement is as follows:

If $\mathcal{S} \rightsquigarrow \mathcal{S}'$, then $\mathsf{grounding\_of}\ \mathcal{S} \rhd^* \mathsf{grounding\_of}\ \mathcal{S}'$, with $\rhd$ based on some extension of ordered resolution with selection function S and the standard redundancy criterion $(\mathcal{R}_\mathcal{F}, \mathcal{R}_\mathcal{I})$.

This raises some questions: (1) Exactly which instance of the calculus are we extending? (2) Which calculus extension should we use? (3) How can we repair the mismatch between $\rhd^*$ in the lemma statement and $\rhd$ where the lemma is invoked?

Regarding question (1), it is not clear which selection function to use. Is the function the same S as in the definition of RP or is it arbitrary? It takes a close inspection of the proof of Lemma 4.13, where Lemma 4.10 is invoked, to find out that the selection function used there is $S_{\mathsf{Liminf}\ \mathcal{O}s}$.

Regarding question (2), the phrase "some extension" is cryptic. It suggests an existential reading, and from the context it would appear that a standard extension (Sect. 5) is meant. However, neither the lemma's proof nor the context where it is invoked supplies the desired existential witness. A further subtlety is that the witness should be independent of $\mathcal{S}$ and $\mathcal{S}'$, so that transitions can be joined to form a single theorem proving derivation. Our approach is to let $\rhd$ be the standard extension for the proof system consisting of all *sound* derivations: $\Gamma = \{(\mathcal{C}, D, E) \mid \forall I.\ I \vDash \mathcal{C} \cup \{D\} \Longrightarrow I \vDash E\}$. This also eliminates the need for Bachmair and Ganzinger's subsumption resolution rule, a special calculus rule that is, from what we understand, implicitly used in the proof of Lemma 4.10 for the subcases associated with RP's reduction rules.

As for question (3), when the lemma is invoked, it is used to join transitions together to whole theorem proving processes. That requires these transitions to be of $\rhd$ – not $\rhd^*$. The need for $\rhd^*$ instead of $\rhd$ arises because one of the cases requires a combination of deduction and deletion, which Bachmair and Ganzinger model as separate transitions. By merging the two transitions (Sect. 5), we avoid the issue altogether and can use $\rhd$ in the formal counterpart of Lemma 4.10.

With these issues resolved, we can prove Lemma 4.10. In Sect. 6 we established that ground instances of the resolution rule are sound. Since our ground proof system consists of all sound inference rules we can reuse that lemma in proving the inference computation case. We prove Lemma 4.10 for single steps and extend it to entire derivations:

lemma *RP_ground_derive*: $\mathcal{S} \rightsquigarrow \mathcal{S}' \Longrightarrow \mathsf{grounding\_of}\ \mathcal{S} \rhd \mathsf{grounding\_of}\ \mathcal{S}'$
lemma *RP_ground_derive_chain*:
  $\mathsf{chain}\ (\rightsquigarrow)\ \mathcal{S}s \Longrightarrow \mathsf{chain}\ (\rhd)\ (\mathsf{lmap}\ \mathsf{grounding\_of}\ \mathcal{S}s)$

The lmap function applies its first argument elementwise to its second argument.

Fairness and Clause Movement. From a given initial state $(\mathcal{N}_0, \{\}, \{\})$, many derivations are possible, reflecting RP's nondeterminism. In some derivations, we could leave a crucial clause in $\mathcal{N}$ or $\mathcal{P}$ without ever reducing it or moving it to $\mathcal{O}$, and then fail to derive $\bot$ even if $\mathcal{N}_0$ is unsatisfiable. For this reason, refutational completeness is guaranteed only for fair derivations. These are defined as derivations such that $\mathsf{Liminf}\ \mathcal{N}s = \mathsf{Liminf}\ \mathcal{P}s = \{\}$, guaranteeing that no clause will stay forever in $\mathcal{N}$ or $\mathcal{P}$.

Fairness is expressed by the fair_state_seq predicate, which is distinct from the fair_clss_seq predicate presented in Sect. 5. In particular, Theorem 4.3 is used in neither the informal nor the formal proof, and appears to play a purely pedagogic role in the chapter. For the rest of this section, we fix a lazy list of states $\mathcal{S}s$, and its projections $\mathcal{N}s$, $\mathcal{P}s$, and $\mathcal{O}s$, such that $\mathsf{chain}\ (\rightsquigarrow)\ \mathcal{S}s$, $\mathsf{fair\_state\_seq}\ \mathcal{S}s$, and $\mathsf{lhd}\ \mathcal{O}s = \{\}$.

Thanks to fairness, any nonredundant clause C in $\mathcal{S}s$'s projection to the ground level eventually ends up in $\mathcal{O}$ and stays there. This is proved informally as Lemma 4.11, but again there are some difficulties. The vagueness concerning the selection function can be resolved as for Lemma 4.10, but there is another, deeper flaw.

Bachmair and Ganzinger's proof idea is as follows. By hypothesis, the ground clause C must be an instance of a first-order clause D in $\mathcal{N}s \mathbin{!} j \cup \mathcal{P}s \mathbin{!} j \cup \mathcal{O}s \mathbin{!} j$ for some index j. If $C \in \mathcal{N}s \mathbin{!} j$, then by nonredundancy of C, fairness of the derivation, and Lemma 4.10, there must exist a clause D' that generalizes C in $\mathcal{P}s \mathbin{!} l \cup \mathcal{O}s \mathbin{!} l$ for some $l > j$. By a similar argument, if D' belongs to $\mathcal{P}s \mathbin{!} l$, it will be in $\mathcal{O}s \mathbin{!} l'$ for some $l' > l$, and finally in all $\mathcal{O}s \mathbin{!} k$ with $k \geq l'$. The flaw is that backward subsumption can delete D' without moving it to $\mathcal{O}$. The subsumer clause would then be a strictly more general version of D' (and of the ground clause C).

Our solution is to choose D, and consequently D', such that it is minimal, with respect to subsumption, among the clauses that generalize C in the derivation. This works because strict subsumption is well founded—which we also proved, by reduction to a well-foundedness result about the strict generalization relation on first-order terms, included in IsaFoR [15, Sect. 2]. By minimality, D' cannot be deleted by backward subsumption. This line of reasoning allows us to prove Lemma 4.11, where $\mathcal{O}$_of extracts the $\mathcal{O}$ component of a state:

lemma *fair_imp_Liminf_minus_Rf_subset_ground_Liminf_state*:
$Gs = \mathsf{lmap\ grounding\_of}\ \mathcal{S}s \Longrightarrow$
$\mathsf{Liminf}\ Gs - \mathcal{R}_{\mathcal{F}}\ (\mathsf{Liminf}\ Gs) \subseteq \mathsf{grounding\_of}\ (\mathcal{O}\mathsf{\_of}\ (\mathsf{Liminf}\ \mathcal{S}s))$

In the formalization of the above proof, we do not prove $l > j$ and $l' > l$. While they guide human intuition, they are not necessary to prove the lemma.

Soundness and Completeness. The chapter's main result is Theorem 4.13, which states that, for fair derivations, the prover is sound and complete. Soundness follows from Lemma 4.2 (*sat_ deriv_Liminf_iff*) and is, perhaps not surprisingly, independent of whether the derivation is fair.

theorem *RP_sound*:
$\{\} \in \mathsf{clss\_of}\ (\mathsf{Liminf}\ Sts) \Longrightarrow \neg\ \mathsf{sat}\ (\mathsf{grounding\_of}\ (\mathsf{lhd}\ Sts))$

Because we have brought Lemmas 4.10, 4.11, and 4.12 into a suitable shape, completeness is not difficult to formalize:

theorem *RP_saturated_if_fair*: $\mathsf{saturated\_upto}\ (\mathsf{Liminf}\ (\mathsf{lmap\ grounding\_of}\ \mathcal{S}s))$

corollary *RP_complete_if_fair*:
$\neg\ \mathsf{sat}\ (\mathsf{grounding\_of}\ (\mathsf{lhd}\ \mathcal{S}s)) \Longrightarrow \{\} \in \mathcal{O}\mathsf{\_of}\ (\mathsf{Liminf}\ \mathcal{S}s)$

A crucial point that is not clear from the text is that we must always use the selection function S on the first-order level and $S_{\mathsf{Liminf}\ \mathcal{O}s}$ on the ground level. Another noteworthy part of the proof is the passage "Liminf Gs (and hence Liminf $\mathcal{S}s$) contains the empty clause" (using our notations). Obviously, if grounding_of (Liminf $\mathcal{S}s$) contains $\bot$, then Liminf $\mathcal{S}s$ must as well. However, the authors do not explain the step from Liminf Gs, the limit of the grounding, to grounding_of (Liminf $\mathcal{S}s$), the grounding of the limit. Fortunately, by Lemma 4.11, the latter contains all the nonredundant clauses of the former, and the empty clause is nonredundant. Hence the informal argument is fundamentally correct. For the other direction, which is used in the soundness proof, we can prove that the former includes the latter.

8 Discussion

Bachmair and Ganzinger cover a lot of ground in a few pages. We found much of the material straightforward to formalize: it took us about two weeks to reach their Sect. 4.3, which introduces the RP prover and establishes its refutational completeness. By contrast, we needed months to fully understand and formalize that section. While the *Handbook* chapter succeeds at conveying the key ideas at the propositional level, the lack of rigor makes it difficult to develop a deep understanding of ordered resolution proving on first-order clauses.

There are several reasons why Sect. 4.3 did not lend itself easily to a formalization. The proofs often depend on lemmas and theorems from previous sections without explicitly mentioning them. The lemmas and proofs do not quite fit together. And while the general idea of the proofs stands up, they have many confusing flaws that must be repaired. Our methodology involved the following steps: (1) rewrite the informal proofs to a handwritten pseudo-Isabelle; (2) fill in the gaps, emphasizing which lemmas are used where; (3) turn the pseudo-Isabelle into real Isabelle, but with **sorry** placeholders for the proofs; and (4) replace the **sorry**s with proofs. Progress was not always linear. As we worked on each step, more than once we discovered an earlier mistake.

The formalization helps us answer questions such as, “Is effectiveness of ordered resolution (Lemma 3.13) actually needed, and if so, where?” (Answer: It is needed to prove Theorem 3.15.) It also allows us to track definitions and hypotheses precisely, so that we always know the scope and meaning of every definition, lemma, or theorem. If a hypothesis appears too strong or superfluous, we can try to rephrase or eliminate it; the proof assistant tells us where the proof breaks. If a definition is changed, the proof assistant tells us where proofs of the related lemmas break. In the best case, the proofs do not break at all since the automation of the proof assistant is flexible enough to still prove them. This happened, for example, when we changed the definition of $\rhd$ to combine deduction and deletion.

Starting from RP, we could refine it to obtain an efficient imperative implementation, following the lines of Fleury, Blanchette, and Lammich’s verified SAT solver with the two-watched-literals optimization [13]. However, this would probably involve a huge amount of work. To increase provers’ trustworthiness, a more practical approach is to have them generate detailed proofs that record all inferences leading to the empty clause [27, 31]. Such output can be independently checked by verified programs or reconstructed using a proof assistant’s inference kernel. This is the approach implemented in Sledgehammer [8], which integrates automatic provers in Isabelle. Formalized metatheory could in principle be used to deduce a formula’s satisfiability from a finite saturation.

We found Isabelle/HOL eminently suitable to this kind of formalization work. Its logic—classical simple type theory extended with polymorphism, type classes, and the axiom of choice—balances expressiveness and automatability. We nowhere felt the need for dependent types. We benefited from many features of the system, including codatatypes [5], Isabelle/jEdit [37], the Isar proof language [36], locales [4], and Sledgehammer [8]. It is perhaps indicative of the maturity of theorem proving technology that most of the issues we encountered were unrelated to Isabelle. The main challenge was to understand the informal proof well enough to design suitable locale hierarchies and state the definitions and lemmas precisely, and correctly.

9 Related Work

Formalizing the metatheory of logic and deduction is an enticing proposition for many researchers in interactive theorem proving. In this section, we briefly review some of the main related work, without claim to exhaustiveness. Two recent, independent developments are particularly pertinent.

Peltier [24] proved static refutational completeness of a variant of the superposition calculus in Isabelle/HOL. Since superposition generalizes ordered resolution, his result subsumes our static completeness theorem. On the other hand, he did not formalize a prover or dynamic completeness, nor did he attempt to develop general infrastructure. It would be interesting to extend his formal development to obtain a verified superposition prover. We could also consider calculus extensions such as polymorphism [11, 34], type classes [34], and AVATAR [33]. Two significant differences between Peltier's work and ours is that he represents clauses as sets instead of multisets (to exploit Isabelle's better proof automation for sets) and that he relies, for terms and unification, on an example theory file included in Isabelle (`Unification.thy`) instead of IsaFoR.

Hirokawa et al. [15] formalized, also in Isabelle/HOL, an abstract Knuth–Bendix completion procedure as well as ordered (unfailing) completion, a method developed by Bachmair, Ganzinger, and Plaisted [1]. Superposition combines ordered resolution (to reason about clauses) and ordered completion (to reason about equality). There are many similarities between their formalization and ours, which is unsurprising given that both follow work by Bachmair and Ganzinger; for example, they need to reason about the limit of fair infinite sequences of sets of equations and rewrite rules to establish completeness.

The literature contains many other formalized completeness proofs, mostly for inference systems of theoretical interest. Early work was carried out in the 1980s and 1990s, notably by Shankar [29] and Persson [25]. Some of our own efforts are also related: completeness of unordered resolution using semantic trees by Schlichtkrull [28]; completeness of a Gentzen system following the Beth–Hintikka style and soundness of a cyclic proof system for first-order logic with inductive definitions by Blanchette, Popescu, and Traytel [9]; and completeness of a SAT solver based on CDCL (conflict-driven clause learning) by Blanchette, Fleury, Lammich, and Weidenbach [6].

The formal Beth–Hintikka-style completeness proof mentioned above has a generic flavor, abstracting over the inference system. Could it be used to prove completeness of the ordered resolution calculus, or even of the RP prover? The central idea is to build a finitely branching tree that encodes a systematic proof attempt. Given a fair strategy for applying calculus rules, infinite branches correspond to countermodels. It should be possible to prove ordered resolution complete using this approach, by storing clause sets N on the tree's nodes. Each node would have at most one child, corresponding to the new clause set after performing a deduction. Such degenerate trees would be isomorphic to derivations $N_0 \rhd N_1 \rhd \cdots$ represented by lazy lists. However, the requirement that inferences can always be postponed, called *persistence* [9, Sect. 3.9], is not met for deletion steps based on a redundancy criterion. Moreover, while the generic framework takes care of applying inferences fairly and of employing König's lemma to extract an infinite path from a failed proof attempt (which is, incidentally, overkill for degenerate trees that have only one infinite path), it offers no help in building a countermodel from an infinite path (i.e., in proving Bachmair and Ganzinger's Theorem 3.9).

Beyond completeness, Gödel's first incompleteness theorem has been formalized in Nqthm by Shankar [30], in Coq by O'Connor [22], in HOL Light by Harrison (in unpublished work), and in Isabelle/HOL by Paulson [23]. Paulson additionally proved Gödel's second incompleteness theorem. We refer to our earlier papers [6, 9, 28] for further discussions of related work.

10 Conclusion

We presented a formal proof that captures the core of Bachmair and Ganzinger's *Handbook* chapter on resolution theorem proving. For all its idiosyncrasies, the chapter withstood the test of formalization, once we had added self-inferences to the RP prover. Given that the text is a basic building block of automated reasoning (as confirmed by its placement as Chapter 2), we believe there is value in clarifying its mathematical content for the next generations of researchers. We hope that our work will be useful to the editors of a future revision of the *Handbook*.

Formalization of the metatheory of logical calculi is one of the many connections between automatic and interactive theorem proving. We expect to see wider adoption of proof assistants by researchers in automated reasoning, as a convenient way to develop metatheory. By building formal libraries of standard results, we aim to make it easier to formalize state-of-the-art research as it emerges. We also see potential uses of formal proofs in teaching automated reasoning, inspired by the use of proof assistants in courses on the semantics of programming languages [19, 26].

Acknowledgment. Christoph Weidenbach discussed Bachmair and Ganzinger's chapter with us on many occasions and hosted Schlichtkrull at the Max-Planck-Institut in Saarbrücken. Christian Sternagel and René Thiemann answered our questions about IsaFoR. Mathias Fleury, Florian Haftmann, and Tobias Nipkow helped enrich and reorganize Isabelle's multiset library. Mathias Fleury, Robert Lewis, Simon Robillard, Mark Summerfield, Sophie Tourret, and the anonymous reviewers suggested many textual improvements.

Blanchette was partly supported by the Deutsche Forschungsgemeinschaft (DFG) project Hardening the Hammer (grant NI 491/14-1). He also received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 713999, Matryoshka). Traytel was partly supported by the DFG program Program and Model Analysis (PUMA, doctorate program 1480).

A Errors and Imprecisions Discovered in the Chapter

We discussed several mathematical errors and imprecisions, of various severity levels, in Bachmair and Ganzinger's chapter. We also found lemmas that are stated but not explicitly applied afterwards. In this appendix, we list our findings exhaustively for reference.

Let us start with the errors and imprecisions. We ignore infelicities that are not mathematical in nature, such as typos and LaTeX macros gone wrong (e.g., "by the defn[candidate model]*candidate model* for N" on page 34); for such errors, careful reading is a more effective antidote than formalization. We also ignore minor ambiguities, such as whether the clause $C \vee A \vee \cdots \vee A$ may contain zero occurrences of A, if they can be resolved easily by appealing to the context and the reader's common sense.

– One of Lemma 3.4's claims is that if clause C is true in I^D, then C is also true in $I_{D'}$, where $C \preceq D \preceq D'$. This does not hold if $C = D = D'$ and C is productive. Similarly, the first sentence of the proof is wrong if $D = D'$ and D is productive: "First, observe that $I_D \subseteq I^D \subseteq I_{D'} \subseteq I^{D'} \subseteq I_N$, whenever $D' \succeq D$."

– The last occurrence of D' in the statement of Lemma 3.7 should be changed to C. In addition, it is not clear whether the phrase "another clause C" implies that $C \neq D$, but the counterexample we gave in Sect. 4 works in both cases. Correspondingly, in the proof, the case distinction is incomplete, as can be seen by specializing the proof for the counterexample.

– In the chapter's Figure 2, in Sect. 3, the selection function is wrongly applied: references to $S(D)$ should be changed to $S(\neg A_1 \vee \cdots \vee \neg A_n \vee D)$. Moreover, in condition (iii), it is not clear with respect to which clause the "selected atom" must be considered, the two candidates being $S(\neg A_1 \vee \cdots \vee \neg A_n \vee D)$ and $S(C_i \vee A_i \vee \cdots \vee A_i)$. We assume the latter is meant. Finally, phrases like "A_1 is maximal with respect to D" (here and in Figure 4) are slightly ambiguous, because it is unclear whether A_1 denotes an atom or a (positive) literal, and whether it must be maximal with respect to D's atoms or literals. From the context, we infer that an atom-with-atom comparison is meant.

– The notation $\bigcup_i \bigcap_{j \geq i} N_j$ used in the chapter's Sect. 4.1 only partially specifies the range of i and j if N_j is a finite sequence. Clearly, j must be bounded by the length of the list, but it is less obvious that i also needs a bound, to avoid the inner intersection to expand to be the set of all clauses for indices i beyond the list's end.

– Soundness is required in the chapter's Sect. 4.1, even though it is claimed in Sect. 2.4 that only consistency-preserving inference systems will be considered.

– In Sect. 4.1, it is claimed that "a fair derivation can be constructed by exhaustively applying inferences to persisting formulas." However, this construction is circular: The notion of persisting formula (i.e., the formulas that belong to the limit) depends itself on the derivation.

– In the proof of Theorem 4.3, the case where $\gamma \in \mathcal{R}_{\mathcal{I}}(N_\infty \setminus \mathcal{R}_{\mathcal{F}}(N_\infty))$ is not covered.

– In Sect. 4.2, the phrase "side premises that are true in N" must be understood as meaning that the side premises both belong to N and are true in I_N.

– Lemma 4.5 states the basic properties of the redundant clause operator $\mathcal{R}_{\mathcal{F}}$ (monotonicity and independence). Lemma 4.6 states the corresponding properties of the redundant inference operator $\mathcal{R}_{\mathcal{I}}$. As justification for Lemma 4.6, the authors tell us that "the proof uses Lemma 4.5," but redundant inferences are a more general concept than redundant clauses, and we see no way to bridge the gap.

– Similarly, in the proof Theorem 4.9, the application of Lemma 4.5 does not fit. What is needed is a generalization of Lemma 4.6.

– In condition (ii) of Figure 4, Sect. 4.2, $A_{ii}\sigma$ should be changed to $A_{ij}\sigma$.

– In nth side premise of Figure 4, Sect. 4.2, A_{1n} should be changed to A_{n1}.

– Sect. 4.3 states "Subsumption defines a well-founded ordering on clauses." A simple counterexample is an infinite sequence repeating some clause. A correct statement would instead be "Proper subsumption defines a well-founded ordering on clauses."

– In Lemma 4.10 it is not clear which selection function is used. When the lemma is applied in the proofs of Lemma 4.11 and Theorem 4.13, it must be $S_{\mathcal{O}_\infty}$.

– In Lemma 4.10 $G(\mathcal{S})$ and $G(\mathcal{S}')$ are related by $\rhd^*$, but $\rhd$ is needed in the proofs of Lemma 4.11 and Lemma 4.13 since then derivations in RP, which are possibly infinite, can be projected to theorem proving processes. However $G(\mathcal{S}) \rhd G(\mathcal{S}')$ does not hold in one of the cases since a combination of deduction and deletion is required. A solution is to change the definition of $\rhd$ to allow such combinations.

– In Lemma 4.10 it is not clear that the extension used should be the same between any considered pair of states. Otherwise, the lemma cannot be used to project derivations in RP to theorem proving processes.

– In Lemma 4.11 it is not clear which selection function is used. When the lemma is applied in the proofs of Theorem 4.13, it must be $S_{\mathcal{O}_\infty}$.

– A step in the proof of Lemma 4.11 considers a clause $D \in \mathcal{P}_l$ which has a nonredundant instance C. It is claimed that when D is removed from $\mathcal{P}$, another clause D' with C as instance appears in some $\mathcal{O}'_l$. That, however, does not follow if D was removed by backward subsumption. The problem can be resolved by choosing D as minimal, with respect to subsumption, among the clauses that generalize C in the derivation. This can be done since proper subsumption is well founded.

- In Lemma 4.11, a minor inconsistency is that the described first-order derivation is indexed from 1 instead of 0.

- In the proof of Theorem 4.13, the conclusion of Lemma 4.11 is stated as $N_\infty \setminus \mathcal{R}(N_\infty) \subseteq \mathcal{O}_\infty$, but it should have been $N_\infty \setminus \mathcal{R}(N_\infty) \subseteq G(\mathcal{O}_\infty)$. Furthermore, when lemma 4.11 was first stated the conclusion was $N_\infty \setminus \mathcal{R}_\mathcal{F}(N_\infty) \subseteq G(\mathcal{S}_\infty)$. The two are by fairness equivalent, but we find $N_\infty \setminus \mathcal{R}(N_\infty) \subseteq G(\mathcal{O}_\infty)$ more intuitive since it more clearly expresses that all nonredundant clauses grow old.

Chief among the factors that contribute to making the chapter hard to follow is that many lemmas are stated (and usually proved) but not referenced later. We already mentioned the unfortunate Lemma 3.7. Sect. 4 contains several other specimens:

- Theorem 4.3 (*fair_ derive_ saturated_ upto*) states a completeness theorem for fair derivations. However, in Sect. 4.3, fairness is defined differently, and neither the text nor the formalization applies this theorem.

- For the same reason, the property stated in the next-to-last sentence of Sect. 4.1 (*standard_ redundancy_ criterion_ extension_fair_ iff*), which lifts fairness with respect to $(\mathcal{R}_\mathcal{F}, \mathcal{R}_\mathcal{I})$ to a standard extension $(\mathcal{R}_\mathcal{F}, \mathcal{R}'_\mathcal{I})$, is not needed later.

- Lemma 4.2 (*sat_ deriv_ Liminf_ iff*, *Ri_ Sup_ subset_ Ri_ Liminf*) is not referenced in the text, but we need it to prove Theorem 4.13 (*fair_ state_ seq_ complete*).

- Lemma 4.2 (*Rf_ Sup_ subset_ Rf_ Liminf*) is not referenced in the text, but we need it to prove Lemma 4.11 (*fair_ imp_ Liminf_ minus_ Rf_ subset_ ground_ Liminf_ state*).

- Lemma 4.6 (*saturated_ upto_ complete_ if*) is not referenced in the text, but we need it to prove Lemma 4.10 (*resolution_prover_ground_ derivation*), Lemma 4.11 (*fair_ imp_ Liminf_ minus_ Rf_ subset_ ground_ Liminf_ state*), and Theorem 4.13 (*fair_ state_ seq_ complete*).

- Theorem 4.8 (*Ri_ effective*) is not referenced in the text, but we need it to prove Theorem 4.13 (*fair_ state_ seq_ complete*).

- Theorem 4.9 (*saturated_ upto_ complete*) is invoked implicitly in the next-to-last sentence in the proof of Theorem 4.13 (*fair_ state_ seq_ complete*).

- The sentence "In that case, if the derivation is fair with respect to inferences in Γ the derivation is also fair with respect to inferences in Γ', and vice versa" on page 38 of Section 4.1 (*redundancy_ criterion_ standard_ extension_ saturated_ upto_ iff*) is not referenced in the text, but we need it to prove Theorem 4.13 (*fair_ state_ seq_ complete*).

References

[1] L. Bachmair, N. Dershowitz, and D. A. Plaisted. Completion without failure. In H. Aït-Kaci and M. Nivat, editors, *Rewriting Techniques—Resolution of Equations in Algebraic Structures*, volume 2, pages 1–30. Academic Press, 1989.

[2] L. Bachmair and H. Ganzinger. Rewrite-based equational theorem proving with selection and simplification. *Journal of Logic and Computation*, 4(3):217–247, 1994.

[3] L. Bachmair and H. Ganzinger. Resolution theorem proving. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, pages 19–99. Elsevier and MIT Press, 2001.

[4] C. Ballarin. Locales: A module system for mathematical theories. *Journal of Automated Reasoning*, 52(2):123–153, 2014.

[5] J. Biendarra, J. C. Blanchette, A. Bouzy, M. Desharnais, M. Fleury, J. Hölzl, O. Kuncar, A. Lochbihler, F. Meier, L. Panny, A. Popescu, C. Sternagel, R. Thiemann, and D. Traytel. Foundational (co)datatypes and (co)recursion for higher-order logic. In C. Dixon and M. Finger, editors, *FroCoS 2017*, volume 10483 of *LNCS*, pages 3–21. Springer, 2017.

[6] J. C. Blanchette, M. Fleury, P. Lammich, and C. Weidenbach. A verified SAT solver framework with learn, forget, restart, and incrementality. *Journal of Automated Reasoning*, 61(3):333–366, 2018.

[7] J. C. Blanchette, M. Fleury, and D. Traytel. Nested multisets, hereditary multisets, and syntactic ordinals in Isabelle/HOL. In D. Miller, editor, *FSCD 2017*, volume 84 of *LIPIcs*, pages 11:1–11:18. Schloss Dagstuhl—Leibniz-Zentrum für Informatik, 2017.

[8] J. C. Blanchette, C. Kaliszyk, L. C. Paulson, and J. Urban. Hammering towards QED. *Journal of Formalized Reasoning*, 9(1):101–148, 2016.

[9] J. C. Blanchette, A. Popescu, and D. Traytel. Soundness and completeness proofs by coinductive methods. *Journal of Automated Reasoning*, 58(1):149–179, 2017.

[10] D. Brand. Proving theorems with the modification method. *SIAM Journal on Computing*, 4(4):412–430, 1975.

[11] S. Cruanes. Logtk: A logic toolkit for automated reasoning and its implementation. In S. Schulz, L. de Moura, and B. Konev, editors, *PAAR-2014*, volume 31 of *EPiC Series in Computing*, pages 39–49. EasyChair, 2014.

[12] J. Denzinger, M. Kronenburg, and S. Schulz. DISCOUNT—a distributed and learning equational prover. *Journal of Automated Reasoning*, 18(2):189–198, 1997.

[13] M. Fleury, J. C. Blanchette, and P. Lammich. A verified SAT solver with watched literals using Imperative HOL. In J. Andronick and A. P. Felty, editors, *CPP 2018*, pages 158–171. ACM, 2018.

[14] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[15] N. Hirokawa, A. Middeldorp, C. Sternagel, and S. Winkler. Infinite runs in abstract completion. In D. Miller, editor, *FSCD 2017*, volume 84 of *LIPIcs*, pages 19:1–19:16. Schloss Dagstuhl—Leibniz-Zentrum für Informatik, 2017.

[16] A. Krauss. Partial recursive functions in higher-order logic. In U. Furbach and N. Shankar, editors, *IJCAR 2006*, volume 4130 of *LNCS*, pages 589–603. Springer, 2006.

[17] W. McCune. Otter 2.0. In M. E. Stickel, editor, *CADE-10*, volume 449 of *LNCS*, pages 663–664. Springer, 1990.

[18] R. Nieuwenhuis and A. Rubio. Paramodulation-based theorem proving. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, pages 371–443. Elsevier and MIT Press, 2001.

[19] T. Nipkow. Teaching semantics with a proof assistant: No more LSD trip proofs. In V. Kuncak and A. Rybalchenko, editors, *VMCAI 2012*, volume 7148 of *LNCS*, pages 24–38. Springer, 2012.

[20] T. Nipkow and G. Klein. *Concrete Semantics: With Isabelle/HOL*. Springer, 2014.

[21] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[22] R. O'Connor. Essential incompleteness of arithmetic verified by Coq. In J. Hurd and T. F. Melham, editors, *TPHOLs 2005*, volume 3603 of *LNCS*, pages 245–260. Springer, 2005.

[23] L. C. Paulson. A machine-assisted proof of Gödel's incompleteness theorems for the theory of hereditarily finite sets. *Review of Symbolic Logic*, 7(3):484–498, 2014.

[24] N. Peltier. A variant of the superposition calculus. *Archive of Formal Proofs*, 2016, 2016.

[25] H. Persson. Constructive completeness of intuitionistic predicate logic—a formalisation in type theory. Licentiate thesis, Chalmers tekniska högskola and Göteborgs universitet, 1996.

[26] B. C. Pierce. Lambda, the ultimate TA: Using a proof assistant to teach programming language foundations. In G. Hutton and A. P. Tolmach, editors, *ICFP 2009*, pages 121–122. ACM, 2009.

[27] G. Reger and M. Suda. Checkable proofs for first-order theorem proving. In G. Reger and D. Traytel, editors, *ARCADE 2017*, volume 51 of *EPiC Series in Computing*, pages 55–63. EasyChair, 2017.

[28] A. Schlichtkrull. Formalization of the resolution calculus for first-order logic. *Journal of Automated Reasoning*, 61(4):455–484, 2018.

[29] N. Shankar. Towards mechanical metamathematics. *Journal of Automated Reasoning*, 1(4):407–434, 1985.

[30] N. Shankar. *Metamathematics, Machines, and Gödel's Proof*, volume 38 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1994.

[31] G. Sutcliffe, J. Zimmer, and S. Schulz. TSTP data-exchange formats for automated theorem proving tools. In W. Zhang and V. Sorge, editors, *Distributed Constraint Problem Solving and Reasoning in Multi-Agent Systems*, volume 112 of *Frontiers in Artificial Intelligence and Applications*, pages 201–215. IOS Press, 2004.

[32] R. Thiemann and C. Sternagel. Certification of termination proofs using CeTA. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *TPHOLs 2009*, volume 5674 of *LNCS*, pages 452–468. Springer, 2009.

[33] A. Voronkov. AVATAR: The architecture for first-order theorem provers. In A. Biere and R. Bloem, editors, *CAV 2014*, volume 8559 of *LNCS*, pages 696–710. Springer, 2014.

[34] D. Wand. Polymorphic+typeclass superposition. In S. Schulz, L. de Moura, and B. Konev, editors, *PAAR-2014*, volume 31 of *EPiC Series in Computing*, pages 105–119. EasyChair, 2014.

[35] C. Weidenbach. Combining superposition, sorts and splitting. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume II, pages 1965–2013. Elsevier and MIT Press, 2001.

[36] M. Wenzel. Isabelle/Isar—a generic framework for human-readable proof documents. In R. Matuszewski and A. Zalewska, editors, *From Insight to Proof: Festschrift in Honour of Andrzej Trybulec*, volume 10(23) of *Studies in Logic, Grammar, and Rhetoric*. University of Białystok, 2007.

[37] M. Wenzel. Isabelle/jEdit—a prover IDE within the PIDE framework. In J. Jeuring, J. A. Campbell, J. Carette, G. D. Reis, P. Sojka, M. Wenzel, and V. Sorge, editors, *CICM 2012*, volume 7362 of *LNCS*, pages 468–471. Springer, 2012.

[38] H. Zhang and D. Kapur. First-order theorem proving using conditional rewrite rules. In E. L. Lusk and R. A. Overbeek, editors, *CADE-9*, volume 310 of *LNCS*, pages 1–20. Springer, 1988.

A Verified Automatic Prover Based on Ordered Resolution

Anders Schlichtkrull[1], Jasmin Christian Blanchette[2], and Dmitriy Traytel[3]

[1] *DTU Compute, Technical University of Denmark, Kongens Lyngby, Denmark*
[2] *Department of Computer Science, Vrije Universiteit Amsterdam, Amsterdam, The Netherlands*
[2] *Max-Planck-Institut für Informatik, Saarland Informatics Campus, Saarbrücken, Germany*
[3] *Institute of Information Security, Department of Computer Science, ETH Zürich, Zürich, Switzerland*

Abstract

First-order theorem provers based on superposition, such as E, SPASS, and Vampire, play an important role in formal software verification. They are based on sophisticated logical calculi that combine ordered resolution and equality reasoning. They also employ advanced algorithms, data structures, and heuristics. As a step towards verifying the correctness of state-of-the-art provers, we specify, using the Isabelle/HOL proof assistant, a purely functional ordered resolution prover and formally establish its soundness and refutational completeness. Methodologically, we apply stepwise refinement to obtain, from an abstract specification of a nondeterministic prover, a verified deterministic program, written in a subset of Isabelle/HOL from which we extract purely functional Standard ML code that constitutes a semidecision procedure for first-order logic.

CSS Concepts: Theory of computation / Logic and verification; Theory of computation / Automated reasoning; Software and its engineering / Completeness

Additional Key Words and Phrases: automatic theorem provers, proof assistants, first-order logic, refinement

1 Introduction

Formal verification of programs aims at ensuring correctness by mechanically checking their behavior with respect to a logical specification. Automatic theorem provers based on superposition, such as E [Schulz, 2013b], SPASS [Weidenbach et al., 2009], and Vampire [Kovács and Voronkov, 2013], are often employed as backends in verification tools. They are used to discharge verification conditions [Bobot et al., 2011; Paulson and Blanchette, 2012], but also to generate loop invariants [Kovács and Voronkov, 2009]. Superposition is a highly successful logical calculus for first-order logic with equality, which generalizes both ordered resolution [Bachmair and Ganzinger, 2001] and ordered (unfailing) completion [Bachmair et al., 1989].

Resolution does not operate on first-order formulas but instead on sets of clauses. A clause is an n-ary disjunction of literals $L_1 \vee \cdots \vee L_n$ whose free variables are interpreted universally. Each literal is either an atom A or its negation $\neg A$. An atom is a symbol applied to a tuple of terms—e.g., $\mathsf{divides}(2, n)$. The empty clause, which is false, is denoted by $\bot$. Resolution works by refutation: Conceptually, the calculus proves a conjecture $\forall \bar{x}.\, C$ from axioms A by deriving $\bot$ from $A \cup \{\exists \bar{x}.\, \neg C\}$, indicating its unsatisfiability.

Compared with plain resolution, *ordered resolution* relies on an order on the atoms to restrict the search space. Another important difference is that it uses a redundancy criterion to discard subsumed clauses at any point; for example, $\mathsf{p}(x) \lor \mathsf{q}(x)$ and $\mathsf{p}(\mathsf{5})$ are subsumed by $\mathsf{p}(x)$.

Using formal verification, we aim to develop trustworthy programs. But why should anyone trust verification tools? In particular, modern superposition provers are highly optimized programs that rely on sophisticated calculi, with a rich metatheory, and specialized data structures. In this paper, we propose an answer by verifying, in Isabelle/HOL [Nipkow et al., 2002], a purely functional prover based on ordered resolution. The verification relies on stepwise refinement [Wirth, 1971]. Four layers are connected by three refinement steps:

- Our starting point, layer 1 (Section 4), is an abstract Prolog-style nondeterministic resolution prover in a highly general form, as presented by Bachmair and Ganzinger [2001] and as formalized by Schlichtkrull et al. [2018a,b]. It operates on possibly infinite sets of clauses. Its soundness and refutational completeness are inherited by the other layers.
- Layer 2 (Section 5) operates on finite multisets of clauses and introduces a priority queue to ensure that logical inferences are performed in a fair manner, guaranteeing completeness: Given a valid conjecture, the prover will eventually find a proof.
- Layer 3 (Section 6) is a deterministic program that works on finite lists, committing to a concrete strategy for assigning priorities to clauses. However, it is not fully executable: It abstracts over operations on atoms and employs logical specifications instead of executable functions for some auxiliary notions.
- Finally, layer 4 (Section 7) is a fully executable program. It provides a concrete datatype for atoms and executable definitions for all auxiliary notions, including unifiers, clause subsumption, and the order on atoms.

From layer 4, we can extract Standard ML code by invoking Isabelle's code generator [Haftmann and Nipkow, 2010]. The resulting prover serves first and foremost as a proof of concept: It uses an efficient calculus (layer 1) and a reasonable strategy to ensure fairness (layers 2 and 3), but it depends on naive list-based data structures. Further refinement steps will be required to obtain a prover that is competitive with the state of the art.

The refinement steps connect vastly different levels of abstraction, spanning much of computer science. The most abstract level is occupied by an infinitary logical calculus and the semantics of first-order logic. Soundness and completeness relate these two notions. At the functional programming level, soundness amounts to a safety property: Whenever the program terminates normally, its outcome is correct, whether it is a proof or a finite *saturation* witnessing unprovability. Correspondingly, refutational completeness is a liveness property: The program will always terminate normally with a proof if the conjecture is valid. Our executable functional prover demonstrates that, far from being academic exercises, Bachmair and Ganzinger's [2001] framework and its formalization by Schlichtkrull et al. [2018a,b] accurately capture the metatheory of actual provers.

To our knowledge, our program is the first verified prover for first-order logic implementing an optimized calculus. It is also the first example of the application of refinement in this context. This methodology has been used to verify SAT solvers [Blanchette et al., 2018; Marić, 2010], which decide the satisfiability of propositional formulas, but first-order logic is semidecidable—sound and complete provers are guaranteed to terminate only for unsatisfiable (i.e., provable) clause sets. This complicates the transfer of completeness results across refinement layers.

Our contributions are as follows:

- We unveil a verified sound and complete first-order prover based on ordered resolution.

- We propose a general methodology, using modern tools, for refining an abstract Prolog-style definition of a refutational prover to an ML-style functional program, applicable to provers and other nondeterministic semidecision procedures that can be stated abstractly.
- We present a reusable library of Isabelle/HOL definitions, lemmas, and proofs that supports the methodology. These concern atoms, terms, substitutions, and derivation chains.

In addition, we offer a few "proof pearls"—smaller proving puzzles that illustrate specific techniques and that we find instructive or elegant.

Our work is connected to the IsaFoL (Isabelle Formalization of Logic) project,[1] which aims at developing a library of results about logic and automated reasoning. The Isabelle source files are available in the IsaFoL repository[2] and in the *Archive of Formal Proofs*.[3] The parts specific to the functional prover refinement amount to about 4000 lines of source text. A convenient way to study the files is to open them in the Isabelle/jEdit [Wenzel, 2012] development environment, as explained in the repository's readme file. This will ensure that logical symbols are rendered properly and will let you inspect proof states. The files were created using Isabelle version 2017, but the repositories will be updated to track Isabelle's evolution.

2 Isabelle/HOL

Isabelle [Nipkow and Klein, 2014; Nipkow et al., 2002] is a generic proof assistant that supports multiple object logics. Its most developed instantiation, Isabelle/HOL, provides a version of classical higher-order logic (HOL) [Church, 1940] that supports rank-1 (top-level) polymorphism, Haskell-style type classes, and Hilbert's choice operator. Unlike the type theories that underlie Agda [Bove et al., 2009] and Coq [Bertot and Castéran, 2004], HOL has no built-in notion of computation or executability. Nonetheless, a substantial fragment of HOL corresponds closely to Standard ML or Haskell and can be exported to these languages using a code generator [Haftmann and Nipkow, 2010].

Isabelle's syntax is inspired by both ML and traditional mathematical conventions. The types are built from type variables $'a, 'b, \ldots$ and n-ary type constructors, normally written in postfix notation (e.g., $'a\ \mathit{list}$). The infix type constructor $'a \Rightarrow 'b$ is interpreted as the (total) function space from $'a$ to $'b$. Propositions are terms of type *bool*, a datatype equipped with the constructors False and True. The familiar logical symbols $\forall$, $\exists$, $\neg$, $\wedge$, $\vee$, $\Longrightarrow$, $\Longleftrightarrow$, and $=$ are normal functions, although the quantifiers and equality on functions fall outside the executable fragment.

Isabelle adheres to a tradition initiated by the LCF system [Gordon et al., 1979]: All logical inferences are derived by a small trusted kernel, and types and functions are defined rather than axiomatized to guard against inconsistencies. Isabelle/HOL provides high-level specification mechanisms inspired by typed functional programming (e.g., ML) and logic programming (e.g., Prolog). These let us define large classes of types and operations, such as inductive datatypes, recursive functions, inductive predicates, and their coinductive counterparts. For example, the **codatatype** and **corec** commands [Biendarra et al., 2017] can be used to define codatatype and productive corecursive functions in the style of Haskell, and the **coinductive** command can be used to introduce coinductive predicates. Internally, Isabelle synthesizes suitable low-level nonrecursive definitions and derives the user specifications via primitive inferences. This *foundational approach* allows the system to provide a highly expressive, trustworthy specification language.

[1]`https://bitbucket.org/isafol/isafol/wiki/Home`
[2]`https://bitbucket.org/isafol/isafol/src/master/Functional_Ordered_Resolution_Prover/`
[3]`https://isa-afp.org/entries/Ordered_Resolution_Prover.html`

Isabelle proofs are expressed in a language called Isar [Wenzel, 2007]. It encourages a declarative, hierarchical style reminiscent of the format suggested by Lamport [1995], but with alphanumeric labels to identify intermediate proof steps. Isar also supports low-level *tactics* that manipulate the proof state directly, similar to those offered by Coq and other systems [Milner, 1984].

Most Isabelle formalizations are structured using *locales* [Ballarin, 2014]. A locale is a parameterized module, similar to an ML functor. The parameters may be types or terms satisfying some assumptions. For example, Isabelle/HOL provides the following basic specifications:

locale $\mathit{semigroup} =$
  fixes $* :: {}'a \Rightarrow {}'a \Rightarrow {}'a$
  assumes $(a * b) * c = a * (b * c)$

locale $\mathit{monoid} = \mathit{semigroup} +$
  fixes $1 :: {}'a$
  assumes $1 * a = a$ **and** $a * 1 = a$

The *semigroup* locale is parameterized by a type $'a$ and a binary operation $*$ on $'a$, which must be associative. The *monoid* locale inherits these parameters and assumptions and enriches them with a constant 1 assumed to be left- and right-neutral. Once a locale is declared, we can enter its scope at any point in a formal development. Within a locale's scope, we can use its parameters and assumptions in definitions, lemma statements, and proofs.

To actually use a locale, we must instantiate the parameters with concrete types and terms. For example, we can instantiate *monoid* by taking $('a, *, 1)$ to be $(\mathit{nat}, +, 0)$ or $(\mathit{nat}, \times, 1)$, where *nat* is the type of natural numbers and $0, 1, +, \times$ have their usual semantics. Before we can retrieve the definitions and lemmas from a locale, we must discharge the assumptions (e.g., $0 + a = a$ for all $a :: \mathit{nat}$). If a locale is parameterized by exactly one type variable, it can be introduced as a *type class* instead. This can be useful to offload some bureaucracy onto the type system, but it has its limitations: As in Haskell, a type class can be instantiated with a given type at most once.

3 Atoms and Substitutions

The first three refinement layers are based on an abstract library of first-order atoms and substitutions. In the fourth and final layer, the abstract framework is instantiated with concrete datatypes and functions. We start from the library of clausal logic developed by Blanchette et al. [2018], which is parameterized by a type $'a$ of logical atoms. Literals are defined as an inductive datatype with constructors for positive and negative literals:

datatype $'a\ \mathit{literal} =$
  $\mathsf{Pos}\ 'a$
  $|\ \mathsf{Neg}\ 'a$

The type of clauses is then defined as the abbreviation $'a\ \mathit{clause} = {}'a\ \mathit{literal}\ \mathit{multiset}$, where *multiset* is the type constructor of finite multisets. Thus, the clause $\neg A \vee B$, where A and B are arbitrary atoms, is represented by the multiset $\{\mathsf{Neg}\, A, \mathsf{Pos}\, B\}$, and the empty clause $\bot$ is represented by $\{\}$. The complement operation is defined as $-\,\mathsf{Neg}\, A = \mathsf{Pos}\, A$ and $-\,\mathsf{Pos}\, A = \mathsf{Neg}\, A$ for any atom A.

In automated reasoning, it is customary to view clauses as multisets of literals rather than as sets. One reason is that multisets behave more naturally under substitution. For example, applying the substitution $\{y \mapsto x\}$ to the two-literal clause $\mathsf{p}(x) \vee \mathsf{p}(y)$ results in a two-literal clause $\mathsf{p}(x) \vee \mathsf{p}(x)$, preserving the structure of the clause.

The truth value of ground (i.e., variable-free) atoms is given by a *Herbrand interpretation*: a set, of type $'a\ \mathit{set}$, of all true ground atoms. The "models" predicate $\models$ is defined as $I \models A \iff$

$A \in I$. This definition is lifted to literals, clauses, and sets of clauses in the usual way:

$$I \models \mathsf{Pos}\ A \iff A \in I \qquad\qquad I \models C \iff \exists L \in C.\ I \models L$$
$$I \models \mathsf{Neg}\ A \iff A \notin I \qquad\qquad I \models \mathcal{D} \iff \forall C \in \mathcal{D}.\ I \models C$$

A set of clauses $\mathcal{D}$ is *satisfiable* if there exists an interpretation I such that $I \models \mathcal{D}$.

Ordered resolution crucially depends on a notion of substitution and of most general unifier (MGU). These auxiliary concepts are provided by a third-party library, IsaFoR (Isabelle Formalization of Rewriting) [Thiemann and Sternagel, 2009]. To reduce our dependency on external libraries, we hide them behind abstract locales parameterized by a type of atoms $'a$ and a type of substitutions $'s$. We will usually think of the atoms as being first-order terms, which can be either a variable or a symbol applied to a list of first-order terms. Another possibility would be to use applicative first-order terms, also called λ-free higher-order terms. A substitution is modeled as a function from variables to terms. Substitutions can be applied to first-order terms by mapping them onto the terms' variables.

We start by defining a locale *substitution_ops* that declares the basic operations on substitutions: application ($\cdot$), identity (id), and composition ($\circ$):

```
locale substitution_ops =
  fixes
    $\cdot$ :: $'a \Rightarrow {}'s \Rightarrow {}'a$ and
    $\mathsf{id}$ :: $'s$ and
    $\circ$ :: $'s \Rightarrow {}'s \Rightarrow {}'s$
```

Within the locale's scope, we introduce a number of derived concepts. Ground atoms are defined as atoms that are left unchanged by substitutions:

$$\mathsf{is\_ground}\ A \iff \forall \sigma.\ A = A \cdot \sigma$$

A ground substitution is a substitution whose application always results in ground atoms:

$$\mathsf{is\_ground}\ \sigma \iff \forall A.\ \mathsf{is\_ground}(A \cdot \sigma)$$

Nonstrict and strict generalization are defined as

$$\mathsf{generalizes}\ A\ B \iff \exists \sigma.\ A \cdot \sigma = B$$
$$\mathsf{strictly\_generalizes}\ A\ B \iff \mathsf{generalizes}\ A\ B \land \neg\ \mathsf{generalizes}\ B\ A$$

The operators on atoms are lifted to literals, clauses, and sets of clauses. The grounding of a clause is defined as

$$\mathsf{grounding\_of}\ C = \{C \cdot \sigma \mid \mathsf{is\_ground}\ \sigma\}$$

The operator is lifted to sets of clauses in the obvious way. Clause subsumption is defined as

$$\mathsf{subsumes}\ C\ D \iff \exists \sigma.\ C \cdot \sigma \subseteq D$$
$$\mathsf{strictly\_subsumes}\ C\ D \iff \mathsf{subsumes}\ C\ D \land \neg\ \mathsf{subsumes}\ D\ C$$

Unifiers and MGUs are characterized as follows, where $\mathcal{A} :: {}'a\ set$ represents a unification constraint $A_1 \overset{?}{=} \cdots \overset{?}{=} A_k$ and $\mathcal{S} :: {}'a\ set\ set$ represents a set of unification constraints:

$$\mathsf{is\_unifier}\ \sigma\ \mathcal{A} \iff |\mathcal{A} \cdot \sigma| \leq 1$$
$$\mathsf{is\_unifier}\ \sigma\ \mathcal{S} \iff \forall \mathcal{A} \in \mathcal{S}.\ \mathsf{is\_unifier}\ \sigma\ \mathcal{A}$$
$$\mathsf{is\_mgu}\ \sigma\ \mathcal{S} \iff \mathsf{is\_unifier}\ \sigma\ \mathcal{S} \land (\forall \tau.\ \mathsf{is\_unifier}\ \tau\ \mathcal{S} \Longrightarrow \exists \gamma.\ \tau = \sigma \circ \gamma)$$

The next locale, *substitution*, characterizes the *substitution_ops* operations using assumptions. A separate locale is necessary because we cannot interleave assumptions and definitions in a single locale. In addition, *substitution* fixes a function for renaming clauses apart (so that they do not share any variables) and a function that, given a list of atoms, constructs an atom with these as subterms:

```
**locale** $\mathit{substitution} = \mathit{substitution\_ops}\ +$
  **fixes**
    $\mathsf{renamings\_apart} :: {'a\ \mathit{clause\ list}} \Rightarrow {'s\ \mathit{list}}$ **and**
    $\mathsf{atm\_of\_atms} :: {'a\ \mathit{list}} \Rightarrow {'a}$
  **assumes**
    $A \cdot \mathsf{id} = A$ **and**
    $A \cdot (\sigma \circ \tau) = (A \cdot \sigma) \cdot \tau$ **and**
    $(\forall A.\ A \cdot \sigma = A \cdot \tau) \Longrightarrow \sigma = \tau$ **and**
    $\mathsf{is\_ground\_cls}\ (C \cdot \sigma) \Longrightarrow \exists \tau.\ \mathsf{is\_ground}\ \tau \wedge C \cdot \tau = C \cdot \sigma$ **and**
    $\mathsf{wf\ strictly\_generalizes}$ **and**
    $|\mathsf{renamings\_apart}\ Cs| = |Cs|$ **and**
    $\rho \in \mathsf{renamings\_apart}\ Cs \Longrightarrow \mathsf{is\_renaming}\ \rho$ **and**
    $\mathsf{var\_disjoint}\ (Cs \cdot \mathsf{renamings\_apart}\ Cs)$ **and**
    $\mathsf{atm\_of\_atms}\ As \cdot \sigma = \mathsf{atm\_of\_atms}\ Bs \Longleftrightarrow \mathsf{map}\ (\lambda A.\ A \cdot \sigma)\ As = Bs$
```

The above definition is presented to give a flavor of our development. We refer to the Isabelle theory files for the precise definitions. Inside the locale, we prove further properties of the *substitution_ops* operations. Notably, we prove well-foundedness of the strictly_subsumes predicate based on the well-foundedness of strictly_generalizes, which is stated as an assumption. The atm_of_atms operation is needed for encoding a clause into a single atom in this well-foundedness proof.

Finally, a third locale, *mgu*, extends *substitution* by fixing a function mgu :: $'a\ \mathit{set\ set} \Rightarrow {'s\ \mathit{option}}$ that computes an MGU σ given a set of unification constraints. If a unifier exists, it returns Some σ; otherwise, it returns None.

4 Bachmair and Ganzinger's Prover

The formalization by Schlichtkrull et al. [2018a,b] of a nondeterministic ordered resolution prover presented by Bachmair and Ganzinger [2001] forms layer 1 of our refinement. Resolution is first defined on ground terms and proved sound and complete with respect to a propositional semantics. First-order ordered resolution is then defined and proved sound, and the ground completeness result is lifted to obtain completeness of the first-order resolution prover. The resolution inference rule is n-ary, with an optional "selection" mechanism to guide the proof search. In this paper, we disable selection and hence only need to consider the binary case, which can be implemented efficiently and forms the basis of modern provers such as E, SPASS, and Vampire.

The ordered resolution calculus is parameterized by a total order $>$ ("larger than") on atoms. The ground version of the calculus consists of the single inference rule

$$\frac{C \vee A \vee \cdots \vee A \quad \neg A \vee D}{C \vee D}$$

where atom A must be larger than all the atoms in clause C and larger than or equal to all the atoms in clause D. The side condition is not necessary for soundness, but it rules out many unnecessary inferences, thereby pruning the search space. Because clauses are defined as multisets, the order of the literals in a clause is immaterial; $\neg A \vee B$ and $B \vee \neg A$ are the same clause.

For first-order logic, the order on atoms $>$ is extended to an order $\succ$ on nonground atoms so that $B \succ A$ if and only if for all ground substitutions σ, we have $B \cdot \sigma > A \cdot \sigma$. The nonground version of the calculus consists of the single inference rule

$$\frac{C \vee A_1 \vee \cdots \vee A_k \quad \neg A \vee D}{(C \vee D) \cdot \sigma}$$

where σ is the (canonical) MGU that solves the unification problem $A_1 \overset{?}{=} \cdots \overset{?}{=} A_k \overset{?}{=} A$, each $A_i \cdot \sigma$ is strictly $\succ$-maximal with respect to the atoms in $C \cdot \sigma$, and $A \cdot \sigma$ is $\succ$-maximal with respect to the atoms in $D \cdot \sigma$. An important detail is that to achieve completeness, the rule must be adapted slightly to rename apart the variables occurring in different premises.

Resolution works by saturation. A set of clauses $\mathcal{D}$ is *saturated* if any conclusion from premises in $\mathcal{D}$ is already in $\mathcal{D}$. The ordered resolution calculus is refutationally complete, meaning that any unsatisfiable saturated set of clauses necessarily contains $\bot$.

Resolution provers exploit the calculus's completeness in the following way. They start with a finite set of initial clauses—the input problem—and successively add conclusions from premises in the set. If the inference rule is applied in a fair fashion on the available clauses, the set reaches saturation at the limit; if the set is unsatisfiable, this means $\bot$ is eventually derived, after finitely many steps. Crucially, not only do efficient provers add clauses to their working set, they also remove clauses that are deemed redundant. This requires a refined notion of saturation. We call a set of clauses $\mathcal{D}$ *saturated up to redundancy*, formally $\mathsf{saturated\_upto}\ \mathcal{D}$, if any inference from nonredundant clauses in $\mathcal{D}$ yields a redundant conclusion.

Bachmair and Ganzinger's nondeterministic first-order prover, called RP, captures the "dynamic" aspects of saturation. It builds on the first-order ordered resolution rule and a redundancy criterion. The redundant clauses are those that are tautological (i.e., clauses of the form $C \vee A \vee \neg A$) and those that are subsumed by another clause in the working set—for example, the clauses B and $A \vee B$ are both subsumed if the working set already contains B. Furthermore, RP takes advantage of subsumption resolution, which can be expressed as an inference rule:

$$\frac{D \vee L \quad C \vee (D \vee -L) \cdot \sigma}{C \vee D \cdot \sigma}$$

The conclusion subsumes the second premise, which may therefore be deleted.

The RP prover is defined as an inductive predicate $\rightsquigarrow$ on states, where a state is a triple $\mathcal{S} = (\mathcal{N}, \mathcal{P}, \mathcal{O})$ of *new clauses* $\mathcal{N}$, *processed clauses* $\mathcal{P}$, and *old clauses* $\mathcal{O}$. Initially, $\mathcal{N}$ is the input problem (including the negated conjecture), and $\mathcal{P} \cup \mathcal{O}$ is empty. Clauses can be removed if they are tautological or subsumed or after subsumption resolution has been applied. When all clauses in $\mathcal{N}$ have been processed (either removed entirely or moved to $\mathcal{P}$), a clause from $\mathcal{P}$ can be chosen for *inference computation*: This clause is moved to $\mathcal{O}$, and all its conclusions with premises from the other old clauses are introduced to form the new $\mathcal{N}$.

In Isabelle, an inductive predicate is specified as a set of Horn-style introduction rules, as in Prolog, but with the conclusion on the right. RP is defined as follows:

inductive $\rightsquigarrow$ :: *'a state* $\Rightarrow$ *'a state* $\Rightarrow$ *bool* **where**
  $\mathsf{Neg}\ A \in C \wedge \mathsf{Pos}\ A \in C \Longrightarrow (\mathcal{N} \cup \{C\}, \mathcal{P}, \mathcal{O}) \rightsquigarrow_1 (\mathcal{N}, \mathcal{P}, \mathcal{O})$
| $D \in \mathcal{P} \cup \mathcal{O} \wedge \mathsf{subsumes}\ D\ C \Longrightarrow (\mathcal{N} \cup \{C\}, \mathcal{P}, \mathcal{O}) \rightsquigarrow_2 (\mathcal{N}, \mathcal{P}, \mathcal{O})$
| $D \in \mathcal{N} \wedge \mathsf{strictly\_subsumes}\ D\ C \Longrightarrow (\mathcal{N}, \mathcal{P} \cup \{C\}, \mathcal{O}) \rightsquigarrow_3 (\mathcal{N}, \mathcal{P}, \mathcal{O})$
| $D \in \mathcal{N} \wedge \mathsf{strictly\_subsumes}\ D\ C \Longrightarrow (\mathcal{N}, \mathcal{P}, \mathcal{O} \cup \{C\}) \rightsquigarrow_4 (\mathcal{N}, \mathcal{P}, \mathcal{O})$
| $D \in \mathcal{P} \cup \mathcal{O} \wedge \mathsf{reduces}\ D\ C\ L \Longrightarrow (\mathcal{N} \cup \{C \uplus \{L\}\}, \mathcal{P}, \mathcal{O}) \rightsquigarrow_5 (\mathcal{N} \cup \{C\}, \mathcal{P}, \mathcal{O})$

$$
\begin{aligned}
&|\ D \in \mathcal{N} \wedge \mathsf{reduces}\ D\ C\ L \Longrightarrow (\mathcal{N}, \mathcal{P} \cup \{C \uplus \{L\}\}, \mathcal{O}) \rightsquigarrow_6 (\mathcal{N}, \mathcal{P} \cup \{C\}, \mathcal{O})\\
&|\ D \in \mathcal{N} \wedge \mathsf{reduces}\ D\ C\ L \Longrightarrow (\mathcal{N}, \mathcal{P}, \mathcal{O} \cup \{C \uplus \{L\}\}) \rightsquigarrow_7 (\mathcal{N}, \mathcal{P} \cup \{C\}, \mathcal{O})\\
&|\ (\mathcal{N} \cup \{C\}, \mathcal{P}, \mathcal{O}) \rightsquigarrow_8 (\mathcal{N}, \mathcal{P} \cup \{C\}, \mathcal{O})\\
&|\ (\{\}, \mathcal{P} \cup \{C\}, \mathcal{O}) \rightsquigarrow_9 (\mathsf{concl\_of}\ \text{`}\ \mathsf{infers\_between}\ \mathcal{O}\ C, \mathcal{P}, \mathcal{O} \cup \{C\})
\end{aligned}
$$

Subscripts on $\rightsquigarrow$ identify the rules. The notation f ‘ X stands for the image of the set (or multiset) X under function f, $\mathsf{infers\_between}\ \mathcal{O}\ C$ calculates all the inferences whose premises are a subset of $\mathcal{O} \cup \{C\}$ that contains C, and $\mathsf{reduces}\ D\ C\ L \iff \exists D'\ L'\ \sigma.\ D = D' \uplus \{L'\} \wedge -L = L' \cdot \sigma \wedge D' \cdot \sigma \subseteq C$.

Example 4.1. There are many ways to derive $\bot$ from the unsatisfiable clause set $\{\mathsf{p}(x), \neg\mathsf{p}(\mathsf{a}) \vee \neg\mathsf{p}(\mathsf{b})\}$. The derivation on the left-hand side below relies on the two mandatory rules (rules 8 and 9). On the right-hand side, we show a shorter derivation that exploits reduction and subsumption to avoid performing resolution inferences. In both cases, we assume a reasonable term order.

$$
\begin{aligned}
&(\{\mathsf{p}(x), \neg\mathsf{p}(\mathsf{a}) \vee \neg\mathsf{p}(\mathsf{b})\}, \{\}, \{\})\\
\rightsquigarrow_8\ &(\{\neg\mathsf{p}(\mathsf{a}) \vee \neg\mathsf{p}(\mathsf{b})\}, \{\mathsf{p}(x)\}, \{\})\\
\rightsquigarrow_8\ &(\{\}, \{\mathsf{p}(x), \neg\mathsf{p}(\mathsf{a}) \vee \neg\mathsf{p}(\mathsf{b})\}, \{\})\\
\rightsquigarrow_9\ &(\{\}, \{\neg\mathsf{p}(\mathsf{a}) \vee \neg\mathsf{p}(\mathsf{b})\}, \{\mathsf{p}(x)\})\\
\rightsquigarrow_9\ &(\{\neg\mathsf{p}(\mathsf{a})\}, \{\}, \{\mathsf{p}(x), \neg\mathsf{p}(\mathsf{a}) \vee \neg\mathsf{p}(\mathsf{b})\})\\
\rightsquigarrow_8\ &(\{\}, \{\neg\mathsf{p}(\mathsf{a})\}, \{\mathsf{p}(x), \neg\mathsf{p}(\mathsf{a}) \vee \neg\mathsf{p}(\mathsf{b})\})\\
\rightsquigarrow_9\ &(\{\bot\}, \{\}, \{\mathsf{p}(x), \neg\mathsf{p}(\mathsf{a}) \vee \neg\mathsf{p}(\mathsf{b}), \neg\mathsf{p}(\mathsf{a})\})\\
\rightsquigarrow_8\ &(\{\}, \{\bot\}, \{\mathsf{p}(x), \neg\mathsf{p}(\mathsf{a}) \vee \neg\mathsf{p}(\mathsf{b}), \neg\mathsf{p}(\mathsf{a})\})\\
\rightsquigarrow_9\ &(\{\}, \{\}, \{\mathsf{p}(x), \neg\mathsf{p}(\mathsf{a}) \vee \neg\mathsf{p}(\mathsf{b}), \neg\mathsf{p}(\mathsf{a}), \bot\})
\end{aligned}
\qquad
\begin{aligned}
&(\{\mathsf{p}(x), \neg\mathsf{p}(\mathsf{a}) \vee \neg\mathsf{p}(\mathsf{b})\}, \{\}, \{\})\\
\rightsquigarrow_8\ &(\{\neg\mathsf{p}(\mathsf{a}) \vee \neg\mathsf{p}(\mathsf{b})\}, \{\mathsf{p}(x)\}, \{\})\\
\rightsquigarrow_5\ &(\{\neg\mathsf{p}(\mathsf{b})\}, \{\mathsf{p}(x)\}, \{\})\\
\rightsquigarrow_5\ &(\{\bot\}, \{\mathsf{p}(x)\}, \{\})\\
\rightsquigarrow_3\ &(\{\bot\}, \{\}, \{\})\\
\rightsquigarrow_8\ &(\{\}, \{\bot\}, \{\})\\
\rightsquigarrow_9\ &(\{\}, \{\}, \{\bot\})
\end{aligned}
$$

Example 4.2. The next example shows that RP can diverge even on unsatisfiable clause sets:

$$
\begin{aligned}
&(\{\neg\mathsf{p}(\mathsf{a},\mathsf{a}), \mathsf{p}(x,x), \neg\mathsf{p}(\mathsf{f}(x),y) \vee \mathsf{p}(x,y)\}, \{\}, \{\})\\
\rightsquigarrow_8^+\ &(\{\}, \{\neg\mathsf{p}(\mathsf{a},\mathsf{a}), \mathsf{p}(x,x), \neg\mathsf{p}(\mathsf{f}(x),y) \vee \mathsf{p}(x,y)\}, \{\})\\
\rightsquigarrow_9\ &(\{\}, \{\neg\mathsf{p}(\mathsf{a},\mathsf{a}), \mathsf{p}(x,x)\}, \{\neg\mathsf{p}(\mathsf{f}(x),y) \vee \mathsf{p}(x,y)\})\\
\rightsquigarrow_9\ &(\{\mathsf{p}(x,\mathsf{f}(x))\}, \{\neg\mathsf{p}(\mathsf{a},\mathsf{a})\}, \{\neg\mathsf{p}(\mathsf{f}(x),y) \vee \mathsf{p}(x,y), \mathsf{p}(x,x)\})\\
\rightsquigarrow_8\ &(\{\}, \{\neg\mathsf{p}(\mathsf{a},\mathsf{a}), \mathsf{p}(x,\mathsf{f}(x))\}, \{\neg\mathsf{p}(\mathsf{f}(x),y) \vee \mathsf{p}(x,y), \mathsf{p}(x,x)\})\\
\rightsquigarrow_9\ &(\{\mathsf{p}(x,\mathsf{f}(\mathsf{f}(x)))\}, \{\neg\mathsf{p}(\mathsf{a},\mathsf{a})\}, \{\neg\mathsf{p}(\mathsf{f}(x),y) \vee \mathsf{p}(x,y), \mathsf{p}(x,x), \mathsf{p}(x,\mathsf{f}(x))\})\\
\rightsquigarrow_8\ &\cdots
\end{aligned}
$$

We can leave $\neg\mathsf{p}(\mathsf{a},\mathsf{a})$ in $\mathcal{P}$ forever and always generate more clauses of the form $\mathsf{p}(x, \mathsf{f}^i(x))$, for increasing values of i. This emphasizes the importance of employing a fair strategy for moving clauses from $\mathcal{P}$ to $\mathcal{O}$.

Formally, a *derivation* is a possibly infinite sequence of states $\mathcal{S}_0 \rightsquigarrow \mathcal{S}_1 \rightsquigarrow \mathcal{S}_2 \rightsquigarrow \cdots$. In Isabelle, this is expressed by the codatatype of lazy lists:

codatatype *'a llist* =
  LNil
| LCons *'a* (*'a llist*)

Lazy list operation names are prefixed by an L or l to distinguish them from the corresponding operations on finite lists. For example, lhd *xs* yields *xs*'s head (if $xs \neq \mathsf{LNil}$), and lnth *xs* *i* yields the $(i + 1)$st element of *xs* (if $i < |xs|$).

We capture the mathematical notation $\mathcal{S}_0 \rightsquigarrow \mathcal{S}_1 \rightsquigarrow \mathcal{S}_2 \rightsquigarrow \cdots$ formally as $\mathsf{chain}\ (\rightsquigarrow)\ \mathcal{S}s$, where $\mathcal{S}s$ is a lazy list of states and chain is a coinductive predicate:

coinductive $\mathsf{chain} :: ('a \Rightarrow 'a \Rightarrow \mathit{bool}) \Rightarrow 'a\ \mathit{llist} \Rightarrow \mathit{bool}$ **where**
$\quad\mathsf{chain}\ R\ (\mathsf{LCons}\ x\ \mathsf{LNil})$
$\mid \mathsf{chain}\ R\ xs \wedge R\ x\ (\mathsf{lhd}\ xs) \Longrightarrow \mathsf{chain}\ R\ (\mathsf{LCons}\ x\ xs)$

Coinduction is used to allow infinite chains. The base case is needed to allow finite chains. Chains cannot be empty.

Another important notion is that of the limit of a sequence Xs of sets. It is defined as the set of elements that are members of all positions of Xs except for an at most finite prefix:

definition $\mathsf{Liminf} :: 'a\ \mathit{set}\ \mathit{llist} \Rightarrow 'a\ \mathit{set}$ **where**
$\quad\mathsf{Liminf}\ Xs = \bigcup_{i<|Xs|} \bigcap_{j:i\le j<|Xs|} \mathsf{lnth}\ Xs\ j$

Liminf and other operators working on clause sets are lifted pointwise to states. For example, the limit of a sequence of states is defined as $\mathsf{Liminf}\ \mathcal{S}s = (\mathsf{Liminf}\ \mathcal{N}s, \mathsf{Liminf}\ \mathcal{P}s, \mathsf{Liminf}\ \mathcal{O}s)$, where $\mathcal{N}s$, $\mathcal{P}s$, and $\mathcal{O}s$ are the projections of the $\mathcal{N}$, $\mathcal{P}$, and $\mathcal{O}$ components of $\mathcal{S}s$. For the rest of this section, we assume that $\mathcal{S}s$ is a derivation.

The soundness theorem states that if RP derives $\bot$ (represented by the multiset $\{\}$) from a set of clauses, that set must be unsatisfiable:

theorem *RP_sound*:
$\quad\{\} \in \mathsf{Liminf}\ \mathcal{S}s \Longrightarrow \neg\ \mathsf{satisfiable}\ (\mathsf{grounding\_of}\ (\mathsf{lhd}\ \mathcal{S}s))$

A stronger, finer-grained notion of soundness relates models before and after a transition:

theorem *RP_model*:
$\quad\mathcal{S} \leadsto \mathcal{S}' \Longrightarrow (I \models \mathsf{grounding\_of}\ \mathcal{S}' \Longleftrightarrow I \models \mathsf{grounding\_of}\ \mathcal{S})$

The canonical way of expressing the unsatisfiability of a set or multiset of first-order clauses with respect to Herbrand interpretations is as the unsatisfiability of its grounding.

Completeness of the prover can only be guaranteed when its rules are executed in a fair order, such that clauses do not get stuck forever in $\mathcal{N}$ or $\mathcal{P}$. Accordingly, fairness is defined as $\mathsf{Liminf}\ \mathcal{N}s = \mathsf{Liminf}\ \mathcal{P}s = \{\}$. The completeness theorem states that the limit of a fair derivation is saturated:

theorem *RP_saturated_if_fair*:
$\quad\mathsf{fair}\ \mathcal{S}s \Longrightarrow \mathsf{saturated\_upto}\ (\mathsf{Liminf}\ (\mathsf{grounding\_of}\ \mathcal{S}s))$

In particular, if the initial problem is unsatisfiable, $\bot$ must appear in the $\mathcal{O}$ component of the limit of any fair derivation:

corollary *RP_complete_if_fair*:
$\quad\mathsf{fair}\ \mathcal{S}s \wedge \neg\ \mathsf{satisfiable}\ (\mathsf{grounding\_of}\ (\mathsf{lhd}\ \mathcal{S}s)) \Longrightarrow \{\} \in \mathcal{O}\mathsf{\_of}\ (\mathsf{Liminf}\ \mathcal{S}s)$

5 Ensuring Fairness

The second refinement layer is the prover $\mathsf{RP_w}$, which ensures fairness by assigning a *weight* to every clause and by organizing the set of processed clauses—the $\mathcal{P}$ component of a state—as a priority queue, where lighter clauses are chosen before heavier clauses. By assigning heavier weights to newer clauses, we can guarantee that all derivations are fair.

Another necessary ingredient for completeness is that derivations must be complete; for example, the incomplete derivation consisting of the single state $(\{C\}, \{\}, \{\})$ is not fair because C is never processed. This requirement is expressed formally as $\mathsf{full\_chain}\ (\leadsto_\mathsf{w})\ \mathcal{S}s$, where the full_chain predicate is defined coinductively as

coinductive $\mathsf{full\_chain} :: ('a \Rightarrow 'a \Rightarrow \mathit{bool}) \Rightarrow 'a\ \mathsf{llist} \Rightarrow \mathit{bool}$ **where**
$(\forall y.\ \neg\ R\ x\ y) \Longrightarrow \mathsf{full\_chain}\ R\ (\mathsf{LCons}\ x\ \mathsf{LNil})$
$|\ \mathsf{full\_chain}\ R\ \mathit{xs} \wedge R\ x\ (\mathsf{lhd}\ \mathit{xs}) \Longrightarrow \mathsf{full\_chain}\ R\ (\mathsf{LCons}\ x\ \mathit{xs})$

and characterized by the equivalence

lemma *full_chain_iff_chain*:
$\mathsf{full\_chain}\ R\ \mathit{xs} \Longleftrightarrow \mathsf{chain}\ R\ \mathit{xs} \wedge (\mathsf{lfinite}\ \mathit{xs} \Longrightarrow \forall y.\ \neg\ R\ (\mathsf{llast}\ \mathit{xs})\ y)$

For the rest of this section, we fix a full chain $\mathcal{S}s$ such that $\mathcal{P}\_\mathsf{of}\ (\mathsf{lhd}\ \mathcal{S}s) = \mathcal{O}\_\mathsf{of}\ (\mathsf{lhd}\ \mathcal{S}s) = \{\}$.

Because each $\mathsf{RP_w}$ rule corresponds to an RP rule, it is straightforward to lift the soundness and completeness results from RP to $\mathsf{RP_w}$. The main difficulty is to show that the priority queue ensures fairness of full derivations, which is needed to obtain an unconditional completeness theorem for $\mathsf{RP_w}$, without the assumption fair $\mathcal{S}s$.

5.1 Definition

The weight of a clause C, which defines its priority in the queue, may depend both on the clause itself and on when it was generated. As a result, the $\mathsf{RP_w}$ prover represents clauses by a pair (C, i), where i is the *timestamp*—the larger the timestamp, the newer the clause. A state is now a quadruple $\mathcal{S} = (\mathcal{N}, \mathcal{P}, \mathcal{O}, t)$, where the first three components are finite multisets and t is the timestamp to assign to the next generation of clauses. Formally, we have the following type abbreviations:

type_synonym $'a\ \mathit{wclause} = 'a\ \mathit{clause} \times \mathit{nat}$
type_synonym $'a\ \mathit{wstate} =$
$'a\ \mathit{wclause\ multiset} \times 'a\ \mathit{wclause\ multiset} \times 'a\ \mathit{wclause\ multiset} \times \mathit{nat}$

We extend the *FO_resolution_prover* locale, in which RP is defined, with a weight function that, for any given clause, is strictly monotone with respect to the timestamp, so that older copies of a clause are preferred to newer ones:

locale *weighted_FO_resolution_prover* = *FO_resolution_prover* +
fixes $\mathsf{weight} :: 'a\ \mathit{wclause} \Rightarrow \mathit{nat}$
assumes $i < j \Longrightarrow \mathsf{weight}\ (C, i) < \mathsf{weight}\ (C, j)$

The $\mathsf{RP_w}$ prover uses $'a\ \mathit{wclause}$ for clauses. It is defined inductively as follows:

inductive $\leadsto_{\mathsf{w}} :: 'a\ \mathit{wstate} \Rightarrow 'a\ \mathit{wstate} \Rightarrow \mathit{bool}$ **where**
$\mathsf{Neg}\ A \in C \wedge \mathsf{Pos}\ A \in C \Longrightarrow (\mathcal{N} \uplus \{(C,i)\}, \mathcal{P}, \mathcal{O}, t) \leadsto_{\mathsf{w1}} (\mathcal{N}, \mathcal{P}, \mathcal{O}, t)$
$|\ D \in \mathsf{fst}\ `\ (\mathcal{P} \uplus \mathcal{O}) \wedge \mathsf{subsumes}\ D\ C \Longrightarrow (\mathcal{N} + \{(C,i)\}, \mathcal{P}, \mathcal{O}, t) \leadsto_{\mathsf{w2}} (\mathcal{N}, \mathcal{P}, \mathcal{O}, t)$
$|\ D \in \mathsf{fst}\ `\ \mathcal{N} \wedge C \in \mathsf{fst}\ `\ \mathcal{P} \wedge \mathsf{strictly\_subsumes}\ D\ C \Longrightarrow$
$(\mathcal{N}, \mathcal{P}, \mathcal{O}, t) \leadsto_{\mathsf{w3}} (\mathcal{N}, \{(E,k) \in \mathcal{P}.\ E \neq C\}, \mathcal{O}, t)$
$|\ D \in \mathsf{fst}\ `\ \mathcal{N} \wedge \mathsf{strictly\_subsumes}\ D\ C \Longrightarrow (\mathcal{N}, \mathcal{P}, \mathcal{O} \uplus \{(C,i)\}, t) \leadsto_{\mathsf{w4}} (\mathcal{N}, \mathcal{P}, \mathcal{O}, t)$
$|\ D \in \mathsf{fst}\ `\ (\mathcal{P} \uplus \mathcal{O}) \wedge \mathsf{reduces}\ D\ C\ L \Longrightarrow$
$(\mathcal{N} \uplus \{(C \uplus \{L\}, i)\}, \mathcal{P}, \mathcal{O}, t) \leadsto_{\mathsf{w5}} (\mathcal{N} \uplus \{(C,i)\}, \mathcal{P}, \mathcal{O}, t)$
$|\ D \in \mathsf{fst}\ `\ \mathcal{N} \wedge \mathsf{reduces}\ D\ C\ L \wedge (\forall j.\ (C \uplus \{L\}, j) \in \mathcal{P} \Longrightarrow j \leq i) \Longrightarrow$
$(\mathcal{N}, \mathcal{P} \uplus \{(C \uplus \{L\}, i)\}, \mathcal{O}, t) \leadsto_{\mathsf{w6}} (\mathcal{N}, \mathcal{P} \uplus \{(C,i)\}, \mathcal{O}, t)$
$|\ D \in \mathsf{fst}\ `\ \mathcal{N} \wedge \mathsf{reduces}\ D\ C\ L \Longrightarrow (\mathcal{N}, \mathcal{P}, \mathcal{O} \uplus \{(C \uplus \{L\}, i)\}, t) \leadsto_{\mathsf{w7}} (\mathcal{N}, \mathcal{P} \uplus \{(C,i)\}, \mathcal{O}, t)$
$|\ (\mathcal{N} \uplus \{(C,i)\}, \mathcal{P}, \mathcal{O}, t) \leadsto_{\mathsf{w8}} (\mathcal{N}, \mathcal{P} \uplus \{(C,i)\}, \mathcal{O}, t)$
$|\ (\forall (D,j) \in \mathcal{P}.\ \mathsf{weight}\ (C,i) \leq \mathsf{weight}\ (D,j)) \wedge$
$\mathcal{N} = \mathsf{mset\_set}\ ((\lambda D.\ (D,t))\ `\ \mathsf{concl\_of}\ `\ \mathsf{infers\_between}\ (\mathsf{set\_mset}\ (\mathsf{fst}\ `\ \mathcal{O}))\ C) \Longrightarrow$
$(\{\}, \mathcal{P} \uplus \{(C,i)\}, \mathcal{O}, t) \leadsto_{\mathsf{w9}} (\mathcal{N}, \{(D,j) \in \mathcal{P}.\ D \neq C\}, \mathcal{O} \uplus \{(C,i)\}, t+1)$

where fst is the function that returns the first component of a pair, mset_set converts a set to the multiset with exactly one copy of each element in the set, and set_mset converts a multiset to the set of elements in the multiset. Each $\mathsf{RP_w}$ rule i corresponds to RP rule i.

$\mathsf{RP_w}$ uses finite multisets for representing $\mathcal{N}$, $\mathcal{P}$, and $\mathcal{O}$. They offer a compromise between the layer 1 representation as sets and the layer 3 implementation as lists. Finite multisets help eliminate some unfair derivations:

- The finiteness condition guarantees that each clause in $\mathcal{N}$ gets the opportunity to move to $\mathcal{P}$ (and further to $\mathcal{O}$).
- The set-based RP allows idle transitions, such as $(\mathcal{N} \cup \{C\}, \mathcal{P}, \mathcal{O}) \leadsto (\mathcal{N}, \mathcal{P} \cup \{C\}, \mathcal{O})$ where $C \in \mathcal{N} \cap \mathcal{P}$. The use of multisets and $\uplus$ precludes such steps in $\mathsf{RP_w}$.

In the inductive definition of $\mathsf{RP_w}$, the last rule, which computes inferences, assigns timestamp t to each newly computed clause D and increments t. Since we want $\mathcal{P}$ to work as a priority queue, we let the prover choose a clause C with the smallest weight.

Timestamps are preserved when clauses are moved between $\mathcal{N}$, $\mathcal{P}$, and $\mathcal{O}$. They are also preserved by reduction steps (rules 5 to 7), even though reduction alters the clauses by removing unnecessary literals. This works because reduction can only happen finitely many times—a k-literal clause can be reduced at most k times. Therefore, there is no danger of divergence due to an infinite chain of reductions. Incidentally, it would also be possible to assign the current t as the reduced clause's timestamp, but this would effectively penalize the clause, for no good reason. If anything, a reduced clause becomes more interesting, not less; after all, the most interesting clause by far is $\bot$.

Timestamps introduce a new danger. It may be the case that a clause C is in a limit (of a sequence of states or of a state component) if we project away the timestamps, but that no single timestamped clause (C, i) belongs to the limit, because the timestamps keep changing, as in the infinite sequence $\{(C,0)\}, \{(C,1)\}, \{(C,2)\}, \ldots$. This could in principle arise due to subsumption, leading to derivations such as

$$\begin{aligned} &(\_,\ \_ \uplus \{(C,0)\},\ \_) \leadsto \\ (\_,\ \_ \uplus \{(C,0),(C,1)\},\ \_) \leadsto{}& (\_,\ \_ \uplus \{(C,1)\},\ \_) \leadsto^+ \\ (\_,\ \_ \uplus \{(C,1),(C,2)\},\ \_) \leadsto{}& (\_,\ \_ \uplus \{(C,2)\},\ \_) \leadsto^+ \cdots \end{aligned}$$

To prevent this behavior, the $\mathsf{RP_w}$ rules are formulated so that whenever they remove the earliest copy of any clause $C \in \mathcal{P}$, they also remove all its copies from $\mathcal{P}$. This property is captured by the following lemma, which is proved by case distinction on the rules:

> **lemma** *preserve_min_P*:
> $\mathcal{S} \leadsto_{\mathsf{w}} \mathcal{S}' \wedge (C, i) \in \mathcal{P}\_\mathsf{of}\ \mathcal{S} \wedge (\forall k.\ (C, k) \in \mathcal{P}\_\mathsf{of}\ \mathcal{S} \Longrightarrow k \geq i) \wedge C \in \mathsf{fst}\ `\ \mathcal{P}\_\mathsf{of}\ \mathcal{S}' \Longrightarrow$
> $(C, i) \in \mathcal{P}\_\mathsf{of}\ \mathcal{S}'$

This completes our review of $\mathsf{RP_w}$. As an intermediate step towards a more concrete prover, we restrict the weight function to be a linear equation that considers both timestamps and clause sizes:

> **locale** *weighted_FO_resolution_prover_with_size_timestamp_factors* = *FO_resolution_prover* +
> **fixes**
> $|\ |$ :: *'a* ⇒ *nat* **and**
> size_factor :: *nat* **and**
> timestamp_factor :: *nat*
> **assumes**

```
    timestamp_factor $> 0$
**begin**
**fun** weight :: $'a$ *wclause* $\Rightarrow$ *nat* **where**
  weight $(C, i)$ = size_factor $*$ $|C|$ + timestamp_factor $*$ $i$
**end**
```

where $|C| = \sum_{A\,:\,A \in C \vee \neg A \in C} |A|$. It is easy to prove that this definition of weight is strictly monotone and hence that this locale is a sublocale of *weighted_FO_resolution_prover*. This gives us a correspondingly specialized version of $\mathsf{RP_w}$ that will form the basis of further refinement steps.

The idea of organizing $\mathcal{P}$ as a priority queue is well known in the automated reasoning community. It is mentioned in a footnote in Bachmair and Ganzinger [2001, p. 44], but they require their weight function to be monotone not only in the timestamp but also in the clause size, claiming that this is necessary to ensure fairness. Although it often makes sense to prefer small clauses to large ones, our proof reveals that clause size is irrelevant for fairness, even in the presence of reductions. This demonstrates how working out the details and making all assumptions explicit using a proof assistant can help us clarify fine theoretical points.

Example 5.1. The following derivation, based on the function weight $(C, i) = |C| + i$, follows the second derivation of Example 4.1:

$$\begin{aligned}
&(\{(\mathsf{p}(x), 0), (\neg\mathsf{p(a)} \vee \neg\mathsf{p(b)}, 0)\}, \{\}, \{\}, 1)\\
\rightsquigarrow_{\mathsf{w8}}\ &(\{(\neg\mathsf{p(a)} \vee \neg\mathsf{p(b)}, 0)\}, \{(\mathsf{p}(x), 0)\}, \{\}, 1)\\
\rightsquigarrow_{\mathsf{w8}}\ &(\{\}, \{(\mathsf{p}(x), 0), (\neg\mathsf{p(a)} \vee \neg\mathsf{p(b)}, 0)\}, \{\}, 1)\\
\rightsquigarrow_{\mathsf{w9}}\ &(\{\}, \{(\neg\mathsf{p(a)} \vee \neg\mathsf{p(b)}, 0)\}, \{(\mathsf{p}(x), 0)\}, 2)\\
\rightsquigarrow_{\mathsf{w9}}\ &(\{(\neg\mathsf{p(a)}, 2)\}, \{\}, \{(\mathsf{p}(x), 0), (\neg\mathsf{p(a)} \vee \neg\mathsf{p(b)}, 0)\}, 3)\\
\rightsquigarrow_{\mathsf{w8}}\ &(\{\}, \{(\neg\mathsf{p(a)}, 2)\}, \{(\mathsf{p}(x), 0), (\neg\mathsf{p(a)} \vee \neg\mathsf{p(b)}, 0)\}, 3)\\
\rightsquigarrow_{\mathsf{w9}}\ &(\{(\bot, 3)\}, \{\}, \{(\mathsf{p}(x), 0), (\neg\mathsf{p(a)} \vee \neg\mathsf{p(b)}, 0), (\neg\mathsf{p(a)}, 2)\}, 4)\\
\rightsquigarrow_{\mathsf{w8}}\ &(\{\}, \{(\bot, 3)\}, \{(\mathsf{p}(x), 0), (\neg\mathsf{p(a)} \vee \neg\mathsf{p(b)}, 0), (\neg\mathsf{p(a)}, 2)\}, 4)\\
\rightsquigarrow_{\mathsf{w9}}\ &(\{\}, \{\}, \{(\mathsf{p}(x), 0), (\neg\mathsf{p(a)} \vee \neg\mathsf{p(b)}, 0), (\neg\mathsf{p(a)}, 2), (\bot, 3)\}, 5)
\end{aligned}$$

Due to the weight function, the clause $\mathsf{p}(x)$ must be moved from $\mathcal{P}$ to $\mathcal{O}$ before $\neg\mathsf{p(a)} \vee \neg\mathsf{p(b)}$.

5.2 Refinement Proofs

To lift the soundness and completeness results about RP to $\mathsf{RP_w}$, we must first show that any possible behavior of $\mathsf{RP_w}$ on states of type *wstate* is a possible behavior of RP on the corresponding values of type *state*, without timestamps. Formally:

```
**lemma** *weighted_RP_imp_RP*:
  $\mathcal{S} \rightsquigarrow_{\mathsf{w}} \mathcal{S}' \Longrightarrow$ state_of $\mathcal{S} \rightsquigarrow$ state_of $\mathcal{S}'$
```

The proof is by straightforward induction on the introduction rules of $\mathsf{RP_w}$, with one difficult case. Inference computation (rule 9) converts a set to a finite multiset using mset_set. This operation is undefined for infinite sets. Thus, we must show that from a finite set of clauses, only a finite set of inferences may be performed by infers_between:

```
**lemma** *finite_ord_FO_resolution_inferences_between*:
  finite $\mathcal{D} \Longrightarrow$ finite (infers_between $\mathcal{D}$ $C$)
```

Our formal proof caters for n-ary resolution, but in our application we only need the binary case. A binary resolution inference takes two premises, of the form $CAA = C \vee A_1 \vee \cdots \vee A_k$ and $DA = \neg A \vee D$, and produces a conclusion $E = (C \vee D) \cdot \sigma$. It can be represented compactly by a tuple of the form (CAA, DA, AA, A, E), where $AA = A_1 \vee \cdots \vee A_k$. We must show that the set of such tuples returned by $\mathsf{infers\_between}$ is finite, assuming $\mathcal{D}$ is finite.

First, observe that the E component of a tuple is fully determined by the other four components. Hence it suffices to consider tuples of the form (CAA, DA, AA, A). Let $\mathcal{DC} = \mathcal{D} \cup \{C\}$, and let n be the length of the longest clause in $\mathcal{DC}$. Moreover, let $\mathcal{A} = \bigcup_{D \in \mathcal{DC}} \mathsf{atms\_of}\ D$ and $\mathcal{AA} = \{\mathcal{B} \mid \mathsf{set\_mset}\ \mathcal{B} \subseteq \mathcal{A} \wedge |\mathcal{B}| \leq n\}$. Then all inferences between $\mathcal{D}$ and C belong to $\mathcal{DC} \times \mathcal{DC} \times \mathcal{AA} \times \mathcal{A}$, which is a cartesian product of finite sets.

5.3 Soundness and Completeness Proofs

Using the refinement theorem, it is easy to lift the *RP_model* theorem (Section 4) to $\mathsf{RP_w}$:

> **theorem** *weighted_RP_model*:
> $\mathcal{S} \leadsto_{\mathsf{w}} \mathcal{S}' \Longrightarrow (I \models \mathsf{grounding\_of}\ \mathcal{S}' \Longleftrightarrow I \models \mathsf{grounding\_of}\ \mathcal{S})$

Completeness is considerably more difficult. We first show that the use of timestamps ensures that all full $\mathsf{RP_w}$ derivations are fair. From this fact follows unconditional completeness.

In principle, a full derivation could be unfair by virtue of being finite and ending in a state such as $\mathcal{N}$ or $\mathcal{P}$ is nonempty. However, this is impossible because a transition of rule 8 or 9 could then be taken from the last state, contradicting the hypothesis that the derivation is full. Hence, finite full derivations are necessarily fair:

> **lemma** *fair_if_finite*:
> $\mathsf{lfinite}\ \mathcal{S}s \Longrightarrow \mathsf{fair}\ (\mathsf{lmap\ state\_of}\ \mathcal{S}s)$

There are two ways in which an infinite derivation $\mathcal{S}s$ in $\mathsf{RP_w}$ could be unfair: A clause could get stuck forever in $\mathcal{N}$, or in $\mathcal{P}$. We show that the $\mathcal{N}$ case is impossible by defining a measure on states that decreases with respect to the lexicographic extension of $>$ on natural numbers to pairs, which is a well-founded relation. The measure is

> **abbreviation** $\mathsf{RP\_basic\_measure} :: {'a\ wstate} \Rightarrow nat^2$ **where**
> $\mathsf{RP\_basic\_measure}\ (\mathcal{N}, \mathcal{P}, \mathcal{O}, t) \equiv \big(\mathsf{sum}\ ((\lambda(C, \_).\ |C| + 1)\ \text{`}\ (\mathcal{N} \uplus \mathcal{P} \uplus \mathcal{O})),\ |\mathcal{N}|\big)$

The first component of the pair is the total size of all the clauses in the state, plus 1 for each clause to ensure that empty clauses are counted. The second component is the number of clauses in $\mathcal{N}$.

It is easy to see why the measure is decreasing. Rule 9, inference computation, is not applicable due to our assumption that a clause remains stuck in $\mathcal{N}$. Rule 8, which moves a clause from $\mathcal{N}$ to $\mathcal{P}$, decreases the measure's second component while leaving the first component unchanged. The other rules decrease the first component since they remove clauses or literals. Formally:

> **lemma** *weighted_RP_basic_measure_decreasing_N*:
> $\mathcal{S} \leadsto_{\mathsf{w}} \mathcal{S}' \wedge (C, \_) \in \mathcal{N}\mathsf{\_of}\ \mathcal{S} \Longrightarrow$
> $(\mathsf{RP\_basic\_measure}\ \mathcal{S}', \mathsf{RP\_basic\_measure}\ \mathcal{S}) \in \mathsf{RP\_basic\_rel}$

where $\mathsf{RP\_basic\_rel} = \mathsf{natLess}$ <lex> $\mathsf{natLess}$.

What about the case where a clause C is stuck in $\mathcal{P}$? Lemma *preserve_min_P* (Section 5.1) states that in any step, either all copies of a clause $C \in \mathcal{P}$ are removed or the one with minimum timestamp is preserved. It follows that C's timestamp will either remain stable or decrease over time. Since $>$ is well founded on natural numbers, eventually a fixed i will be reached and will belong to the limit:

lemma *persistent_wclause_in_P_if_persistent_clause_in_P*:
  $C \in \mathsf{Liminf}\ (\mathsf{lmap}\ \mathcal{P}\_\mathsf{of}\ (\mathsf{lmap\ state\_of}\ \mathcal{S}s)) \Longrightarrow$
  $\exists i.\ (C, i) \in \mathsf{Liminf}\ (\mathsf{lmap}\ (\mathsf{set\_mset} \circ \mathcal{P}\_\mathsf{of})\ \mathcal{S}s)$

Again, we define a measure, but it must also decrease when inferences are computed and new clauses appear in $\mathcal{N}$. (In this case, RP_basic_measure may increase.) Our new measure is parameterized by a predicate p that can be used to filter out undesirable clauses:

abbreviation RP_filtered_measure :: $('a\ wclause \Rightarrow bool) \Rightarrow 'a\ wstate \Rightarrow nat^3$ **where**
  RP_filtered_measure $p\ (\mathcal{N}, \mathcal{P}, \mathcal{O}, t) \equiv$
  $\big(\mathsf{sum}\ ((\lambda(C, \_).\ |C|+1)\text{‘}\{Di \in \mathcal{N} \uplus \mathcal{P} \uplus \mathcal{O} \mid p\ Di\}),\ |\{Di \in \mathcal{N} \mid p\ Di\}|,\ |\{Di \in \mathcal{P} \mid p\ Di\}|\big)$

Notice that RP_filtered_measure ($\lambda\_.$ True) essentially amounts to RP_basic_measure. In the formalization, we use RP_filtered_measure ($\lambda\_.$ True) to avoid code duplication.

Suppose the clause C that is stuck in $\mathcal{P}$ has weight w in the limit, and suppose that a clause D is moved from $\mathcal{P}$ to $\mathcal{O}$ by the inference computation rule. That clause's weight must be at most w; otherwise, it would not have been preferred to C.

Infinite derivations necessarily consist of segments each consisting of finitely many applications of rules other than rule 9 followed by an application of rule 9: $(\rightsquigarrow_{\mathsf{w}1\text{–}8}^{*} \circ \rightsquigarrow_{\mathsf{w}9})^{\omega}$. Since each application of rule 9 increases the t component of the state, eventually we reach a state in which $t > w$. As a consequence of strict monotonicity of weight, any clauses generated by inference computation from that point on will have weights above C's, and if C remains stuck, then so must these clauses. Thus, we can ignore these clauses altogether, by using $\lambda(C, i).\ i \leq w$ as the filter p.

We adapt the corresponding relation to consider the extra argument:

abbreviation RP_filtered_rel :: $(nat^3)^2\ set$ **where**
  RP_filtered_rel $\equiv$ natLess <lex> natLess <lex> natLess

The measure RP_filtered_measure $(\lambda(\_, i).\ i \leq w)$ decreases in steps occurring between inference computations and for all steps once we have reached a state where $t > w$ (at which point all inference computations are blocked by C). To obtain a measure that also decreases on inference computation, we add a component $w + 1 - t$ to the measure. We also add a component RP_basic_measure $\mathcal{S}$ to the measure to ensure that it decreases when a clause (C, i) such that $i > w$ is simplified. This yields the combined measure

abbreviation RP_combined_measure :: $nat \Rightarrow 'a\ wstate \Rightarrow nat \times nat^3 \times nat^3$ **where**
  RP_combined_measure $w\ \mathcal{S} \equiv$
  $(w + 1 - \mathsf{t\_of}\ \mathcal{S},$ RP_filtered_measure $(\lambda(\_, i).\ i \leq w)\ \mathcal{S},$ RP_basic_measure $\mathcal{S})$

This measure is indeed decreasing with respect to a left-to-right lexicographic order:

lemma *weighted_RP_basic_measure_decreasing_P*:
  $\mathcal{S} \rightsquigarrow_{\mathsf{w}} \mathcal{S}' \wedge Ci \in \mathcal{P}\_\mathsf{of}\ \mathcal{S} \Longrightarrow$
  (RP_combined_measure (weight Ci) $\mathcal{S}'$, RP_combined_measure (weight Ci) $\mathcal{S}$)
  $\in$ natLess <lex> RP_filtered_rel <lex> RP_basic_rel

By combining the two lemmas *weighted_*RP*_basic_measure_decreasing_N* and *weighted_*RP*_basic_measure_decreasing_P*, we can prove fairness for all derivations starting with $\mathcal{P} = \mathcal{O} = \{\}$:

theorem *weighted_RP_fair*:
  fair (lmap state_of $\mathcal{S}s$)

Since all derivations in $\mathsf{RP_w}$ are fair and its derivations are also derivations of RP, it is trivial to lift RP's saturation and completeness theorems, *RP_saturated_if_fair* and *RP_complete_if_fair*:

corollary *weighted_RP_saturated*:
  $\mathsf{saturated\_upto}\ (\mathsf{Liminf}\ (\mathsf{lmap}\ \mathsf{grounding\_of}\ \mathcal{S}s))$

corollary *weighted_RP_complete*:
  $\neg\ \mathsf{satisfiable}\ (\mathsf{grounding\_of}\ (\mathsf{lhd}\ \mathcal{S}s)) \Longrightarrow \{\} \in \mathcal{O}\mathsf{\_of}\ (\mathsf{Liminf}\ (\mathsf{lmap}\ \mathsf{state\_of}\ \mathcal{S}s))$

6 Eliminating Nondeterminism

The third refinement layer defines a functional program $\mathsf{RP_d}$ that embodies a specific rule application strategy, thereby eliminating the nondeterminism present in $\mathsf{RP_w}$. Clauses are now represented as lists, and multisets of clauses as lists of lists. Although the program is deterministic, some auxiliary functions are specified mathematically and are not directly executable; making these executable is the objective of the fourth refinement layer (Section 7).

6.1 Definition

Our prover corresponds roughly to the following pseudocode:

```
function $\mathsf{RP_d}\ (\mathcal{N}, \mathcal{P}, \mathcal{O}, t)$ is
  repeat forever
    if $\bot \in \mathcal{P} \uplus \mathcal{O}$ then
      return $\mathcal{P} \uplus \mathcal{O}$
    else if $N = P = \{\}$ then
      return $\mathcal{O}$
    else if $N = \{\}$ then
      let $C$ be a minimal-weight clause in $\mathcal{P}$;
      $\mathcal{N}$ := conclusions of all inferences from $\mathcal{O} \uplus \{C\}$ involving $C$, with timestamp $t$;
      move $C$ from $\mathcal{P}$ to $\mathcal{O}$;
      $t := t + 1$
    else
      remove an arbitrary clause $C$ from $\mathcal{N}$;
      reduce $C$ using $\mathcal{P} \uplus \mathcal{O}$;
      if $C = \bot$ then
        return $\{\bot\}$
      else if $C$ is neither a tautology nor subsumed by a clause in $\mathcal{P} \uplus \mathcal{O}$ then
        reduce $\mathcal{P}$ using $C$;
        reduce $\mathcal{O}$ using $C$, moving any reduced clauses from $\mathcal{O}$ to $\mathcal{P}$;
        remove all clauses from $\mathcal{P}$ and $\mathcal{O}$ that are strictly subsumed by $C$;
        add $C$ to $\mathcal{P}$
```

The function should be invoked with $\mathcal{N}$ as the input problem, $\mathcal{P} = \mathcal{O} = \{\}$, and an arbitrary timestamp t. The loop is loosely modeled after the proof procedure implemented in Vampire [Voronkov, 2014, Section 3].

Instead of finite multisets, the actual $\mathsf{RP_d}$ definition in Isabelle uses finite lists, bringing us closer to executable code. The # operator abbreviates the Cons constructor, and @ is the append operator. The list-based representations compel us to introduce the following type abbreviations:

```
    **type_synonym** $'a\ lclause = 'a\ literal\ list$
    **type_synonym** $'a\ dclause = 'a\ lclause \times nat$
    **type_synonym** $'a\ dstate = 'a\ dclause\ list \times 'a\ dclause\ list \times 'a\ dclause\ list \times nat$
```

A state is a tuple $(\mathcal{N}, \mathcal{P}, \mathcal{O}, t)$ as before, but with different types.

The prover is defined inside a locale that inherits *weighted_ FO_ resolution_ prover_ with_ size_ timestamp_ factors*. The core function, $\mathsf{RP_d}$_step, performs a single iteration of the main loop. Here is the complete definition, excluding auxiliary functions:

```
fun RP_d_step :: 'a dstate ⇒ 'a dstate where
  RP_d_step $(\mathcal{N}, \mathcal{P}, \mathcal{O}, t)$ =
  if $\exists Ci \in \mathcal{P} @ \mathcal{O}.$ fst $Ci = []$ then
    $([], [],$ remdups $\mathcal{P} @ \mathcal{O}, t + |$remdups $\mathcal{P}|)$
  else
    (case $\mathcal{N}$ of
      $[] \Rightarrow$
      (case $\mathcal{P}$ of
        $[] \Rightarrow (\mathcal{N}, \mathcal{P}, \mathcal{O}, t)$
      | $P_0 \# \mathcal{P}' \Rightarrow$
        let
          $(C, i) =$ select_min_weight_clause $P_0\ \mathcal{P}'$;
          $\mathcal{N} =$ map $(\lambda D. (D, t))$ (remdups (resolve_rename $C\ C$
            @ concat (map (resolve_rename_either_way $C \circ$ fst) $\mathcal{O}$)));
          $\mathcal{P} =$ filter $(\lambda(D, j).$ mset $D \neq$ mset $C)\ \mathcal{P}$;
          $\mathcal{O} = (C, i) \# \mathcal{O}$;
          $t = t + 1$
        in
          $(\mathcal{N}, \mathcal{P}, \mathcal{O}, t))$
    | $(C, i) \# \mathcal{N} \Rightarrow$
      let
        $C =$ reduce (map fst $(\mathcal{P} @ \mathcal{O})$) $[]\ C$
      in
        if $C = []$ then
          $([], [], [([], i)], t + 1)$
        else if is_tautology $C \vee$ subsume (map fst $(\mathcal{P} @ \mathcal{O})$) $C$ then
          $(\mathcal{N}, \mathcal{P}, \mathcal{O}, t)$
        else
          let
            $\mathcal{P} =$ reduce_all $C\ \mathcal{P}$;
            (back_to_$\mathcal{P}, \mathcal{O}$) = reduce_all2 $C\ \mathcal{O}$;
            $\mathcal{P} =$ back_to_$\mathcal{P} @ \mathcal{P}$;
            $\mathcal{O} =$ filter $((\neg) \circ$ strictly_subsume $[C] \circ$ fst) $\mathcal{O}$;
            $\mathcal{P} =$ filter $((\neg) \circ$ strictly_subsume $[C] \circ$ fst) $\mathcal{P}$;
            $\mathcal{P} = (C, i) \# \mathcal{P}$
          in
            $(\mathcal{N}, \mathcal{P}, \mathcal{O}, t))$
```

The code above relies on some nonexecutable constructs, such as the existential quantifier. The quantifier is unproblematic because it ranges over a finite set, but some of the auxiliary functions rely on infinite quantification. Notably, subsumption of D by C is defined as $\exists \sigma.\ C \cdot \sigma \subseteq D$ (Section 3), where σ ranges over all substitutions. Nonexecutable constructs are acceptable if we know that we can replace them by equivalent executable constructs further down the refinement chain; for example, an implementation of subsumption can compute a finite set of candidates for σ using matching, instead of blindly enumerating all possibilities.

The prover's main program is a tail-recursive function that repeatedly calls $\mathsf{RP_d\_step}$ until a final state, of the form $([], [], \mathcal{O}, t)$, is reached, at which point it returns $\mathcal{O}$ stripped of its timestamps:

> **partial_function** (*option*) $\mathsf{RP_d}$ :: *'a dstate ⇒ 'a lclause list option* **where**
> $\mathsf{RP_d}\ \mathcal{S} = \text{if } \mathsf{is\_final}\ \mathcal{S} \text{ then } \mathsf{Some}\ (\mathsf{map\ fst}\ (\mathcal{O}\mathsf{\_of}\ \mathcal{S})) \text{ else } \mathsf{RP_d}\ (\mathsf{RP_d\_step}\ \mathcal{S})$

Since there are no guarantees that the recursion will terminate, we cannot introduce the function using the **fun** command [Krauss, 2006], which is restricted to well-founded recursion. Instead, we use **partial_function** (*option*) [Krauss, 2010], which puts the computation in an option monad. The function's result is of the form $\mathsf{Some}\ R$ if the recursion terminates and None if the computation diverges. Executing the function would never actually return None, but it is convenient to define it mathematically in this way. For example, it allows us to state and prove a characterization such as the following, which can be used to replace a terminating call $\mathsf{RP_d}\ \mathcal{S}$ by a finite iteration $\mathsf{RP_d\_step}^k\ \mathcal{S}$:

> **lemma** *deterministic_RP_SomeD*:
> $\mathsf{RP_d}\ \mathcal{S} = \mathsf{Some}\ R \Longrightarrow \exists \mathcal{S}'\ k.\ \mathsf{RP_d\_step}^k\ \mathcal{S} = \mathcal{S}' \wedge \mathsf{is\_final}\ \mathcal{S}' \wedge R = \mathsf{map\ fst}\ (\mathcal{O}\mathsf{\_of}\ \mathcal{S}')$

6.2 Refinement Proofs

Using refinement, we connect the $\mathsf{RP_d\_step}$ function to the $\mathsf{RP_w}$ predicate. $\mathsf{RP_d\_step}$ has a coarser granularity than $\mathsf{RP_w}$: A single invocation on a nonfinal state $\mathcal{S}$ can amount to a chain of $\mathsf{RP_w}$ transitions. This is captured by the following weak-refinement property:

> **lemma** *nonfinal_deterministic_RP_step*:
> $\neg\ \mathsf{is\_final}\ \mathcal{S} \Longrightarrow \mathsf{wstate\_of}\ \mathcal{S} \leadsto_{\mathsf{w}}^{+} \mathsf{wstate\_of}\ (\mathsf{RP_d\_step}\ \mathcal{S})$

where $\mathsf{wstate\_of}$ converts $\mathsf{RP_d}$ states to $\mathsf{RP_w}$ states. The entire proof, including key lemmas, is about 1300 lines long. It follows the case distinctions present in the definition of $\mathsf{RP_d\_step}$:

case $\exists Ci \in \mathcal{P} \mathbin{@} \mathcal{O}.\ \mathsf{fst}\ Ci = []$:

By induction on $|\mathsf{remdups}\ \mathcal{P}|$, there must exist a derivation of the form

$$\begin{aligned} & \mathsf{wstate\_of}\ (\mathcal{N}, \mathcal{P}, \mathcal{O}, t) \\ \leadsto_{\mathsf{w2}}^{*}\ & \mathsf{wstate\_of}\ ([], \mathcal{P}, \mathcal{O}, t) \\ \leadsto_{\mathsf{w9}}\ & \mathsf{wstate\_of}\ (\mathcal{N}', \mathcal{P}', (C, i) \mathbin{\#} \mathcal{O}, t+1) \\ \leadsto_{\mathsf{w}}^{*}\ & \mathsf{wstate\_of}\ ([], [], \mathsf{remdups}\ \mathcal{P}' \mathbin{@} \mathcal{O}, t + |\mathsf{remdups}\ \mathcal{P}'|) \end{aligned}$$

for $\mathcal{P}' = \mathsf{filter}\ (\lambda(D, j).\ \mathsf{mset}\ D \neq \mathsf{mset}\ C)\ \mathcal{P}$ and suitable $\mathcal{N}'$ and $(C, i) \in \mathcal{P}$. The last step is justified by the induction hypothesis.

case $\mathcal{N} = \mathcal{P} = []$:

Contradiction with the assumption that $(\mathcal{N}, \mathcal{P}, \mathcal{O}, t)$ is a nonfinal state.

case $\mathcal{N} = []$:

It suffices to show that the transition

$$\mathsf{wstate\_of}\ ([], \mathcal{P}, \mathcal{O}, t) \ \leadsto_{\mathsf{w9}}\ \mathsf{wstate\_of}\ (\mathcal{N}', \mathcal{P}', (C, i) \mathbin{\#} \mathcal{O}, t + 1)$$

is possible, where $(C, i) \in \mathcal{P}$ is a minimal-weight clause and

$$\begin{aligned}\mathcal{N}' &= \mathsf{map}\ (\lambda D.\ (D, t))\ (\mathsf{remdups}\ (\mathsf{resolve\_rename}\ C\ C \\ &\qquad @\ \mathsf{concat}\ (\mathsf{map}\ (\mathsf{resolve\_rename\_either\_way}\ C \circ \mathsf{fst})\ \mathcal{O}))) \\ \mathcal{P}' &= \mathsf{filter}\ (\lambda(D, j).\ \mathsf{mset}\ D \neq \mathsf{mset}\ C)\ \mathcal{P}\end{aligned}$$

The main proof obligation is that $\mathcal{N}'$, converted to multisets, equals the multiset $\mathsf{mset\_set}\ ((\lambda D.\ (D, t))\ `\ \mathsf{concl\_of}\ `\ \mathsf{infers\_between}\ (\mathsf{set\_mset}\ (\mathsf{fst}\ `\ \mathcal{O}))\ C)$ specified in rule $\leadsto_{\mathsf{w9}}$. The distance between the functional program and its mathematical specification is at its greatest here. The proof is tedious but straightforward.

otherwise:

Let $C' = \mathsf{reduce}\ (\mathsf{map}\ \mathsf{fst}\ \mathcal{P}\ @\ \mathsf{map}\ \mathsf{fst}\ \mathcal{O})\ []\ C$. If $C' = []$, then

$$\begin{aligned} & \mathsf{wstate\_of}\ ((C, i) \mathbin{\#} \mathcal{N}', \mathcal{P}, \mathcal{O}, t) \\ \leadsto^*_{\mathsf{w5}}\ & \mathsf{wstate\_of}\ (([], i) \mathbin{\#} \mathcal{N}', \mathcal{P}, \mathcal{O}, t) \\ \leadsto^*_{\mathsf{w3}}\ & \mathsf{wstate\_of}\ (([], i) \mathbin{\#} \mathcal{N}', [], \mathcal{O}, t) \\ \leadsto^*_{\mathsf{w4}}\ & \mathsf{wstate\_of}\ (([], i) \mathbin{\#} \mathcal{N}', [], [], t) \\ \leadsto_{\mathsf{w8}}\ & \mathsf{wstate\_of}\ (\mathcal{N}', [([], i)], [], t) \\ \leadsto^*_{\mathsf{w2}}\ & \mathsf{wstate\_of}\ ([], [([], i)], [], t) \\ \leadsto_{\mathsf{w9}}\ & \mathsf{wstate\_of}\ ([], [], [([], i)], t) \end{aligned}$$

Otherwise, if $\mathsf{is\_tautology}\ C' \vee \mathsf{subsume}\ (\mathsf{map}\ \mathsf{fst}\ (\mathcal{P}\ @\ \mathcal{O}))\ C'$, then

$$\begin{aligned} & \mathsf{wstate\_of}\ ((C, i) \mathbin{\#} \mathcal{N}, \mathcal{P}, \mathcal{O}, t) \\ \leadsto^*_{\mathsf{w5}}\ & \mathsf{wstate\_of}\ ((C', i) \mathbin{\#} \mathcal{N}, \mathcal{P}, \mathcal{O}, t) \\ \leadsto_{\mathsf{w1,2}}\ & \mathsf{wstate\_of}\ (\mathcal{N}, \mathcal{P}, \mathcal{O}, t) \end{aligned}$$

Otherwise:

$$\begin{aligned} & \mathsf{wstate\_of}\ ((C, i) \mathbin{\#} \mathcal{N}', \mathcal{P}, \mathcal{O}, t) \\ \leadsto^*_{\mathsf{w5}}\ & \mathsf{wstate\_of}\ ((C', i) \mathbin{\#} \mathcal{N}', \mathcal{P}, \mathcal{O}, t) \\ \leadsto^*_{\mathsf{w6}}\ & \mathsf{wstate\_of}\ ((C', i) \mathbin{\#} \mathcal{N}', \mathcal{P}', \mathcal{O}, t) \\ \leadsto^*_{\mathsf{w7}}\ & \mathsf{wstate\_of}\ ((C', i) \mathbin{\#} \mathcal{N}', \mathsf{back\_to\_}\mathcal{P}\ @\ \mathcal{P}', \mathcal{O}', t) \\ \leadsto^*_{\mathsf{w4}}\ & \mathsf{wstate\_of}\ ((C', i) \mathbin{\#} \mathcal{N}', \mathsf{back\_to\_}\mathcal{P}\ @\ \mathcal{P}', \mathcal{O}'', t) \\ \leadsto^*_{\mathsf{w3}}\ & \mathsf{wstate\_of}\ ((C', i) \mathbin{\#} \mathcal{N}', \mathcal{P}'', \mathcal{O}'', t) \\ \leadsto_{\mathsf{w8}}\ & \mathsf{wstate\_of}\ (\mathcal{N}', (C', i) \mathbin{\#} \mathcal{P}'', \mathcal{O}'', t) \end{aligned}$$

for suitable clause lists $\mathcal{P}'$, $\mathsf{back\_to\_}\mathcal{P}$, $\mathcal{O}'$, $\mathcal{O}''$, and $\mathcal{P}''$.

The above refinement theorem, about computations from nonfinal states, is complemented by the following trivial result concerning final states:

lemma *final_deterministic_RP_step*:
  $\mathsf{is\_final}\ \mathcal{S} \Longrightarrow \mathsf{RP_d\_step}\ \mathcal{S} = \mathcal{S}$

6.3 Soundness and Completeness Proofs

Let $\mathcal{S}_0 = (\mathcal{N}_0, [], [], t_0)$ be an arbitrary initial state. For $\mathsf{RP_d}$, soundness means that whenever $\mathsf{RP_d}\ \mathcal{S}_0$ terminates with some clause set R, then R is a saturation that satisfies the same models as $\mathcal{N}_0$. In addition, if $\mathcal{N}_0$ is unsatisfiable, then R contains $\bot$, which provides a simple syntactic check for unsatisfiability. Completeness means that divergence is possible only if $\mathcal{N}_0$ is satisfiable. Note that for satisfiable clause sets $\mathcal{N}_0$, both termination and divergence are possible.

To lift soundness and completeness results from $\mathsf{RP_w}$ to $\mathsf{RP_d}$, we first define $\mathcal{S}s$ as a full chain of nontrivial $\mathsf{RP_d}$ steps starting from $\mathcal{S}_0$. Formally, we let $\mathcal{S}s = \mathsf{derivation\_from}\ \mathcal{S}_0$, with

> **primcorec** derivation_from :: *'a dstate* $\Rightarrow$ *'a dstate llist* **where**
> derivation_from $\mathcal{S}$ = LCons $\mathcal{S}$ (if is_final $\mathcal{S}$ then LNil else derivation_from ($\mathsf{RP_d}$_step $\mathcal{S}$))

Based on $\mathcal{S}s$, we let $w\mathcal{S}s = \mathsf{lmap\ wstate\_of}\ \mathcal{S}s$ and note that $w\mathcal{S}s$ is a full chain of "big" $\leadsto_{\mathsf{w}}^{+}$ steps. Using a lemma that will be proved in Section 6.4, we obtain a full chain $ssw\mathcal{S}s$ of "small" $\leadsto_{\mathsf{w}}$ steps. This chain satisfies the conditions postulated on $\mathcal{S}s$ in Section 5.3, allowing us to lift the results presented there.

The soundness results are proved in a nameless locale, or *context*, that assumes termination:

> **context**
> **fixes** R :: *'a lclause list*
> **assumes** $\mathsf{RP_d}\ \mathcal{S}_0$ = Some R

The definition of $\mathsf{RP_d}$, using **partial_function**, gives us an induction rule restricted to the case where $\mathsf{RP_d}$ terminates (i.e., returns a Some value). This rule can be used to prove that $\mathcal{S}s$ and hence $w\mathcal{S}s$ and $ssw\mathcal{S}s$ are finite sequences.

Soundness takes the form of a pair of theorems that lift *weighted_RP_model* and *weighted_RP_saturated*:

> **theorem** *deterministic_RP_model*:
> $I \models$ grounding_of $\mathcal{N}_0 \iff I \models$ grounding_of R
>
> **theorem** *deterministic_RP_saturated*:
> saturated_upto (grounding_of R)

Admittedly, the terminology is somewhat confusing. For RP and $\mathsf{RP_w}$, it is natural—indeed, it is conform to the literature—to classify saturation as a completeness property. However, for finite derivations, such as those considered here, saturations amount to a soundness property.

In most applications, all that matters is the satisfiability status of the set $\mathcal{N}_0$. It can be retrieved syntactically:

> **corollary** *deterministic_RP_refutation*:
> $\neg$ satisfiable (grounding_of $\mathcal{N}_0$) $\iff \{\} \in R$

Completeness is proved in a separate nameless locale that assumes nontermination: $\mathsf{RP_d}\ \mathcal{S}_0$ = None. The strongest result we prove is that this assumption implies the satisfiability of $\mathcal{N}_0$:

> **theorem** *deterministic_RP_complete*:
> satisfiable (grounding_of $\mathcal{N}_0$)

The proof is by contradiction:

> Assume that $\neg$ satisfiable (grounded_of $\mathcal{N}_0$). Hence, by *weighted_RP_complete* we have $\{\} \in \mathcal{O}$_of $ssw\mathcal{S}s$. It is easy to show that $ssw\mathcal{S}s$'s limit is a subset of $w\mathcal{S}s$'s limit; hence $\{\} \in \mathcal{O}$_of $w\mathcal{S}s$. This implies the existence of a natural number k such that $\{\} \in \mathcal{O}$_of (lnth $w\mathcal{S}s$ k). Hence $\{\} \in \mathcal{O}$_of ($\mathsf{RP_d}$_step^k $\mathcal{S}_0$). However, by an induction on k, we can show that $\mathsf{RP_d}$ must terminate after at most k iterations, contradicting the assumption that $\mathsf{RP_d}$ diverges.

6.4 A Coinductive Puzzle

A single "big" step of the deterministic prover $\mathsf{RP_d}$ may consist of multiple "small" steps of the weighted prover $\mathsf{RP_w}$. To transfer the results from $\mathsf{RP_w}$ to $\mathsf{RP_d}$, we must expand $\mathsf{RP_d}$'s big steps. The core of the expansion is an abstract property of chains and a relation's transitive closure:

> Let R be a relation and xs a chain of R^+ transitions. There exists a chain of R transitions that embeds xs—i.e., that contains all elements of xs in the same order and with only finitely many elements inserted between each pair of consecutive elements of xs.

On finite chains, this property would follow by straightforward induction. But the completeness proof must also consider infinite chains. To prove the property on infinite chains requires us to use coinduction and corecursion up-to techniques.

The desired property is formalized as follows:

lemma *chain_tranclp_imp_exists_chain*:
  $\mathsf{chain}\ R^+\ xs \Longrightarrow \exists ys.\ \mathsf{chain}\ R\ ys \wedge xs \sqsubseteq ys \wedge \mathsf{lhd}\ xs = \mathsf{lhd}\ ys \wedge \mathsf{llast}\ xs = \mathsf{llast}\ ys$

where the embedding $\sqsubseteq$ of lazy lists is defined coinductively using the function $+\!\!+$, which prepends a finite list to a lazy list:

coinductive $\sqsubseteq$ :: *'a llist* $\Rightarrow$ *'a llist* $\Rightarrow$ *bool* **where**
  $\mathsf{lfinite}\ xs \Longrightarrow \mathsf{LNil} \sqsubseteq xs$
| $xs \sqsubseteq ys \Longrightarrow \mathsf{LCons}\ x\ xs \sqsubseteq zs +\!\!+ \mathsf{LCons}\ x\ ys$

fun $+\!\!+$ :: *'a list* $\Rightarrow$ *'a llist* $\Rightarrow$ *'a llist* **where**
  $[] +\!\!+ xs = xs$
| $(z \# zs) +\!\!+ xs = \mathsf{LCons}\ z\ (zs +\!\!+ xs)$

The definition of $\sqsubseteq$ ensures that infinite lazy lists only embed other infinite lazy lists, but not the finite ones. Formally: $xs \sqsubseteq ys \Longrightarrow (\mathsf{lfinite}\ xs \Longleftrightarrow \mathsf{lfinite}\ ys)$. The unguarded calls to llast may seem worrying, but the function is conveniently defined to always return the same unspecified element for infinite lists (i.e., $\neg\ \mathsf{lfinite}\ xs \wedge \neg\ \mathsf{lfinite}\ ys \Longrightarrow \mathsf{llast}\ xs = \mathsf{llast}\ ys$).

To prove *chain_tranclp_imp_exists_chain*, we instantiate the existential quantifier by the following corecursively defined witness:

corec wit :: $('a \Rightarrow 'a \Rightarrow \mathit{bool}) \Rightarrow$ *'a llist* $\Rightarrow$ *'a llist* **where**
  $\mathsf{wit}\ R\ xs =$ (case xs of
    $\mathsf{LCons}\ x\ (\mathsf{LCons}\ y\ ys) \Rightarrow \mathsf{LCons}\ x\ (\mathsf{pick}\ R\ x\ y +\!\!+ \mathsf{wit}\ R\ (\mathsf{LCons}\ y\ ys))$
  | $\_ \Rightarrow xs$)

Here $\mathsf{pick}\ R\ x\ y$ returns an arbitrary finite list of R-related intermediate states connecting the R^+-related states x and y. Formally,

$$\mathsf{pick}\ R\ x\ y = \mathsf{SOME}\ zs.\ \mathsf{chain}\ R\ (\mathsf{llist\_of}\ (x \# zs \mathbin{@} [y]))$$

where llist_of converts finite lists into lazy list and SOME is Hilbert's choice operator. Thus, pick satisfies the characteristic property $R^+\ x\ y \Longrightarrow \mathsf{chain}\ R\ (\mathsf{llist\_of}\ (x \# \mathsf{pick}\ R\ x\ y \mathbin{@} [y]))$. The use of Hilbert choice makes pick, and wit, nonexecutable. This is acceptable because these constants are used only in the proofs and not in the actual prover's code.

The definition of wit is not primitively corecursive. Although there is a guarding LCons constructor, the corecursive call occurs under $+\!\!+$, which makes the productivity of this function subtle. This syntactic structure of the definition is called *corecursive up to* $+\!\!+$. Ultimately, wit is productive because $+\!\!+$ does not remove any LCons constructors from its second arguments. A slightly weaker requirement, called *friendliness*, is supported by Isabelle's **corec** command [Blanchette et al., 2017]. Hence, $+\!\!+$ must be registered as a "friend," which involves a one-line proof, for the above definition to be accepted by Isabelle.

The four conjuncts in *chain_ tranclp_ imp_ exists_ chain* are then discharged separately under the common assumption chain R^+ xs. In increasing difficulty: lhd (wit R xs) = lhd xs follows by simple rewriting. Next, llast (wit R xs) = llast xs requires an induction in the case of finite chains xs. For any infinite chain zs of type $'a$ *llist*, llast zs is defined as a fixed but not further specified value of type $'a$. The properties $xs \sqsubseteq$ wit R xs and chain R (wit R xs) require a coinduction on $\sqsubseteq$ and chain, respectively. In keeping with the definitional principle of corecursion up to $+\!\!+$, plain coinduction on $\sqsubseteq$ and chain does not suffice and we must use coinduction up to $+\!\!+$ on $\sqsubseteq$ and chain. We contrast the coinduction (left) and coinduction up to $+\!\!+$ (right) rules for chain:

$$\frac{\begin{array}{r}\forall xs.\ P\ xs \Longrightarrow (\exists z.\ xs = \mathsf{LCons}\ z\ \mathsf{Nil}) \lor \\ (\exists z\ zs.\ xs = \mathsf{LCons}\ z\ zs \land \\ P\ zs \land R\ z\ (\mathsf{lhd}\ zs))\end{array}}{\forall xs.\ P\ xs \Longrightarrow \mathsf{chain}\ R\ xs} \qquad \frac{\begin{array}{r}\forall xs.\ P\ xs \Longrightarrow (\exists z.\ xs = \mathsf{LCons}\ z\ \mathsf{Nil}) \lor \\ (\exists z\ zs\ ys.\ xs = \mathsf{LCons}\ z\ (ys +\!\!+ zs) \land \\ P\ zs \land \mathsf{chain}\ R\ (z \# ys\ @\ [\mathsf{lhd}\ zs])\end{array}}{\forall xs.\ P\ xs \Longrightarrow \mathsf{chain}\ R\ xs}$$

The property *chain_ tranclp_ imp_ exists_ chain* easily extends to full chains (because the last element in the case of finite chains remains unchanged), as required in Section 6.3.

7 Obtaining Executable Code

Our deterministic prover $\mathsf{RP_d}$ is already quite close to being an executable program. There are two main ingredients missing: a concrete representation of terms, over which we have abstracted so far, and an executable algorithm for clause subsumption.

7.1 First-Order Terms

First-order terms are a core data structure in various fields of computer science, be it logic, rewriting, (tree) automata theory, or programming languages (as types of the simply typed λ-calculus). It should be no surprise that various formalizations of terms exist. We instantiate our abstract notion of atoms using a particularly comprehensive formalization of terms developed as part of the IsaFoR library [Thiemann and Sternagel, 2009]. This rewriting-independent part of IsaFoR has recently migrated to the *Archive of Formal Proofs* [Sternagel and Thiemann, 2018].

IsaFoR terms are defined as the following datatype:

$$\begin{array}{l}\textbf{datatype}\ \ ('f, 'v)\ \mathit{term} = \\ \quad \mathsf{Var}\ 'v \\ \mid \mathsf{Fun}\ 'f\ (('f, 'v)\ \mathit{term}\ \mathit{list})\end{array}$$

To simplify notation, in this paper we fix $'f = 'v = \mathit{nat}$ and abbreviate $('f,\ 'v)$ *term* by *term*. In the formalization, polymorphic types are used whenever possible. IsaFoR also defines the standard monadic term substitution $\cdot :: \mathit{term} \Rightarrow ('v \Rightarrow \mathit{term}) \Rightarrow \mathit{term}$ and a function unify $:: (\mathit{term} \times \mathit{term})\ \mathit{list} \Rightarrow \mathit{lsubst} \Rightarrow \mathit{lsubst}$, where $\mathit{lsubst} = ('v \times \mathit{term})\ \mathit{list}$ is the list-based representation of a finite substitution. The function unify computes the MGU for a list of unification constraints that is compatible with a given substitution. IsaFoR includes a wealth of theorems about the defined functions, including the correctness of unify and the well-foundedness of strict term generalization $> :: \mathit{term} \Rightarrow \mathit{term} \Rightarrow \mathit{bool}$ defined by $t > s \Longleftrightarrow (\exists \sigma.\ s \cdot \sigma = t) \land (\nexists \sigma.\ t \cdot \sigma = s)$.

This infrastructure allows us to conveniently instantiate our locales *substitution_ops*, *substitution*, and *mgu*. We instantiate the type $'a$ of atoms with *term* and the type $'s$ of substitutions with $'v \Rightarrow term$ and the constants $\cdot$, id, $\circ$, and atm_of_atms with $\cdot$, Var, $\lambda\sigma\,\tau\,x.\ \sigma\,x \cdot \tau$, and Fun 0, respectively. (The function symbol name 0 is arbitrary.) For the computation of the MGU, there is a slight type mismatch: IsaFoR offers a list-based unifier, whereas our locale requires the type $term\ set\ set \Rightarrow ('v \Rightarrow term)\ option$. It is easy to translate a finite set of finite sets of terms (where the inner sets of terms are the ones to be unified) into a finite list of pairs of constraints. To be executable, the translation requires us to sort the terms contained in a set with respect to some arbitrary (but executable) linear order on terms.

Only the function renamings_apart was not present in IsaFoR. We supply this definition:

```
fun renamings_apart :: $term\ clause\ list \Rightarrow ('v \Rightarrow term)\ list$ where
  renamings_apart [] = []
| renamings_apart ($C$ # $Cs$) =
  let
    $\sigma s$ = renamings_apart $Cs$;
    $\sigma = \lambda v.\ v + \max\ (\{0\} \cup$ vars_clause_list $(Cs \cdot \sigma s)) + 1$
  in $\sigma$ # $\sigma s$
```

where vars_clause_list :: $term\ clause\ list \Rightarrow 'v\ set$ returns the variables contained in a list of clauses. The creation of fresh variable names relies on $'v = nat$.

Finally, the *FO_resolution_prover* locale further requires that the type of atoms supports two comparison operators: a well-order $>$ and a comparison $\succ$ that is stable under substitution (i.e., $B \succ A \Longrightarrow B \cdot \sigma \succ A \cdot \sigma$). Moreover, $>$ and $\succ$ must coincide on ground atoms. Our approach is to instantiate $\succ$ with the Knuth–Bendix order (KBO) [Knuth and Bendix, 1970] on terms, which is formalized in IsaFoR [Sternagel and Thiemann, 2013]. KBO is executable, stable under substitution, well founded, and total on ground terms. The well-order $>$, which must be total on *all* terms, is then defined as an arbitrary extension of a partial well-founded order $\succ$ to a well-order, using Hilbert choice. This makes $>$ nonexecutable, which is unproblematic given that $>$ is used only in proofs and not in the actual prover's code (which relies on $\succ$).

Working with different orders poses a slight technical challenge in Isabelle. Orders are organized as type classes, which are comfortable to work with as they hide the order assumption. However, a type class can be instantiated with a concrete order at most once—in our case by $>$. This instantiation propagates to subsequent definitions, such as sorting or computing the minimum. To use a different order for sorting, we must resort to lower-level definitions that are explicitly parameterized by the comparison operation. This is inconvenient when we are defining programs and even more so when we are reasoning about them.

7.2 Clause Subsumption

The second hurdle concerns clause subsumption. Its mathematical definition, subsumes $C\ D \Longleftrightarrow \exists\sigma.\ C \cdot \sigma \subseteq D$, involves an infinite quantification ranging over substitutions.

The problem of deciding whether such a substitution exists is NP-complete [Kapur and Narendran, 1986]. We start with the following naive code. In contrast to the mathematical definition, which operates on multisets of literals, our function operates on lists:

```
fun subsumes_list :: $term\ literal\ list \Rightarrow term\ literal\ list \Rightarrow osubst \Rightarrow bool$ where
  subsumes_list [] $Ks\ \sigma$ = True
| subsumes_list ($L$ # $Ls$) $Ks\ \sigma$ =
  ($\exists K \in$ set $Ks$. is_pos $K$ = is_pos $L$ $\wedge$
      case match_term_list [(atm_of $L$, atm_of $K$)] $\sigma$ of
        None $\Rightarrow$ False
      | Some $\rho \Rightarrow$ subsumes_list $Ls$ (remove1 $K$ $Ks$) $\rho$)
```

In the type declaration, *osubst* abbreviates $'v \Rightarrow$ *term option*. The function recurses on its first argument. In the recursive case, we must consider all possible matching literals for L from Ks compatible with the substitution σ. The bounded existential quantification that expresses this nondeterminism can be executed by iterating over the finite list Ks. The functions is_pos and atm_of are the discriminator and selector for literals. The function match_term_list is provided by IsaFoR. It attempts to extend a given substitution into Some matcher for a list of matching constraints, given as term pairs. If the extension is impossible, match_term_list returns None. This substitution-passing style is typical of purely functional implementations of unification and matching procedures and is inherited by our subsumes_list.

It is easy to prove that the above executable function correctly implements clause subsumption: subsumes (mset Ls) (mset Ks) $=$ subsumes_list Ls Ks ($\lambda x.$ None), where mset converts lists to multisets by forgetting the order of the elements. After the registration of this equation, Isabelle's code generator will rewrite any code that contains the nonexecutable left-hand side to use the executable right-hand side instead.

Clause subsumption is a hot spot in a resolution prover. This has led to the empirical studies of various heuristics to improve on the naive exhaustive search [Schulz, 2013a; Tammet, 1998]. Following Tammet [1998], we implement a heuristic that often reduces the number of calls to match_term_list, which is linear in the size of the input terms, by first performing a simpler, imprecise comparison. For example, terms with different root symbols will never match, and these can be compared in constant time. Similarly, literals with opposite polarities cannot match. Accordingly, we sort our (list-represented) clauses with respect to a literal quasi-order (i.e., a transitive and reflexive relation) leq_lit such that

$$\text{is\_pos}\ L = \text{is\_pos}\ K \wedge \text{match\_term\_list}\ [(\text{atm\_of}\ L,\ \text{atm\_of}\ K)]\ \sigma = \text{Some}\ \rho \Longrightarrow \text{leq\_lit}\ L\ K$$

Any quasi-order satisfying this property can be used in a refinement of subsumes_list to remove too small literals (with respect to leq_lit), as highlighted in gray below:

```
  fun subsumes_list' :: term literal list ⇒ term literal list ⇒ osubst ⇒ bool where
    subsumes_list' [] $Ks$ $\sigma$ = True
  | subsumes_list' ($L$ # $Ls$) $Ks$ $\sigma$ =
      let $Ks$ = filter (leq_lit $L$) $Ks$ in
        ($\exists K \in$ set $Ks$. is_pos $K$ = is_pos $L$ $\wedge$
          case match_term_list [(atm_of $L$, atm_of $K$)] $\sigma$ of
            None $\Rightarrow$ False
          | Some $\rho \Rightarrow$ subsumes_list' $Ls$ (remove1 $K$ $Ks$) $\rho$)
```

The theorem subsumes_list Ls Ks ρ $=$ subsumes_list′ (sort leq_lit Ls) Ks ρ allows the code generator to refine the unoptimized version. In our prover, we let leq_lit be a quasi-order that considers negative literals smaller than positive ones, that considers variables smaller than non-variable terms, and that sorts terms according to a total order on their root symbols.

This refinement is a local optimization: It requires us to explicitly sort one of the input clauses. A more efficient but also more intrusive refinement would be to maintain the invariant that all clauses in the prover's state are sorted with respect to leq_lit. Sorting Ls for each invocation of clause subsumption could then be avoided, and filtering Ks could be performed more efficiently. However, maintaining the invariant would require changes throughout the prover's code.

7.3 The End Result

Finally, Isabelle can export our prover to Standard ML, Haskell, OCaml, or Scala. The command

export_code prover **in** *SML* **module_name** *RP*

generates a Standard ML module containing the implementation of our prover in slightly more than 1000 lines of code, including dependencies. The generated module exports the ML function

```
val prover : ((nat, nat) term literal list * nat) list -> bool
```

Even though in Isabelle we have proved that for any unsatisfiable input prover will terminate and return False, the code generator guarantees only partial correctness of its output: If the generated program terminates on the ML input generated from the Isabelle term t and evaluates to the Boolean result b, the proposition prover $t = b$ is provable in Isabelle. (There is recent work towards providing stronger guarantees [Hupel and Nipkow, 2018].) By soundness, we also know that the Boolean b indicates the satisfiability of the input clause set.

After we have worked hard to obtain an executable prover, it would be a shame not to run it on some example. We selected benchmark MSC015 from the TPTP library [Sutcliffe, 2017], a particularly challenging family Φ_n of first-order problems. Each problem consists of the following $n + 2$ clauses (2 unit clauses and n two-literal clauses):

$$\begin{array}{rcl} \neg\,\mathsf{p}(\mathsf{b}, \ldots, \mathsf{b}) & & \mathsf{p}(\mathsf{a}, \ldots, \mathsf{a}) \\ \neg\,\mathsf{p}(\mathsf{a}, \mathsf{b}, \ldots, \mathsf{b}) & \vee & \mathsf{p}(\mathsf{b}, \mathsf{a}, \ldots, \mathsf{a}) \\ \neg\,\mathsf{p}(x_1, \mathsf{a}, \mathsf{b}, \ldots, \mathsf{b}) & \vee & \mathsf{p}(x_1, \mathsf{b}, \mathsf{a}, \ldots, \mathsf{a}) \\ & \vdots & \\ \neg\,\mathsf{p}(x_1, \ldots, x_{n-2}, \mathsf{a}, \mathsf{b}) & \vee & \mathsf{p}(x_1, \ldots, x_{n-2}, \mathsf{b}, \mathsf{a}) \\ \neg\,\mathsf{p}(x_1, \ldots, x_{n-2}, x_{n-1}, \mathsf{a}) & \vee & \mathsf{p}(x_1, \ldots, x_{n-2}, x_{n-1}, \mathsf{b}) \end{array}$$

A comment in the benchmark warns us that back in 2007, no prover could solve the Φ_{23} within an hour. Even in 2018, only one prover solves Φ_{22} within 300 s, and four provers solve Φ_{20} within 300 s. Our verified prover solves Φ_{20} in 100 s and Φ_{22} in 200 s. Although our prover cannot yet challenge state-of-the-art provers in general, its performance is respectable and could be improved further using refinement.

8 Discussion and Related Work

We found Bachmair and Ganzinger's [2001] chapter and its formalization by Schlichtkrull et al. [2018a,b] suitable as a starting point for a verified prover. Nonetheless, we faced some difficulties, notably concerning the identification of suitable refinement layers. We developed layers 2, 3, and 4 largely in parallel, with each of the authors working on a separate layer. Bringing layer 2 into a state such that it both ensures fairness and could be refined further by layer 3 required several iterations.

Stepwise refinement helped us achieve separation of concerns: fairness, determinism, and executability were achieved successively. Another strength of refinement is that it allows us to prove results at a high level of abstraction; for example, the fairness of layer 2 is inherited by layers 3 and 4 and could be inherited by further layers. The main weakness of refinement is that some nontrivial machinery is necessary to lift results from one layer to the next. We believe the gain in modularity makes this worthwhile.

It took us quite some time to design a suitable measure to prove the fairness of the layer 2 prover $\mathsf{RP_w}$. Our solution amounts to advancing to a state carrying a suitably high timestamp and filtering out all overly heavy clauses. Initially, our proof consisted of two steps—advancing and filtering—each with its own measure. This proof gave us the insurance that $\mathsf{RP_w}$ was fair, but we found that combining the measures is both more succinct and more intelligible.

The main goal of our formalization effort was not to obtain a "QED" as quickly as possible but to investigate how to harness a modern proof assistant to formalize the metatheory of automatic theorem provers. We found Isabelle suitable for this verification task. The Isar proof language allows us to state key intermediate steps, as in a paper proof. Standard tactics, including Isabelle's simplifier, can be used to discharge proof obligations. The Sledgehammer tool [Paulson and Blanchette, 2012] uses superposition provers and SMT (satisfiability-modulo-theories) solvers to swiftly identify which lemmas are necessary to prove a goal; standard Isabelle tactics are then used to certify the proof. Isabelle's support for coinductive methods, including the **coinductive**, **codatatype**, and **corec** commands, helps reason about infinite processes. Locales are a useful abstraction for defining the refinement layers. And the libraries included in the Isabelle distribution, the *Archive of Formal Proofs*, and the third-party IsaFoR certainly saved us months of work.

The *Archive* also includes a refinement framework [Lammich, 2013], which has been used in a separate effort to connect the imperative code of an efficient SAT solver to an abstract calculus [Blanchette et al., 2018]. The framework is helpful in a variety of situations, including when the refinement relation between a concrete and an abstract data representation is not a function. But since converting a list to a multiset (between our levels 3 and 2) or a multiset to a set (between levels 2 and 1) *is* a function, we did not see a need to employ it. Moreover, the framework is currently not designed for refining semidecision procedures, as acknowledged privately by its author, Peter Lammich. We conjecture that its support for separation logic could be useful if we were to refine the prover further to obtain imperative code.

Thanks to the verification, we can trust to a very high extent that our ordered resolution prover is sound and complete. To make the prover's performance competitive with E, SPASS, and Vampire, we would need to extend the current work along two axes. First, we should use superposition, together with its extensive simplification machinery, as the base calculus. A good starting point would be to apply our methodology to Peltier's [2016] formalization of (a generalization of) superposition. Given that most of a modern superposition prover's code consists of heuristics, which are easy to verify, the full verification of a competitive superposition prover appears to be a realistic objective for a forthcoming Ph.D. thesis. Second, the chain of refinement should be continued to cover optimized algorithms and data structures. These could be specified by refining layer 4 further, along the lines of Fleury et al.'s [2018] refinement of an imperative SAT solver.

In computer science, metatheories and implementations are often left unconnected. A metatheory may inspire an implementation, or vice versa, but the connection is rarely made explicit. By formalizing the metatheory, the implementation, and their connection, we can demonstrate not only the implementation's correctness but also the metatheory's adequacy for describing potential implementations. In particular, we have now confirmed that Bachmair and Ganzinger [2001] (with the exceptions noted by Schlichtkrull et al. [2018b]) accurately describe the abstract principles of an executable functional prover, even though they provide few details beyond layer 1.

We built our verified prover on Schlichtkrull et al.'s [2018a,b] formalization of ordered resolution. Related efforts, developed using Isabelle/HOL, include Peltier's [2016] formalization of superposition and Schlichtkrull's [2018] formalization of unordered resolution. However, these developments only cover logical calculi; had we started with any of them, the first step would have been to define an abstract prover in the style of layer 1 and prove basic properties about it. Another related effort is Hirokawa et al.'s [2017] formalization of ordered completion, which (like ordered resolution) can be regarded as a special case of superposition.

Formalizing a theorem proving tool using a theorem proving tool is a thrilling (if self-referential) prospect for many researchers. An early result is Ridge and Margetson's [2005] verified first-order prover, based on a sequent calculus for first-order logic without full first-order terms but only variables. Kumar et al. [2016] formalized the soundness of a proof assistant for higher-order logic. Jensen et al. [2018] verified the soundness of a kernel for a proof assistant for first-order logic that includes a tableau prover. There are several verified SAT solvers [Blanchette et al., 2018; Lescuyer, 2011; Marić, 2008, 2010; Oe et al., 2012; Shankar and Vaucher, 2011]. Among these, two implement efficient data structures such as the two watched literals [Blanchette et al., 2018; Oe et al., 2012]. SAT being a decidable problem, termination has been proved for most solvers. First-order logic, on the other hand, is semidecidable, which is partly what makes our present work original. Lifting, via refinement, an abstract completeness result expressed in terms of the limit of a possibly infinite derivation to a possibly nonterminating functional program is something we have not found anywhere in the literature.

A pragmatic approach to combine the efficiency of unverified code with the trustworthiness of verified code consists of checking certificates produced by reasoning tools—e.g., proofs produced by SAT solvers [Cruz-Filipe et al., 2017; Lammich, 2017]. Researchers from the first-order theorem proving community are now advocating this approach for their systems as well [Reger and Suda, 2017]. An ad hoc version of this approach is used in Sledgehammer and similar tools to reconstruct proofs found by first-order provers [Blanchette et al., 2016; Kaliszyk and Urban, 2013].

9 Conclusion

Starting from Schlichtkrull et al.'s [2018a,b] abstract specification of an ordered resolution prover, we verified, through a refinement chain, a purely functional prover that uses lists as its main data structure. The resulting program is interesting in its own right and could be refined further to obtain an implementation that is competitive with the state of the art.

The stepwise refinement methodology is a keystone of our approach, and we found it entirely adequate for this kind of work. Each refinement step cleanly isolates concerns, yielding intelligible proof obligations. Refinement also helped us identify an unnecessary assumption in Bachmair and Ganzinger [2001] and generally clarify the argument. Lifting results from one layer to another required some thought, especially the completeness results, which correspond to liveness properties. Having now established a methodology and built basic formal libraries, we expect that verifying other saturation-based provers, using Isabelle/HOL or other systems, will be substantially easier.

Acknowledgements Alexander Bentkamp, Mathias Fleury, Andreas Halkjær From, Carsten Fuhs, Peter Lammich, Mark Summerfield, and Jørgen Villadsen suggested many textual improvements. Blanchette has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 713999, Matryoshka).

References

Bachmair, L., Dershowitz, N., and Plaisted, D. A. (1989). Completion without failure. In Aït-Kaci, H. and Nivat, M., editors, *Rewriting Techniques—resolution of Equations in Algebraic Structures*, volume 2, pages 1–30. Academic Press.

Bachmair, L. and Ganzinger, H. (2001). Resolution theorem proving. In Robinson, A. and Voronkov, A., editors, *Handbook of Automated Reasoning*, volume I, pages 19–99. Elsevier and MIT Press.

Ballarin, C. (2014). Locales: A module system for mathematical theories. *Journal of Automated Reasoning*, 52(2):123–153.

Bertot, Y. and Castéran, P. (2004). *Interactive Theorem Proving and Program Development—Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. Springer.

Biendarra, J., Blanchette, J. C., Bouzy, A., Desharnais, M., Fleury, M., Hölzl, J., Kuncar, O., Lochbihler, A., Meier, F., Panny, L., Popescu, A., Sternagel, C., Thiemann, R., and Traytel, D. (2017). Foundational (co)datatypes and (co)recursion for higher-order logic. In Dixon, C. and Finger, M., editors, *FroCoS 2017*, volume 10483 of *LNCS*, pages 3–21. Springer.

Blanchette, J. C., Böhme, S., Fleury, M., Smolka, S. J., and Steckermeier, A. (2016). Semi-intelligible Isar proofs from machine-generated proofs. *Journal of Automated Reasoning*, 56(2):155–200.

Blanchette, J. C., Bouzy, A., Lochbihler, A., Popescu, A., and Traytel, D. (2017). Friends with benefits: Implementing corecursion in foundational proof assistants. In Yang, H., editor, *ESOP 2017*, volume 10201 of *LNCS*, pages 111–140. Springer.

Blanchette, J. C., Fleury, M., Lammich, P., and Weidenbach, C. (2018). A verified SAT solver framework with learn, forget, restart, and incrementality. *Journal of Automated Reasoning*, 61(1–4):333–365.

Bobot, F., Filliâtre, J.-C., Marché, C., and Paskevich, A. (2011). Why3: Shepherd your herd of provers. In Leino, K. R. M. and Moskal, M., editors, *Boogie 2011*, pages 53–64.

Bove, A., Dybjer, P., and Norell, U. (2009). A brief overview of Agda—a functional language with dependent types. In Berghofer, S., Nipkow, T., Urban, C., and Wenzel, M., editors, *TPHOLs 2009*, volume 5674 of *LNCS*, pages 73–78. Springer.

Church, A. (1940). A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(2):56–68.

Cruz-Filipe, L., Heule, M. J. H., Jr., W. A. H., Kaufmann, M., and Schneider-Kamp, P. (2017). Efficient certified RAT verification. In de Moura, L., editor, *CADE-26*, volume 10395 of *LNCS*, pages 220–236. Springer.

Fleury, M., Blanchette, J. C., and Lammich, P. (2018). A verified SAT solver with watched literals using Imperative HOL. In Andronick, J. and Felty, A. P., editors, *CPP 2018*, pages 158–171. ACM.

Gordon, M. J. C., Milner, R., and Wadsworth, C. P. (1979). *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *LNCS*. Springer.

Haftmann, F. and Nipkow, T. (2010). Code generation via higher-order rewrite systems. In Blume, M., Kobayashi, N., and Vidal, G., editors, *FLOPS 2010*, volume 6009 of *LNCS*, pages 103–117. Springer.

Hirokawa, N., Middeldorp, A., Sternagel, C., and Winkler, S. (2017). Infinite runs in abstract completion. In Miller, D., editor, *FSCD 2017*, volume 84 of *LIPIcs*, pages 19:1–19:16. Schloss Dagstuhl—Leibniz-Zentrum für Informatik.

Hupel, L. and Nipkow, T. (2018). A verified compiler from Isabelle/HOL to CakeML. In Ahmed, A., editor, *ESOP 2018*, volume 10801 of *LNCS*, pages 999–1026. Springer.

Jensen, A. B., Larsen, J. B., Schlichtkrull, A., and Villadsen, J. (2018). Programming and verifying a declarative first-order prover in Isabelle/HOL. *AI Commun.*, 31(3):281–299.

Kaliszyk, C. and Urban, J. (2013). PRocH: Proof reconstruction for HOL Light. In Bonacina, M. P., editor, *CADE-24*, volume 7898 of *LNCS*, pages 267–273. Springer.

Kapur, D. and Narendran, P. (1986). NP-completeness of the set unification and matching problems. In Siekmann, J. H., editor, *CADE-8*, volume 230 of *LNCS*, pages 489–495. Springer.

Knuth, D. E. and Bendix, P. B. (1970). Simple word problems in universal algebras. In Leech, J., editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press.

Kovács, L. and Voronkov, A. (2009). Finding loop invariants for programs over arrays using a theorem prover. In Watt, S. M., Negru, V., Ida, T., Jebelean, T., Petcu, D., and Zaharie, D., editors, *SYNASC 2009*, page 10. IEEE Computer Society.

Kovács, L. and Voronkov, A. (2013). First-order theorem proving and Vampire. In Sharygina, N. and Veith, H., editors, *CAV 2013*, volume 8044 of *LNCS*, pages 1–35. Springer.

Krauss, A. (2006). Partial recursive functions in higher-order logic. In Furbach, U. and Shankar, N., editors, *IJCAR 2006*, volume 4130 of *LNCS*, pages 589–603. Springer.

Krauss, A. (2010). Recursive definitions of monadic functions. *EPTCS*, 43:1–13.

Kumar, R., Arthan, R., Myreen, M. O., and Owens, S. (2016). Self-formalisation of higher-order logic: Semantics, soundness, and a verified implementation. *Journal of Automated Reasoning*, 56(3):221–259.

Lammich, P. (2013). Automatic data refinement. In Blazy, S., Paulin-Mohring, C., and Pichardie, D., editors, *ITP 2013*, volume 7998 of *LNCS*, pages 84–99. Springer.

Lammich, P. (2017). The GRAT tool chain—efficient (UN)SAT certificate checking with formal correctness guarantees. In Gaspers, S. and Walsh, T., editors, *SAT 2017*, volume 10491 of *LNCS*, pages 457–463. Springer.

Lamport, L. (1995). How to write a proof. *American Mathematical Monthly*, 7(102):600–608.

Lescuyer, S. (2011). *Formalizing and Implementing a Reflexive Tactic for Automated Deduction in Coq*. PhD thesis, Université Paris-Sud.

Marić, F. (2008). Formal verification of modern SAT solvers. *Archive of Formal Proofs*. Formal Proof Development. `http://isa-afp.org/entries/SATSolverVerification.html`.

Marić, F. (2010). Formal verification of a modern SAT solver by shallow embedding into Isabelle/HOL. *Theoretical Computer Science*, 411(50):4333–4356.

Milner, R. (1984). The use of machines to assist in rigorous proof. *Philosophical Transactions of the Royal Society A*, 312:411–422.

Nipkow, T. and Klein, G. (2014). *Concrete Semantics: With Isabelle/HOL*. Springer.

Nipkow, T., Paulson, L. C., and Wenzel, M. (2002). *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer.

Oe, D., Stump, A., Oliver, C., and Clancy, K. (2012). `versat`: A verified modern SAT solver. In Kuncak, V. and Rybalchenko, A., editors, *VMCAI 2012*, volume 7148 of *LNCS*, pages 363–378. Springer.

Paulson, L. C. and Blanchette, J. C. (2012). Three years of experience with Sledgehammer, a practical link between automatic and interactive theorem provers. In Sutcliffe, G., Schulz, S., and Ternovska, E., editors, *IWIL-2010*, volume 2 of *EPiC Series in Computing*, pages 1–11. EasyChair.

Peltier, N. (2016). A variant of the superposition calculus. *Archive of Formal Proofs*. Formal Proof Development. `http://isa-afp.org/entries/SuperCalc.html`.

Reger, G. and Suda, M. (2017). Checkable proofs for first-order theorem proving. In Reger, G. and Traytel, D., editors, *ARCADE 2017*, volume 51 of *EPiC Series in Computing*, pages 55–63. EasyChair.

Ridge, T. and Margetson, J. (2005). A mechanically verified, sound and complete theorem prover for first order logic. In Hurd, J. and Melham, T., editors, *TPHOL's 2005*, volume 3603 of *LNCS*, pages 294–309. Springer.

Schlichtkrull, A. (2018). Formalization of the resolution calculus for first-order logic. *Journal of Automated Reasoning*, 61(1–4):455–484.

Schlichtkrull, A., Blanchette, J. C., Traytel, D., and Waldmann, U. (2018a). Formalization of Bachmair and Ganzinger's ordered resolution prover. *Archive of Formal Proofs*. Formal Proof Development. `http://isa-afp.org/entries/Ordered_Resolution_Prover.html`.

Schlichtkrull, A., Blanchette, J. C., Traytel, D., and Waldmann, U. (2018b). Formalizing Bachmair and Ganzinger's ordered resolution prover. In Galmiche, D., Schulz, S., and Sebastiani, R., editors, *IJCAR 2018*, volume 10900 of *LNCS*, pages 89–107. Springer.

Schulz, S. (2013a). Simple and efficient clause subsumption with feature vector indexing. In Bonacina, M. P. and Stickel, M. E., editors, *Automated Reasoning and Mathematics—Essays in Memory of William W. McCune*, volume 7788 of *LNCS*, pages 45–67. Springer.

Schulz, S. (2013b). System description: E 1.8. In McMillan, K., Middeldorp, A., and Voronkov, A., editors, *LPAR-19*, volume 8312 of *LNCS*, pages 735–743. Springer.

Shankar, N. and Vaucher, M. (2011). The mechanical verification of a DPLL-based satisfiability solver. *Electronic Notes in Theoretical Computer Science*, 269:3–17. LSFA 2010.

Sternagel, C. and Thiemann, R. (2013). Formalizing Knuth-Bendix orders and Knuth-Bendix completion. In van Raamsdonk, F., editor, *RTA 2013*, volume 21 of *LIPIcs*, pages 287–302. Schloss Dagstuhl—Leibniz-Zentrum für Informatik.

Sternagel, C. and Thiemann, R. (2018). First-order terms. *Archive of Formal Proofs*. Formal Proof Development. `http://isa-afp.org/entries/First_Order_Terms.html`.

Sutcliffe, G. (2017). The TPTP problem library and associated infrastructure—from CNF to th0, TPTP v6.4.0. *Journal of Automated Reasoning*, 59(4):483–502.

Tammet, T. (1998). Towards efficient subsumption. In Kirchner, C. and Kirchner, H., editors, *CADE-15*, volume 1421 of *LNCS*, pages 427–441. Springer.

Thiemann, R. and Sternagel, C. (2009). Certification of termination proofs using CeTA. In Berghofer, S., Nipkow, T., Urban, C., and Wenzel, M., editors, *TPHOLs 2009*, volume 5674 of *LNCS*, pages 452–468. Springer.

Voronkov, A. (2014). AVATAR: The architecture for first-order theorem provers. In Biere, A. and Bloem, R., editors, *CAV 2014*, volume 8559 of *LNCS*, pages 696–710. Springer.

Weidenbach, C., Dimova, D., Fietzke, A., Kumar, R., Suda, M., and Wischnewski, P. (2009). SPASS version 3.5. In Schmidt, R. A., editor, *CADE-22*, volume 5663 of *LNCS*, pages 140–145. Springer.

Wenzel, M. (2007). Isabelle/Isar—a generic framework for human-readable proof documents. In Matuszewski, R. and Zalewska, A., editors, *From Insight to Proof: Festschrift in Honour of Andrzej Trybulec*, volume 10(23) of *Studies in Logic, Grammar, and Rhetoric*. University of Białystok.

Wenzel, M. (2012). Isabelle/jEdit—a prover IDE within the PIDE framework. In Jeuring, J., Campbell, J. A., Carette, J., Reis, G. D., Sojka, P., Wenzel, M., and Sorge, V., editors, *CICM 2012*, volume 7362 of *LNCS*, pages 468–471. Springer.

Wirth, N. (1971). Program development by stepwise refinement. *Communications of the ACM*, 14(4).

NaDeA: A Natural Deduction Assistant with a Formalization in Isabelle

Jørgen Villadsen, Alexander Birch Jensen, and Anders Schlichtkrull

DTU Compute, Technical University of Denmark, Kongens Lyngby, Denmark

Abstract

We present a new software tool for teaching logic based on natural deduction. Its proof system is formalized in the proof assistant Isabelle such that its definition is very precise. Soundness of the formalization has been proved in Isabelle. The tool is open source software developed in TypeScript / JavaScript and can thus be used directly in a browser without any further installation. Although developed for computer science bachelor students who are used to study and program concrete computer code in a programming language we consider the approach relevant for a broader audience and for other proof systems as well.

Keywords Natural Deduction, Formalization, Isabelle Proof Assistant, First-Order Logic, Higher-Order Logic.

1 Introduction

In this paper we present the NaDeA software tool. First, we provide the motivation and a short description. We then present the natural deduction system as it is done in a popular textbook [15] and as is it done in NaDeA by looking at its formalization in Isabelle. This illustrates the differences between the two approaches. We also present the semantics of first-order logic as formalized in Isabelle. Thereafter we explain how NaDeA is used to construct a natural deduction proof. After that, we explain the soundness proof of the natural deduction proof system in Isabelle. Lastly, we compare NaDeA to other natural deduction assistants and consider how NaDeA could be improved.

1.1 Motivation

We have been teaching a bachelor logic course — with logic programming — for a decade using a textbook with emphasis on tableaux and resolution [1]. We have started to use the proof assistant Isabelle [2] and refutation proofs are less preferable here. The proof system of natural deduction [3, 4, 5, 15] with the introduction and elimination rules as well as a discharge mechanism seems more suitable. The natural deduction proof system is widely known, used and studied among

The present article substantially extends our presentation and tool demonstration at TTL 2015 (Fourth International Conference on Tools for Teaching Logic, Rennes, France, 9–12 June 2015). In particular, a new section with elaboration on the formalization in Isabelle has been added. We would like to thank Stefan Berghofer, Jasmin Christian Blanchette, Mathias Fleury and Dmitriy Traytel for discussions about the formalization of logical inference systems in Isabelle. We would also like to thank Andreas Halkjær From, Andreas Viktor Hess, John Bruntse Larsen, Ashokaditya Mohanty and the referees for comments on the paper.

logicians throughout the world. However, our experience shows that many of our computer science bachelor students struggle to understand the most difficult aspects.

This also goes for other proof systems. We find that teaching logic to computer science bachelor students can be hard because in our case they do not have a strong theoretical mathematical background. Instead, most students are good at understanding concrete computer code in a programming language. The syntax used in Isabelle is in many ways similar to a programming language. A clear and explicit formalization of first-order logic and a proof system may help the students in understanding important details.

We find it important to teach both the semantics of first-order logic and the soundness proof to bachelor students. In the present course the formal semantics as well as the soundness proof in Isabelle are presented to the students. The formalization is also available online in NaDeA and the entire Isabelle file is available in NaDeA too. However, in the present course the students are not expected to be able to construct such a formalization in Isabelle from scratch.

The proof assistant Isabelle is different from a programming language because the expressions are not necessarily computable. For instance, quantifications over infinite domains are possible.

1.2 The Tool

We present the natural deduction assistant NaDeA with a formalization of its proof system in the proof assistant Isabelle. It can be used directly in a browser without any further installation and is available here:

`http://nadea.compute.dtu.dk/`

NaDeA is open source software developed in TypeScript / JavaScript and stored on GitHub. The formalization of its proof system in Isabelle is available here:

`http://logic-tools.github.io/`

Once NaDeA is loaded in the browser — about 250 KB with the jQuery Core library — no internet connection is required. Therefore NaDeA can also be stored locally.

We present the proof in an explicit code format that is equivalent to the Isabelle syntax, but with a few syntactic differences to make it easier to understand for someone trying to learn Isabelle. In this format, we present the proof in a style similar to that of Fitch's diagram proofs. We avoid the seemingly popular Gentzen's tree style to focus less on a visually pleasing graphical representation that is presumably much more challenging to implement.

We find that the following requirements constitute the key ideals for any natural deduction assistant. It should be:

- Easy to use.
- Clear and explicit in every detail of the proof.
- Based on a formalization that can be proved at least sound, but preferably also complete.

Based on this, we saw an opportunity to develop NaDeA which offers help for new users, but also serves to present an approach that is relevant to the advanced users.

In a paper considering the tools developed for teaching logic over the last decade [14, p. 137], the following is said about assistants (not proof assistants like Isabelle but tools for learning/teaching logic):

> Assistants are characterized by a higher degree of interactivity with the user. They provide menus and dialogues to the user for interaction purposes. This kind of tool gives the students the feeling that they are being helped in building the solution. They provide error messages and hints in the guidance to the construction of the answer. Many of them usually offer construction of solution in natural deduction proofs. [...] They are usually free licensed and of open access.

We think that this characterization in many ways fits NaDeA. While NaDeA might not bring something new to the table in the form of delicate graphical features, we emphasize the fact that it has some rather unique features such as a formalization of its proof system in Isabelle.

2 Natural Deduction in a Textbook

We consider natural deduction as presented in a popular textbook on logic in computer science [15]. First, we take a look at substitution, which is central to the treatment of quantifiers in natural deduction.

2.1 On Substitution

The following definition for substitution is used in [15, p. 105 top]:

> Given a variable x, a term t and a formula ϕ we define $\phi[t/x]$ to be the formula obtained by replacing each free occurrence of variable x in ϕ with t.

The usual side conditions that come with rules using this substitution seem to be omitted, but we are shortly after [15, p. 106 top] given the following definition of what it means that 't must be free for x in ϕ':

> Given a term t, a variable x and a formula ϕ, we say that t is free for x in ϕ if no free x leaf in ϕ occurs in the scope of $\forall y$ or $\exists y$ for any variable y occurring in t.

The following quote [15, p. 106 bottom] emphasizes the side conditions:

> It might be helpful to compare 't is free for x in ϕ' with a precondition of calling a procedure for substitution. If you are asked to compute $\phi[t/x]$ in your exercises or exams, then that is what you should do; but any reasonable implementation of substitution used in a theorem prover would have to check whether t is free for x in ϕ and, if not, rename some variables with fresh ones to avoid the undesirable capture of variables.

In our formalization such notions and their complications become easier to explain because all side conditions of the rules are very explicitly stated. We see it as one of the major advantages of presenting this formalization to students.

2.2 Natural Deduction Rules

We now present the natural deduction rules as described in the literature, again using [15]. The first 9 are rules for classical propositional logic and the last 4 are for first-order logic. Intuitionistic logic can be obtained by omitting the rule *PBC* (proof by contradiction, called "Boole" later) and adding the $\bot$-elimination rule (also known as the rule of explosion) [16]. The rules are as follows:

$$\frac{\boxed{\begin{array}{c}\neg\phi\\ \vdots\\ \bot\end{array}}}{\phi}\ PBC \qquad \frac{\phi \qquad \phi \to \psi}{\psi} \to E \qquad \frac{\boxed{\begin{array}{c}\phi\\ \vdots\\ \psi\end{array}}}{\phi \to \psi} \to I$$

$$\frac{\phi \vee \psi \qquad \boxed{\begin{array}{c}\phi\\ \vdots\\ \chi\end{array}} \qquad \boxed{\begin{array}{c}\psi\\ \vdots\\ \chi\end{array}}}{\chi} \vee E \qquad \frac{\phi}{\phi \vee \psi} \vee I_1 \qquad \frac{\psi}{\phi \vee \psi} \vee I_2$$

$$\frac{\phi \wedge \psi}{\phi} \wedge E_1 \qquad \frac{\phi \wedge \psi}{\psi} \wedge E_2 \qquad \frac{\phi \qquad \psi}{\phi \wedge \psi} \wedge I$$

$$\frac{\exists x\, \phi \qquad \boxed{\begin{array}{cc}x_0 & \phi\,[x_0/x]\\ & \vdots\\ & \chi\end{array}}}{\chi} \exists E \qquad \frac{\phi\,[t/x]}{\exists x\, \phi} \exists I$$

$$\frac{\forall x\, \phi}{\phi\,[t/x]} \forall E \qquad \frac{\boxed{\begin{array}{cc}x_0 & \\ & \vdots\\ & \phi\,[x_0/x]\end{array}}}{\forall x\, \phi} \forall I$$

Side conditions to rules for quantifiers:

$\exists E$: x_0 does not occur outside its box (and therefore not in χ).

$\exists I$: t must be free for x in ϕ.

$\forall E$: t must be free for x in ϕ.

$\forall I$: x_0 is a new variable which does not occur outside its box.

In addition there is a special copy rule [15, p. 20]:

> A final rule is required in order to allow us to conclude a box with a formula which has already appeared earlier in the proof. [...] The copy rule entitles us to copy formulas that appeared before, unless they depend on temporary assumptions whose box has already been closed.

The copy rule is not needed in our formalization due to the way it manages a list of assumptions.

As it can be seen, there are no rules for truth, negation or biimplication, but the following equivalences can be used:

$$\begin{array}{rcl}\top & \equiv & \bot \to \bot\\ \neg A & \equiv & A \to \bot\\ A \leftrightarrow B & \equiv & (A \to B) \wedge (B \to A)\end{array}$$

The symbols A and B are arbitrary formulas.

3 Natural Deduction in NaDeA

One of the unique features of NaDeA is that it comes with a formalization in Isabelle of the natural deduction proof system, including a proof in Isabelle of the soundness theorem for the proof system. In this section we present the definitions necessary for expressing the soundness theorem and the proof in Isabelle is presented in section 5.

3.1 Syntax for Terms and Formulas

The terms and formulas of the first-order logic language are defined as the datatypes term and formula (later abbreviated tm and fm, respectively). The type identifier represents predicate and function symbols (later abbreviated id).

```
identifier := string
term := Var nat | Fun identifier [term, ..., term]
formula := Falsity | Pre identifier [term, ..., term] | Imp formula formula |
           Dis formula formula | Con formula formula |
           Exi formula | Uni formula
```

Truth, negation and biimplication are abbreviations. In the syntax of our formalization, we refer to variables by use of the de Bruijn indices. That is, instead of identifying a variable by use of a name, usually x, y, z etc., each variable has an index that determines its scope. The use of de Bruijn indices instead of named variables allows for a simple definition of substitution. Furthermore, it also serves the purpose of teaching the students about de Bruijn indices. Note that we are not advocating that de Bruijn indices replace the standard treatment of variables in general. It arguably makes complex formulas harder to read, but the pedagogical advantage is that the notion of scope is practiced.

3.2 Natural Deduction Rules

Provability in NaDeA is defined inductively as follows (OK p z means that the formula p follows from the list of assumptions z and member p z means that p is a member of the list z):

$$\frac{\text{member p z}}{\text{OK p z}}\ \text{Assume} \qquad \frac{\text{OK Falsity ((Imp p Falsity) \# z)}}{\text{OK p z}}\ \text{Boole}$$

$$\frac{\text{OK (Imp p q) z} \qquad \text{OK p z}}{\text{OK q z}}\ \text{Imp\_E} \qquad \frac{\text{OK q (p \# z)}}{\text{OK (Imp p q) z}}\ \text{Imp\_I}$$

$$\frac{\text{OK (Dis p q) z} \qquad \text{OK r (p \# z)} \qquad \text{OK r (q \# z)}}{\text{OK r z}}\ \text{Dis\_E}$$

$$\frac{\text{OK p z}}{\text{OK (Dis p q) z}}\ \text{Dis\_I1} \qquad \frac{\text{OK q z}}{\text{OK (Dis p q) z}}\ \text{Dis\_I2}$$

$$\frac{\text{OK (Con p q) z}}{\text{OK p z}}\ \text{Con\_E1} \qquad \frac{\text{OK (Con p q) z}}{\text{OK q z}}\ \text{Con\_E2} \qquad \frac{\text{OK p z} \qquad \text{OK q z}}{\text{OK (Con p q) z}}\ \text{Con\_I}$$

$$\frac{\text{OK (Exi p) z} \quad \text{OK q ((sub 0 (Fun c []) p) \# z)} \quad \text{news c (p\#q\#z)}}{\text{OK q z}}\ \text{Exi\_E}$$

$$\frac{\text{OK (sub 0 t p) z}}{\text{OK (Exi p) z}}\ \text{Exi\_I}$$

$$\frac{\text{OK (Uni p) z}}{\text{OK (sub 0 t p) z}}\ \text{Uni\_E} \qquad \frac{\text{OK (sub 0 (Fun c []) p) z} \quad \text{news c (p \# z))}}{\text{OK (Uni p) z}}\ \text{Uni\_I}$$

Instead of writing OK p z we could also use the syntax z ⊢ p, even in Isabelle, but we prefer a more programming-like approach.

The operator # is between the head and the tail of a list. news c l checks if the identifier c does not occur in any of the formulas in the list l and sub n t p returns the formula p where the term t has been substituted for the variable with the de Bruijn index n.

Note that new constants instead of variables not occuring in the assumptions are used in the existential elimination rule and in the universal introduction rule.

In the types we use ⇒ for function spaces. We include the definitions of member, news and sub because they are necessary for the soundness theorem and also for the formalization in section 5:

```
member :: fm ⇒ fm list ⇒ bool

member p [] = False
member p (q # z) = (if p = q then True else member p z)

new_term :: id ⇒ tm ⇒ bool

new_term c (Var n) = True
new_term c (Fun i l) = (if i = c then False else new_list c l)

new_list :: id ⇒ tm list ⇒ bool

new_list c [] = True
new_list c (t # l) = (if new_term c t then new_list c l else False)

new :: id ⇒ fm ⇒ bool

new c Falsity = True
new c (Pre i l) = new_list c l
new c (Imp p q) = (if new c p then new c q else False)
new c (Dis  p q) = (if new c p then new c q else False)
new c (Con p q) = (if new c p then new c q else False)
new c (Exi  p) = new c p
new c (Uni p) = new c p

news :: id ⇒ fm list ⇒ bool

news c [] = True
news c (p # z) = (if new c p then news c z else False)
```

```
inc_term :: tm ⇒ tm

inc_term (Var n) = Var (n + 1)
inc_term (Fun i l) = Fun i (inc_list l)

inc_list :: tm list ⇒ tm list

inc_list [] = []
inc_list (t # l) = inc_term t # inc_list l


sub_term :: nat ⇒ tm ⇒ tm ⇒ tm

sub_term v s (Var n) = (if n < v then Var n else if n = v then s else Var (n − 1))
sub_term v s (Fun i l) = Fun i (sub_list v s l)

sub_list :: nat ⇒ tm ⇒ tm list ⇒ tm list

sub_list v s [] = []
sub_list v s (t # l) = sub_term v s t # sub_list v s l


sub :: nat ⇒ tm ⇒ fm ⇒ fm

sub v s Falsity = Falsity
sub v s (Pre i l) = Pre i (sub_list v s l)
sub v s (Imp p q) = Imp (sub v s p) (sub v s q)
sub v s (Dis  p q) = Dis  (sub v s p) (sub v s q)
sub v s (Con p q) = Con (sub v s p) (sub v s q)
sub v s (Exi  p) = Exi  (sub (v + 1) (inc_term s) p)
sub v s (Uni p) = Uni (sub (v + 1) (inc_term s) p)
```

3.3 Semantics for Terms and Formulas

To give meaning to formulas and to prove NaDeA sound we need a semantics of the first-order logic language. We present the semantics below. e is the environment, i.e. a mapping of variables to elements. f maps function symbols to the maps they represent. These maps are from lists of elements of the universe to elements of the universe. Likewise, g maps predicate symbols to the maps they represent. 'a is a type variable that represents the universe. It can be instantiated with any type. For instance, it can be instantiated with the natural numbers, the real numbers or strings.

```
semantics_term :: (nat ⇒ 'a) ⇒ (id ⇒ 'a list ⇒ 'a) ⇒ tm ⇒ 'a

semantics_term e f (Var n) = e n
semantics_term e f (Fun i l) = f i (semantics_list e f l)

semantics_list :: (nat ⇒ 'a) ⇒ (id ⇒ 'a list ⇒ 'a) ⇒ tm list ⇒ 'a list

semantics_list e f [] = []
semantics_list e f (t # l) = semantics_term e f t # semantics_list e f l

semantics :: (nat ⇒ 'a) ⇒ (id ⇒ 'a list ⇒ 'a) ⇒ (id ⇒ 'a list ⇒ bool) ⇒
                                                                    fm ⇒ bool

semantics e f g Falsity = False
semantics e f g (Pre i l) = g i (semantics_list e f l)
semantics e f g (Imp p q) = (if semantics e f g p then semantics e f g q else True)
semantics e f g (Dis  p q) = (if semantics e f g p then True else semantics e f g q)
semantics e f g (Con p q) = (if semantics e f g p then semantics e f g q else False)
semantics e f g (Exi  p) =
                         (? x. semantics (% n. if n = 0 then x else e (n - 1)) f g p)
semantics e f g (Uni p) =
                         (! x. semantics (% n. if n = 0 then x else e (n - 1)) f g p)
```

Most of the cases of semantics should be self-explanatory, but the Uni case is complicated. The details are not important here, but in the case for Uni it uses the universal quantifier (!) of Isabelle's higher-order logic to consider all values of the universe. It also uses the lambda abstraction operator (%) to keep track of the indices of the variables. Likewise, the case for Exi uses the existential quantifier (?) of Isabelle's higher-order logic.

We have proved soundness of the formalization in Isabelle (shown here as a derived rule):

$$\frac{\mathsf{OK\ p\ []}}{\mathsf{semantics\ e\ f\ g\ p}}\ \text{Soundness}$$

This result makes NaDeA interesting to a broader audience since it gives confidence in the formulas proved using the tool. The proof in Isabelle of the soundness theorem is presented in section 5.

4 Construction of a Proof

We show here how to build and edit proofs in NaDeA. Furthermore, we describe the presentation of proofs in NaDeA.

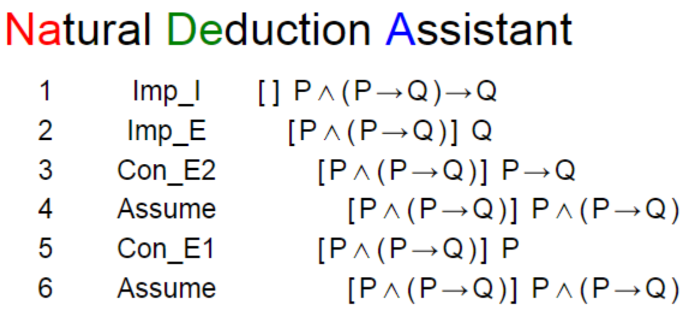
In order to start a proof, you have to start by specifying the goal formula, that is, the formula you wish to prove. To do so, you must enable editing mode by clicking the Edit button in the top menu bar. This will show the underlying proof code and you can build formulas by clicking the red ¤ symbol. Alternatively, you can load a number of tests by clicking the Load button.

At all times, once you have fully specified the conclusion of any given rule, you can continue the proof by selecting the next rule to apply. Again you can do this by clicking the red ¤ symbol. Furthermore, NaDeA allows for undoing and redoing editing steps with no limits.

All proofs are conducted in backward-chaining mode. That is, you must start by specifying the formula that you wish to prove. You then apply the rules inductively until you reach a proof — if you can find one. The proof is finished by automatic application of the Assume rule once the conclusion of a rule is found in the list of assumptions.

To start over on a new proof, you can load the blank proof by using the Load button, or you can refresh the page.

In NaDeA we present any given natural deduction proof (or an attempt at one) in two different types of syntax. One syntax follows the rules as defined in section 3 and is closely related to the formalization in Isabelle, but with a simplified syntax that makes it suitable for teaching purposes. The proof is not built as most often seen in the literature about natural deduction. Usually, for each rule the premises are placed above its conclusion separated by a line. We instead follow the procedure of placing each premise of the rule on separate lines below its conclusion with an additional level of indentation. Here is a screenshot followed by the proof tree:



Natural Deduction Assistant

```
1    Imp_I     [] P∧(P→Q)→Q
2    Imp_E       [P∧(P→Q)] Q
3    Con_E2        [P∧(P→Q)] P→Q
4    Assume          [P∧(P→Q)] P∧(P→Q)
5    Con_E1        [P∧(P→Q)] P
6    Assume          [P∧(P→Q)] P∧(P→Q)
```

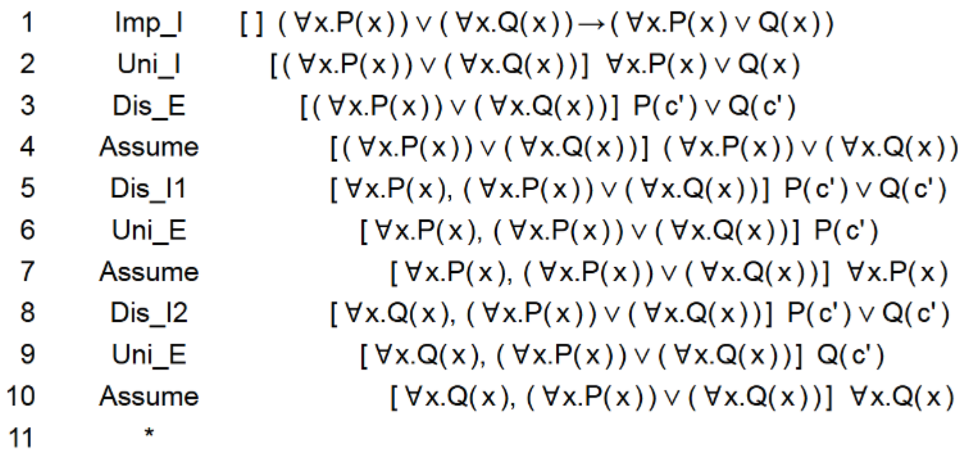
$$\dfrac{\dfrac{\dfrac{\dfrac{}{p \land (p \to q)}\,{\scriptstyle (1)}}{p \to q} \qquad \dfrac{\dfrac{}{p \land (p \to q)}\,{\scriptstyle (1)}}{p}}{q}}{p \land (p \to q) \to q}\,{\scriptstyle (1)}$$

The above proof can also be written in terms of the OK syntax as follows:

```
1   OK (Imp (Con (Pre "P" []) (Imp (Pre "P" []) (Pre "Q" []))) (Pre "Q" [])) []   Imp_I
2     OK (Pre "Q" []) [(Con (Pre "P" []) (Imp (Pre "P" []) (Pre "Q" [])))]         Imp_E
3       OK (Imp (Pre "P" []) (Pre "Q" []))
        [(Con (Pre "P" []) (Imp (Pre "P" []) (Pre "Q" [])))]                       Con_E2
4         OK (Con (Pre "P" []) (Imp (Pre "P" []) (Pre "Q" [])))
        [(Con (Pre "P" []) (Imp (Pre "P" []) (Pre "Q" [])))]                       Assume
5       OK (Pre "P" []) [Con (Pre "P" []) (Imp (Pre "P" []) (Pre "Q" []))]         Con_E1
6         OK (Con (Pre "P" []) (Imp (Pre "P" []) (Pre "Q" [])))
        [(Con (Pre "P" []) (Imp (Pre "P" []) (Pre "Q" [])))]                       Assume
```

So in a way we have the two presentation styles. However, the standard form displayed in the screenshot is always presented and the programming style with the OK syntax is switched on and off with a single click in the browser. The programming style is mandatory when a formula must be entered. We find that the students in general prefer the standard form but also that the switch to the programming style when necessary is rather unproblematic.

For a small but quite interesting example of a proof of a first-order formula consider the following screenshot:



```
1    Imp_I    [] (∀x.P(x)) ∨ (∀x.Q(x)) → (∀x.P(x) ∨ Q(x))
2    Uni_I      [(∀x.P(x)) ∨ (∀x.Q(x))]  ∀x.P(x) ∨ Q(x)
3    Dis_E        [(∀x.P(x)) ∨ (∀x.Q(x))]  P(c') ∨ Q(c')
4    Assume         [(∀x.P(x)) ∨ (∀x.Q(x))]  (∀x.P(x)) ∨ (∀x.Q(x))
5    Dis_I1         [∀x.P(x), (∀x.P(x)) ∨ (∀x.Q(x))]  P(c') ∨ Q(c')
6    Uni_E            [∀x.P(x), (∀x.P(x)) ∨ (∀x.Q(x))]  P(c')
7    Assume             [∀x.P(x), (∀x.P(x)) ∨ (∀x.Q(x))]  ∀x.P(x)
8    Dis_I2         [∀x.Q(x), (∀x.P(x)) ∨ (∀x.Q(x))]  P(c') ∨ Q(c')
9    Uni_E            [∀x.Q(x), (∀x.P(x)) ∨ (∀x.Q(x))]  Q(c')
10   Assume             [∀x.Q(x), (∀x.P(x)) ∨ (∀x.Q(x))]  ∀x.Q(x)
11     *
```

The line with the * in the proof is for the side condition that requires that the constant c' is new. By clicking on the proof the check is displayed in the OK syntax as follows:

```
news (Fun "c*" []) [(Uni (Dis (Pre "P" [Var 0]) (Pre "Q" [Var 0]))),
                    (Dis (Uni (Pre "P" [Var 0])) (Uni (Pre "Q" [Var 0])))]
```

The constant c' is written as "c*" here.

5 Formalization in Isabelle

Formalizations in Isabelle are written in a language that combines functional programming and logic. Our computer science bachelor students know programming from an introductory programming course and are introduced to logic in our course. This makes Isabelle a well suited way to present a sound proof system compared to a more abstract and mathematical approach. Furthermore, the language used in Isabelle is somewhat close to English, which also aids the intuitions of the students. Isabelle also allows the students to interactively inspect the different states of the proof and get an overview of the lemmas and theorems that are used in the steps – all in one screen. In this section we present the soundness proof using our formalization and show the concepts known from programming and logic.

5.1 An Overview

We first give an overview of the formalization in Isabelle. In the overview we see a number of datatypes *tm* and *fm*, that represent the objects that we want to reason about. We also see a primitive recursive function *member* which is used in the inductive definition *OK* of the proof system. Lastly, we see the *soundness* theorem of the proof system. We will explain these concepts as well as show and elaborate on the parts of the formalization that we did not put in the overview.

theory *NaDeA* ***imports*** *Main* ***begin***

type_synonym *id = "char list"*

```
datatype tm = Var nat | Fun id "tm list"

datatype fm = Falsity | Pre id "tm list" | Imp fm fm | Dis fm fm | Con fm fm |
              Exi fm | Uni fm

primrec
  member :: "fm ⇒ fm list ⇒ bool"
where
  "member p [] = False" |
  "member p (q # z) = (if p = q then True else member p z)"

(* More primitive recursive functions as included in the previous sections *)

inductive
  OK :: "fm ⇒ fm list ⇒ bool"
where
Assume:
        "member p z ⟹ OK p z" |
Boole:
        "OK Falsity ((Imp p Falsity) # z) ⟹ OK p z" |
Imp_E:
        "OK (Imp p q) z ⟹ OK p z ⟹ OK q z" |
Imp_I:
        "OK q (p # z) ⟹ OK (Imp p q) z" |

(* More rules as included in the previous sections *)

(* A proof of soundness' is included in the following sections *)

theorem soundness: "OK p [] ⟹ semantics e f g p"
  proof (simp add: soundness') qed

end
```

5.2 Terms and Formulas

Terms are defined by a datatype *tm*. Datatypes are a well-known concept from functional programming. A term is either a variable or a function application. Therefore, we have a constructor *Var* which constructs a variable from a *nat* representing its de Bruijn index. Likewise, we have a constructor *Fun* which constructs a function application from an *id* which is its function identifier and a *"tm list"* which represents its subterms.

When we introduce a datatype in Isabelle, we implicitly state that all terms can be constructed from its constructors. We also implicitly state that if two terms are equal then they must have been constructed from the same constructor and arguments. [18]

Formulas are also formalized as a datatype *fm*. It has a constructor for each operator and quantifier of our first-order logic.

5.3 Membership and Other Primitive Recursive Functions

List membership is defined as a primitive recursive function *member* over lists. The constructor for lists is # which separates the head of the list from the tail. The *member* function is primitive recursive because it removes a constructor from one of its arguments in every recursive call

[2]. In Isabelle, primitive recursive functions are defined in much the same way as in functional programming, namely by stating cases for the different constructors.

The intuition of the function is that *member p z* returns true if the formula *p* is found in the list of formulas *z* and false otherwise. The function considers two cases: either the list is empty or it has a head and a tail. In the first case it is clear that the formula is not a member of the list. In the second case, we use the pattern *(q # z)* where *q* is the head of the list and *z* is the tail. If the head is equal to *p* it is true that *p* is a member of the list. Otherwise, we continue by looking in the tail of the list.

Other primitive recursive functions used in the theory are *semantics_term*, *semantics_list*, *semantics*, *new*, *news*, *inc_term*, *inc_list*, *sub_term*, *sub_list* and *sub*. These functions define the semantics, increasing the de Bruijn indices of a term, a constant being new to an expression, and substitution.

5.4 Proof System

Our proof system is defined by an inductive predicate. Each of the rules of the system is a case in the inductive predicate. For instance, consider the following rule:

Assume: "member p z $\Longrightarrow$ OK p z"

The rule means that *OK p z* follows from *member p z*. Another case is the more complex rule:

Imp_E: "OK (Imp p q) z $\Longrightarrow$ OK p z $\Longrightarrow$ OK q z"

It states that *OK q z* follows from *OK (Imp p q) z* and *OK p z*. This corresponds to the usual notation for inference rules:

$$\frac{\mathsf{OK\ (Imp\ p\ q)\ z} \qquad \mathsf{OK\ p\ z}}{\mathsf{OK\ q\ z}}\ \mathsf{Imp\_E}$$

That a predicate is inductive means that it holds exactly when it can be derived using the given cases.

5.5 Proof of Soundness

We are now ready for the proof of soundness.

fun
  put :: "(nat $\Rightarrow$ 'a) $\Rightarrow$ nat $\Rightarrow$ 'a $\Rightarrow$ nat $\Rightarrow$ 'a"
where
  "put e v x = (λn. ***if*** *n < v* ***then*** *e n* ***else if*** *n = v* ***then*** *x* ***else*** *e (n − 1))"*

The function *put* updates an environment by mapping variable *v* to value *x*. This is used in the definition of the quantifiers, but always for the outermost bound variable. Existing variables greater than *v* are pushed one position up, i.e. variable *i* now points to the value of variable *i − 1* in the old environment.

We use ***fun*** to declare many different functions without being restricted to the primitive recursive form. The operator λ is for lambda abstraction applied to occurrences of the parameter value and is known from functional programming. More informally, if *E* is some expression in Isabelle then $\lambda x.\ E\ x$ is the function that takes an input, for instance *y*, and returns *E y*.

lemma *"put e 0 x = (λn.* ***if*** *n = 0* ***then*** *x* ***else*** *e (n − 1))"* ***proof*** *simp* ***qed***

This lemma shows that *put* is a generalization of the expression

$$\lambda n.\ \textbf{\textit{if}}\ n = 0\ \textbf{\textit{then}}\ x\ \textbf{\textit{else}}\ e\ (n - 1)$$

which appears in the semantics. We use this generalization to prove properties of putting that we use in our soundness proof. The lemma is followed by a proof. In this case, the proof is performed automatically by the simplifier *simp*. The beginning of the proof is marked by ***proof*** and the end is marked by ***qed***. The proof method *simp* works by applying simplification rules [18]. It contains rules that are generated from definitions of functions, datatypes, etc., in addition to simplification rules from the Isabelle library.

```
lemma increment:
  "semantics_term (put e 0 x) f (inc_term t) = semantics_term e f t"
  "semantics_list (put e 0 x) f (inc_list l) = semantics_list e f l"
proof (induct t and l rule: semantics_term.induct semantics_list.induct)
qed simp_all
```

The lemma *increment* shows that we preserve the semantics of a term when we increment its de Bruijn indices while putting a value x at index 0. The reason is that putting pushes the values one index up in the environment. The proof is by induction on t and l, which is stated as

induct t ***and*** *l rule: semantics_term.induct semantics_list.induct*

and it generates four proof goals; one for each of the cases in *semantics_term* and *semantics_list*. These goals can be inspected in the Isabelle editor by placing the cursor right after

(induct t ***and*** *l rule: semantics_term.induct semantics_list.induct)*

and looking in the so-called state panel. The proof method *simp_all* applies the simplifier to all available proof goals [18]. We place *simp_all* after ***qed*** in order to finish the proof and to allow inspection of the proof state interactively in Isabelle.

```
lemma commute: "put (put e v x) 0 y = put (put e 0 y) (v + 1) x"
proof force qed
```

The lemma *commute* shows that the function *put* commutes. More precisely, we want to put a value at position $v + 1$ in the environment and one at position 0, and the theorem shows that the order in which we do this does not matter, as long as we are careful with the indices.

The proof is automatic and uses the proof method *force*, which works by simplification and classical reasoning [2].

```
fun
  all :: "(fm ⇒ bool) ⇒ fm list ⇒ bool"
where
  "all b z = (∀p. if member p z then b p else True)"
```

The function *all* checks if the predicate b is true for all formulas in a list. The $\forall$ operator is for universal quantification.

```
lemma allhead: "all b (p # z) ⟹ b p" proof simp qed
```

```
lemma alltail: "all b (p # z) ⟹ all b z" proof simp qed
```

```
lemma allnew: "all (new c) z = news c z"
proof (induct z) qed (simp, simp, metis)
```

The lemma *allhead* states that if *b* holds for the entire list, then it holds for the head of the list in particular. The lemma *alltail* is similar, but for the tail of the list. Finally, the lemma *allnew* shows the equivalence between *news* and *all* combined with *new*. The proof uses the proof methods *simp* and *metis* in the order they are written, i.e. *simp* the first proof goal generated by the structural induction on *z*. Then *simp* simplifies the second proof goal which is afterwards proved by *metis*. The *metis* proof method is a resolution theorem prover [17].

lemma *map':*
  "new_term c t $\Longrightarrow$ semantics_term e (f(c := m)) t = semantics_term e f t"
  "new_list c l $\Longrightarrow$ semantics_list e (f(c := m)) l = semantics_list e f l"
proof *(induct t* ***and*** *l rule: semantics_term.induct semantics_list.induct)*
qed *(simp, simp, metis, simp, simp, metis)*

lemma *map:*
  "new c p $\Longrightarrow$ semantics e (f(c := m)) g p = semantics e f g p"
proof *(induct p arbitrary: e)*
qed *(simp, simp, metis map'(2), simp, metis, simp, metis, simp, metis, simp_all)*

lemma *allmap:*
  "news c z $\Longrightarrow$ all (semantics e (f(c := m)) g) z = all (semantics e f g) z"
proof *(induct z)* ***qed*** *(simp, simp, metis map)*

The lemma *map'* shows that we preserve the semantics of a term if we map a constant that is new to the term to another value. Here, *f(c := m)* maps function identifier *c* to *m* in the function map *f*. Because the lemma is quite obvious it can be proved automatically. The first and third goals are proved by *simp*, and the second and fourth are simplified by *simp* and then proved by *metis*. The lemma *map* shows that the property of *map'* can be extended to also hold for formulas. This can also be proved automatically. There are seven proof goals of the induction corresponding to each of the formula constructors. We use *simp* to discharge of the first proof goal, then *simp* followed by *metis* for the next four. This time we use *metis map'(2)* to prove the case for predicates. This works by applying *metis* with the addition of the second part of *map'* as a fact with which it can reason. The last two proof goals are proved with the simplifier using *simp_all*. The lemma *allmap* further extends the property of the lemma *map'* to also hold for lists of formulas. We prove it using *simp* and *metis map*.

lemma *substitute':*
  "semantics_term e f (sub_term v s t) =
    semantics_term (put e v (semantics_term e f s)) f t"
  "semantics_list e f (sub_list v s l) =
    semantics_list (put e v (semantics_term e f s)) f l"
proof *(induct t* ***and*** *l rule: semantics_term.induct semantics_list.induct)*
qed *simp_all*

The lemma *substitute'* is the famous substitution lemma for terms. This lemma shows a relation between the world of syntax and the world of semantics. More specifically, the relation is between the syntactical operation of substitution and the semantic notion of variable environments. The two are related because a substitution instantiates a variable with a term, and this term represents a value. Thus we get the same semantics of the term if we instead of substitution put the value directly at the index of the variable in the environment. The proof is by induction and *simp_all*.

lemma *substitute:*
  "semantics e f g (sub v t p) = semantics (put e v (semantics_term e f t)) f g p"

```
proof (induct p arbitrary: e v t)
  fix i l e v t
  show "semantics e f g (sub v t (Pre i l)) =
      semantics (put e v (semantics_term e f t)) f g (Pre i l)"
  proof (simp add: substitute'(2)) qed
next
  fix p e v t assume ∗: "semantics e' f g (sub v' t' p) =
      semantics (put e' v' (semantics_term e' f t')) f g p" for e' v' t'
  have "semantics e f g (sub v t (Exi p)) =
      (∃x. semantics (put (put e 0 x) (v + 1)
          (semantics_term (put e 0 x) f (inc_term t))) f g p)"
    using ∗ proof simp qed
  also have "... =
      (∃x. semantics (put (put e v (semantics_term e f t)) 0 x) f g p)"
    using commute increment(1) proof metis qed
  finally show "semantics e f g (sub v t (Exi p)) =
      semantics (put e v (semantics_term e f t)) f g (Exi p)" proof simp qed
  have "semantics e f g (sub v t (Uni p)) =
      (∀x. semantics (put (put e 0 x) (v + 1)
          (semantics_term (put e 0 x) f (inc_term t))) f g p)"
    using ∗ proof simp qed
  also have "... =
      (∀x. semantics (put (put e v (semantics_term e f t)) 0 x) f g p)"
    using commute increment(1) proof metis qed
  finally show "semantics e f g (sub v t (Uni p)) =
      semantics (put e v (semantics_term e f t)) f g (Uni p)" proof simp qed
qed simp_all
```

The lemma *substitute* extends the substitution lemma to hold also for formulas. The proof is by induction on a formula p. In the proof we write *arbitrary: e v t* because then e, v and t are also arbitrary in the induction hypothesis. This more general induction hypothesis is necessary for the proof. Most cases can be proven by the simplifier without any instructions, but we prove the cases for predicates *Pre*, existential quantification *Exi* and universal quantification *Uni* more explicitly. For the predicates, we only need instruct the simplifier to use *substitute'(2)* as a simplification rule by writing *(simp add: substitute'(2))*. For the existential quantification we make an explicit proof. We fix the subformula p of an existential quantification for which we want to prove the property. As said, we want to prove it with an arbitrary variable environment e, an arbitrary variable v, and an arbitrary term t so we fix those as well. We then state the induction hypothesis $*$ which says that for the subformula p of our existential quantification we can put the value of the term t in the environment instead of doing substitution with t:

```
assume ∗: "semantics e' f g (sub v' t' p) =
    semantics (put e' v' (semantics_term e' f t')) f g p" for e' v' t'
```

The ***for*** keyword ensures that *e'*, *v'*, and *t'* are arbitrary as we wished. We wish to prove the substitution lemma for the existential quantification *Exi p*, i.e. that

```
semantics e f g (sub v t (Exi p)) =
    semantics (put e v (semantics_term e f t)) f g (Exi p)
```

The keyword ***also*** together with ***finally*** is used to make a proof from left to right of the equality of two expressions. This is what we want to do, and thus we start from the left-hand side:

semantics e f g (sub v t (Exi p))

and realize that by the definition of substitution and the semantics of *Exi* we just need a single value x for which the semantics of *sub (v + 1) (inc_ term t) p* is true under the environment *put e 0 x*. At the same time, we realize that we can now use the induction hypothesis. Therefore, instead of considering the semantics of *sub (v + 1) (inc_ term t) p* under *put e 0 x*, we equivalently consider the semantics of p under the variable environment which is *put e 0 x* with the value of t put on index $v + 1$. We must thus continue our proof from

```
(∃x. semantics (put (put e 0 x) (v + 1)
       (semantics_ term (put e 0 x) f (inc_ term t))) f g p)
```

We can make this expression much simpler by using *commute* and *increment(1)*.

(∃x. semantics (put (put e v (semantics_ term e f t)) 0 x) f g p)

We finish our proof using the semantics of *Exi*, as well as the fact that *put* generalizes putting at index 0, and we get the right-hand side we were looking for:

semantics (put e v (semantics_ term e f t)) f g (Exi p)

Then follows a proof of substitution for the universal quantification *Uni* since it has the same induction hypothesis. The proof is very similar. Finally we write ***qed*** *simp_ all* to prove the remaining cases by simplification.

```
lemma soundness': "OK p z ⟹ all (semantics e f g) z ⟹ semantics e f g p"
proof (induct arbitrary: f rule: OK.induct)
  fix f p z assume "all (semantics e f g) z"
      "all (semantics e f' g) (Imp p Falsity # z) ⟹
          semantics e f' g Falsity" for f'
  then show "semantics e f g p" proof force qed
next
  fix f p q z r assume "all (semantics e f g) z"
      "all (semantics e f' g) z ⟹ semantics e f' g (Dis p q)"
      "all (semantics e f' g) (p # z) ⟹ semantics e f' g r"
      "all (semantics e f' g) (q # z) ⟹ semantics e f' g r" for f'
  then show "semantics e f g r" proof (simp, metis) qed
next
  fix f p q z assume "all (semantics e f g) z"
      "all (semantics e f' g) z ⟹ semantics e f' g (Con p q)" for f'
  then show "semantics e f g p" "semantics e f g q"
  proof (simp, metis, simp, metis) qed
next
  fix f p z q c assume *: "all (semantics e f g) z"
      "all (semantics e f' g) z ⟹ semantics e f' g (Exi p)"
      "all (semantics e f' g) (sub 0 (Fun c []) p # z) ⟹ semantics e f' g q"
      "news c (p # q # z)" for f'
  obtain x where "semantics (λn. if n = 0 then x else e (n − 1)) f g p"
    using *(1) *(2) proof force qed
  then have "semantics (put e 0 x) f g p" proof simp qed
  then have "semantics (put e 0 x) (f(c := λw. x)) g p"
    using *(4) allhead allnew map proof blast qed
  then have "semantics e (f(c := λw. x)) g (sub 0 (Fun c []) p)"
    proof (simp add: substitute) qed
  moreover have "all (semantics e (f(c := λw. x)) g) z"
    using *(1) *(4) alltail allnew allmap proof blast qed
  ultimately have "semantics e (f(c := λw. x)) g q" using *(3) proof simp qed
  then show "semantics e f g q" using *(4) allhead alltail allnew map
  proof blast qed
next
  fix f z t p assume "all (semantics e f g) z"
      "all (semantics e f' g) z ⟹ semantics e f' g (sub 0 t p)" for f'
  then have "semantics (put e 0 (semantics_term e f t)) f g p"
  proof (simp add: substitute) qed
  then show "semantics e f g (Exi p)" proof (simp, metis) qed
next
  fix f z t p assume "all (semantics e f g) z"
      "all (semantics e f' g) z ⟹ semantics e f' g (Uni p)" for f'
  then show "semantics e f g (sub 0 t p)" proof (simp add: substitute) qed
next
  fix f c p z assume *: "all (semantics e f g) z"
      "all (semantics e f' g) z ⟹ semantics e f' g (sub 0 (Fun c []) p)"
      "news c (p # z)" for f'
  have "semantics (λn. if n = 0 then x else e (n − 1)) f g p" for x
  proof −
    have "all (semantics e (f(c := λw. x)) g) z"
      using *(1) *(3) alltail allnew allmap proof blast qed
    then have "semantics e (f(c := λw. x)) g (sub 0 (Fun c []) p)"
      using *(2) proof simp qed
    then have "semantics (λn. if n = 0 then x else e (n − 1))
        (f(c := λw. x)) g p"
      proof (simp add: substitute) qed
    then show "semantics (λn. if n = 0 then x else e (n − 1)) f g p"
      using *(3) allhead alltail allnew map proof blast qed
  qed
  then show "semantics e f g (Uni p)" proof simp qed
qed simp_all
```

The lemma *soundness'* shows the soundness of the proof system. It is done by rule induction on the rules of the proof system. We have to prove that assuming that the derivations in the premises follow logically, then so does the derivation in the conclusion. For the rules *Boole*, *Dis_E*, *Con_E1*, *Con_E2* and *Uni_E* we state the induction hypothesis, and the assumption that the premises are satisfied. We then do the proof by automation. For *Uni_I*, *Exi_E* and *Exi_I* we write out the proofs explicitly because they are more complicated. We prove the remaining rules sound by automation with the substitution lemma as simplification rule. The keyword ***next*** is used to separate the different cases.

Let us look at how we proved *Uni_I* sound. The $*$ states our induction hypothesis which states that if our assumptions z are satisfied by any function map then so is p with a constant

Fun c [] substituted for 0.

$$\textit{all (semantics e f' g) z} \Longrightarrow \textit{semantics e f' g (sub 0 (Fun c []) p)}$$

We additionally assume that the side condition that c is new to $p\#z$.

$$\textit{news c (p \# z)}$$

Since we want to prove the derivation from z to *Uni p* sound we also assume that the premises z are satisfied by a fixed *f* and a fixed g.

$$\textit{all (semantics e f g) z}$$

We then wish to prove that so is *Uni p*. Since the premises are satisfied by *f* and since c is new to them they must also be satisfied by *f(c := λw. x)*.

$$\textit{all (semantics e (f(c := }\lambda w.\textit{ x)) g) z}$$

In this step we used the proof method *blast* which is a tableau prover [17]. Then it follows by our induction hypothesis that also p with c substituted for 0 is satisfied.

$$\textit{semantics e (f(c := }\lambda w.\textit{ x)) g (sub 0 (Fun c []) p)}$$

We then use the substitution lemma to add the value of t to the environment instead of doing the substitution.

$$\textit{semantics } (\lambda n.\ \textbf{\textit{if}}\ n = 0\ \textbf{\textit{then}}\ x\ \textbf{\textit{else}}\ e\ (n - 1))\ (f(c := \lambda w.\ x))\ g\ p$$

Since c is new to p we might as well evaluate it in *f* instead of *f(c := λw. x)* and this concludes the proof.

$$\textit{semantics } (\lambda n.\ \textbf{\textit{if}}\ n = 0\ \textbf{\textit{then}}\ x\ \textbf{\textit{else}}\ e\ (n - 1))\ f\ g\ p$$

5.6 A Consistency Corollary to the Soundness Theorem

Soundness is the main theorem about the formalization of the natural deduction proof system. As a corollary we immediately prove the following consistency result about the proof system:

Something, but not everything, can be proved.

In Isabelle we can prove it using the simplifier (*simp*), some simple rules and Isabelle's prover for intuitionistic logic (*iprover*), although a classical prover (say, *metis*) would work too, of course:

```
corollary "∃p. OK p []" "∃p. ¬ OK p []"
proof –
  have "OK (Imp p p) []" for p proof (rule Imp_I, rule Assume, simp) qed
  then show "∃p. OK p []" proof iprover qed
  have "¬ semantics (e :: nat ⇒ unit) f g Falsity" for e f g proof simp qed
  then show "∃p. ¬ OK p []" using soundness proof iprover qed
qed
```

Recall that ∃ is the existential quantifier in Isabelle. The symbol ¬ is negation in Isabelle. The first part (∃*p. OK p []*) follows from a simple proof of $p \to p$ (for an arbitrary formula p in first-order logic). The second part (∃*p.* ¬ *OK p []*) follows from the proof of soundness and from the fact that the semantics of *Falsity* is always false (for simplicity we consider universes with just one element, provided by the *unit* type).

5.7 Style of the Proof

When you do a proof in Isabelle, you need to choose how close you want the steps of the proof to be to each other. On one hand the proof should be understandable, but on the other hand you do not want the readers to get lost in small details. Larger steps also allow the reader to think for himself instead of having everything spelled out in detail. If a student wants to gain more insight, she can expand it, and let Isabelle check if the details she added were correct. Isabelle also has tools that allow its users to see which steps *simp* used to prove a result.

The notation we chose to use is close to that of programming rather than that of mathematics and set theory. Isabelle, however, also supports a more classical notation. Our motivation for the choice is our students' background from programming, as well as to show that a very well-defined structure lies beneath the logical symbols both at the object and the meta levels.

We use the formal semantics and soundness proof in our teaching. Among other things the students can make calculation using the formal semantics in Isabelle and also make changes to the formal semantics (for example, replacing the if-then-else with logical operators in Isabelle, or adding negation to the logic).

6 Related Work

Formalizations of model theory and proof theory of first-order logic are rare, for example [6, 7, 11, 20, 21].

Throughout the development of NaDeA we have considered some of the natural deduction assistants currently available. Several of the tools available share some common flaws. They can be hard to get started with, or depend on a specific platform. However, there are also many tools that each bring something useful and unique to the table. One of the most prominent is PANDA, described in [13]. PANDA includes a lot of graphical features that make it fast for the experienced user to conduct proofs, and it helps the beginners to tread safely. Another characteristic of PANDA is the possibility to edit proofs partially before combining them into a whole. It definitely serves well to reduce the confusion and complexity involved in conducting large proofs. However, we still believe that the way of presenting the proof can be more explicit. In NaDeA, every detail is clearly stated as part of the proof code. In that sense, the students should become more aware of the side conditions to rules and how they work.

Another tool that deserves mention is ProofWeb [10] which is open source software for teaching natural deduction. It provides interaction between some proof assistants (Coq, Isabelle, Lego) and a web interface. The tool is highly advanced in its features and uses its own syntax. Also, it gives the user the possibility to display the proof in different formats. However, the advanced features come at the cost of being very complex for bachelor students and require that you learn a new syntax. It serves as a great tool for anyone familiar with natural deduction that wants to conduct complex proofs that can be verified by the system. It may, on the other hand, prove less useful for teaching natural deduction to beginners since there is no easy way to get started. In NaDeA, you are free to apply any (applicable) rule to a given formula, and thus, beginners have the freedom to play around with the proof system in a safe way. Furthermore, the formalized soundness result for the proof system of NaDeA makes it relevant for a broader audience, since this gives confidence in that the formulas proved with the tool are actually valid.

7 Further Work

In NaDeA there is support for proofs in propositional logic as well as first-order logic. We would also like to extend to more complex logic languages, the most natural step being higher-order logic. This could be achieved using the CakeML approach [8].

Other branches of logic would also be interesting. Apart from just extending the natural deduction proof system to support other branches of logic, another option is to implement other proof systems as well.

Because the NaDeA tool has a formalization in Isabelle of its proof system, we would like to provide features that allow for a more direct integration with Isabelle. For instance, we would like to allow for proofs to be exported to a format suitable for Isabelle such that Isabelle could verify the correctness of the proofs. A formal verification of the implementation would require much effort, but perhaps it could be reimplemented on top of Isabelle (although probably not in TypeScript / JavaScript) or using Isabelle's code generation facility.

We would like to extend NaDeA with more features in order to help the user in conducting proofs and in understanding logic. For example, the tool could be extended with step-by-step execution of the auxiliary primitive recursive functions used in the side conditions of the natural deduction rules.

NaDeA has been successfully classroom tested in a regular course with around 70 bachelor students in computer science each year. The students find the formal semantics and the proof of the soundness theorem relevant and instructive. We have extended NaDeA with a so-called ProofJudge system [19] which allows students to submit solutions and get feedback. We are in the process of adding to NaDeA a simple automated theorem prover [20, 21], verified by the Isabelle proof assistant and developed using Isabelle's code generation facility, in order to make it possible to better guide the students if for example sub-proofs are started and there is in fact no possible proof.

References

[1] Mordechai Ben-Ari. Mathematical Logic for Computer Science. Third Edition. Springer 2012.

[2] Tobias Nipkow, Lawrence C. Paulson and Markus Wenzel. Isabelle/HOL - A Proof Assistant for Higher-Order Logic. Lecture Notes in Computer Science 2283, Springer 2002.

[3] Dag Prawitz. Natural Deduction. A Proof-Theoretic Study. Stockholm: Almqvist & Wiksell 1965.

[4] Francis Jeffry Pelletier. A Brief History of Natural Deduction. History and Philosophy of Logic, 1-31, 1999.

[5] Melvin Fitting. First-Order Logic and Automated Theorem Proving. Second Edition Springer 1996.

[6] John Harrison. Formalizing Basic First Order Model Theory. Lecture Notes in Computer Science 1497, 153–170, Springer 1998.

[7] Stefan Berghofer. First-Order Logic According to Fitting. Formal Proof Development. Archive of Formal Proofs 2007.

[8] Ramana Kumar, Rob Arthan, Magnus O. Myreen and Scott Owens. HOL with Definitions: Semantics, Soundness, and a Verified Implementation. Lecture Notes in Computer Science 8558, 308–324, Springer 2014.

[9] Jørgen Villadsen, Anders Schlichtkrull and Andreas Viktor Hess. Meta-Logical Reasoning in Higher-Order Logic. Accepted at 29th International Symposium Logica, Hejnice Monastery, Czech Republic, 15-19 June 2015.

[10] ProofWeb. Online `http://proofweb.cs.ru.nl/login.php` (ProofWeb is both a system for teaching logic and for using proof assistants through the web). Accessed September 2016.

[11] Jasmin Christian Blanchette, Andrei Popescu and Dmitriy Traytel. Unified Classical Logic Completeness - A Coinductive Pearl. Lecture Notes in Computer Science 8562, 46–60, 2014.

[12] Krysia Broda, Jiefei Ma, Gabrielle Sinnadurai and Alexander Summers. Pandora: A Reasoning Toolbox Using Natural Deduction Style. Logic Journal of IGPL, 15(4):293–304, 2007.

[13] Olivier Gasquet, François Schwarzentruber and Martin Strecker. Panda: A Proof Assistant in Natural Deduction for All. A Gentzen Style Proof Assistant for Undergraduate Students. Lecture Notes in Computer Science 6680, 85–92. Springer 2011.

[14] Antonia Huertas. Ten Years of Computer-Based Tutors for Teaching Logic 2000-2010: Lessons learned. Lecture Notes in Computer Science 6680, 131–140, Springer 2011.

[15] Michael Huth and Mark Ryan. Logic in Computer Science: Modelling and Reasoning About Systems. Second Edition. Cambridge University Press 2004.

[16] Jonathan P. Seldin. Normalization and Excluded Middle. I. Studia Logica, 48(2):193–217, 1989.

[17] Jasmin Christian Blanchette, Lukas Bulwahn and Tobias Nipkow. Automatic Proof and Disproof in Isabelle/HOL. Lecture Notes in Computer Science 6989, 12-27, 2011.

[18] Tobias Nipkow and Gerwin Klein. Concrete Semantics with Isabelle/HOL. Springer 2014.

[19] Jørgen Villadsen. ProofJudge: Automated Proof Judging Tool for Learning Mathematical Logic. Exploring Teaching for Active Learning in Engineering Education Conference (ETALEE), Copenhagen, Denmark, 2015.

[20] Jørgen Villadsen, Anders Schlichtkrull and Andreas Halkjær From. Code Generation for a Simple First-Order Prover. Isabelle Workshop, Nancy, France, 2016.

[21] Alexander Birch Jensen, Anders Schlichtkrull and Jørgen Villadsen. Verification of an LCF-Style First-Order Prover with Equality. Isabelle Workshop, Nancy, France, 2016.

Programming and Verifying a Declarative First-Order Prover in Isabelle/HOL

Alexander Birch Jensen, John Bruntse Larsen, Anders Schlichtkrull, and Jørgen Villadsen

DTU Compute, Technical University of Denmark, Kongens Lyngby, Denmark

Abstract

We certify in the proof assistant Isabelle/HOL the soundness of a declarative first-order prover with equality. The LCF-style prover is a translation we have made, to Standard ML, of a prover in John Harrison's *Handbook of Practical Logic and Automated Reasoning.* We certify it by replacing its kernel with a certified version that we program, certify and generate code from; all in Isabelle/HOL. In a declarative proof each step of the proof is declared, similar to the sentences in a thorough paper proof. The prover allows proofs to mix the declarative style with automatic theorem proving by using a tableau prover. Our motivation is teaching how automated and declarative provers work and how they are used. The prover allows studying concrete code and a formal verification of correctness. We show examples of proofs and how they are made in the prover. The entire development runs in Isabelle's ML environment as an interactive application or can be used standalone in OCaml or Standard ML (or in other functional programming languages like Haskell and Scala with some additional work).

Keywords Isabelle, verification, declarative proofs for first-order logic with equality, soundness, LCF-style prover

1 Introduction

There are two styles of writing proofs in provers – the procedural style and the declarative style. In the procedural style, users write a script of instructions that tells the prover how to prove a theorem. Only by executing each instruction can the user see what happens in the proof. In the declarative style, proofs resemble thorough proofs on paper because they are written as a chain of sentences of varying level of detail. Thus, a user can read and understand a declarative proof without executing the prover. The declarative style is supported in advanced proof assistants such as Isabelle/HOL [29].

We develop a declarative prover intended mainly for educational purposes that users can quite easily inspect and for which a formal soundness proof is also accessible in Isabelle/HOL. We do this by translating, to the functional programming language Standard ML (SML), John Harrison's interactive theorem prover for classical first-order logic with equality from his *Handbook of Practical Logic and Automated Reasoning* [13]. The kernel of his prover is based on a proof system that uses equality which is advantageous because it means that it avoids substitution.

$$
\begin{aligned}
&(\forall x.\ P(x) \land (\forall y.\ P(y) \land H(y,x) \longrightarrow G(y)) \longrightarrow G(x)) \land \\
&((\exists x.\ P(x) \land \neg G(x)) \longrightarrow \\
&\quad (\exists x.\ P(x) \land \neg G(x) \land (\forall y.\ P(y) \land \neg G(y) \longrightarrow J(x,y)))) \land \\
&(\forall xy.\ P(x) \land P(y) \land H(x,y) \longrightarrow \neg J(y,x)) \longrightarrow \\
&(\forall x.\ P(x) \longrightarrow G(x))
\end{aligned}
$$

Figure 1: Pelletier's problem 46

1.1 The LCF-style Prover

The main aim of the present work has been to evaluate the prospects of a simple LCF-style prover as a certified declarative first-order prover that is generated from a specification in Isabelle/HOL and that can be inspected for educational purposes.

The prover follows the LCF-style of having a trusted kernel on which other components are built [14]. The main benefit is that if the user trusts the kernel, then she can also trust the other components.

We take advantage of this in our certification. By certifying the soundness of the kernel, we ensure the soundness of all the other components because they rely on the kernel to generate theorems.

We therefore program a kernel similar to Harrison's in Isabelle/HOL and certify its soundness by defining a semantics on the first-order formulas with equality. Then we use Isabelle's code reflection facility to generate a module that represents the kernel and import it into the special Isabelle/ML environment for Standard ML. Hereafter we load the rest of the prover into Isabelle/ML along with a number of examples.

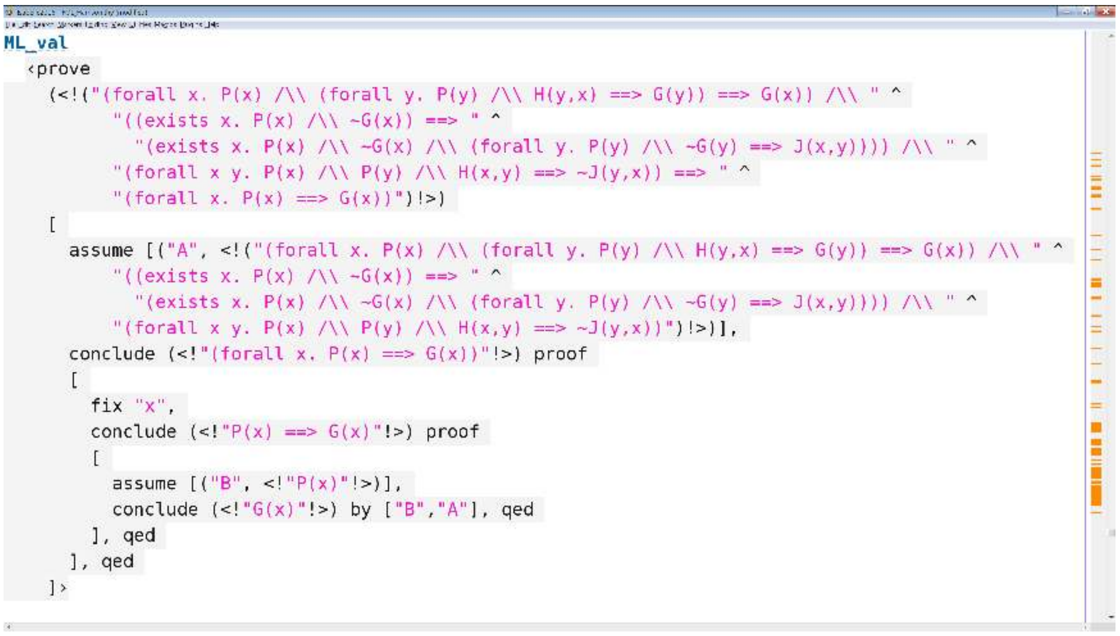
The whole prover, its verification and many examples of proofs are available in the Archive of Formal Proofs (AFP) [19]. The AFP entry is a single theory file which is structured as follows:

- A definition of the syntax of FOL with equality.
- A definition of the axioms and rules of the kernel.
- A definition of a semantics of FOL with equality.
- Definitions of a proof system consisting of the axioms and rules of the kernel.
- A soundness proof of the proof system.
- Code reflection of the axioms and rules.
- The prover that builds on top of the kernel.
- Examples of proofs in the prover.

The prover includes a tableau prover which allows proofs to mix the declarative proof style with automatic theorem proving, and we show several examples of such proofs.

1.2 Declarative Proof of Pelletier's Problem 46

As test cases for our declarative prover we have considered several of the most difficult first-order logic problems in Pelletier's *Seventy-Five Problems for Testing Automatic Theorem Provers* [31].



```
ML_val
  ‹prove
    (<!("(forall x. P(x) /\\ (forall y. P(y) /\\ H(y,x) ==> G(y)) ==> G(x)) /\\ " ^
          "((exists x. P(x) /\\ ~G(x)) ==> " ^
            "(exists x. P(x) /\\ ~G(x) /\\ (forall y. P(y) /\\ ~G(y) ==> J(x,y)))) /\\ " ^
          "(forall x y. P(x) /\\ P(y) /\\ H(x,y) ==> ~J(y,x)) ==> " ^
          "(forall x. P(x) ==> G(x))")!>)
    [
      assume [("A", <!("(forall x. P(x) /\\ (forall y. P(y) /\\ H(y,x) ==> G(y)) ==> G(x)) /\\ " ^
          "((exists x. P(x) /\\ ~G(x)) ==> " ^
            "(exists x. P(x) /\\ ~G(x) /\\ (forall y. P(y) /\\ ~G(y) ==> J(x,y)))) /\\ " ^
          "(forall x y. P(x) /\\ P(y) /\\ H(x,y) ==> ~J(y,x))")!>)],
      conclude (<!"(forall x. P(x) ==> G(x))"!>) proof
      [
        fix "x",
        conclude (<!"P(x) ==> G(x)"!>) proof
        [
          assume [("B", <!"P(x)"!>)],
          conclude (<!"G(x)"!>) by ["B","A"], qed
        ], qed
      ], qed
    ]›
```

Figure 2: Example of a declarative proof in our prover running inside Isabelle/jEdit. The initial string encodes Pelletier's problem 46 and the prover function calls in black (`prove`, `assume`, `conclude`, `proof`, `fix` and `qed`) mark steps in the proof. The prover runs in the Isabelle/ML environment.

As mentioned earlier, the prover allows proofs to mix the declarative style with use of automatic theorem proving. Let us illustrate this by considering Pelletier's problem 46 in Figure 1.

We prove the formula in a declarative style with our prover as shown in Figure 2 including the use of the automatic tableau prover. To explain the intuition of the proof, Figure 3 shows the proof recast in natural language. The proof is structured in the same way as the declarative proof.

The idea of the proof is that we break down the structure of the formula until we are in a state where the automation can take care of the rest.

In the next section we provide a short introduction to Isabelle/HOL and in the subsequent sections we describe first the architecture for the declarative prover and then the formalization in Isabelle/HOL.

Parts of this paper are adapted from our previous workshop paper [17].

2 Isabelle/HOL

Isabelle/HOL is a proof assistant for higher-order logic. Higher-order logic can be thought of as a mix of logic and typed functional programming. Isabelle/HOL includes the usual logical connectives $\longrightarrow$, $\longleftrightarrow$, $\vee$, $\wedge$, $\neg$ as well as equality $=$ and non-equality $\neq$. Additionally Isabelle/HOL allows us to specify rules using $\Longrightarrow$. For instance Isabelle/HOL axiomatizes modus-ponens as

We will prove Pelletier's problem 46 (Figures 1 and 2).

Since its outermost structure is an implication, we start by assuming the three formulas in the conjunction on the left-hand side of the arrow:

- We assume $\forall x.\ P(x) \land (\forall y.\ P(y) \land H(y,x) \longrightarrow G(y)) \longrightarrow G(x)$ and
  $\exists x.\ P(x) \land \neg G(x)) \longrightarrow (\exists x.\ P(x) \land \neg G(x) \land (\forall y.\ P(y) \land \neg G(y) \longrightarrow J(x,y)))$ and
  $\forall xy.\ P(x) \land P(y) \land h(x,y) \longrightarrow \neg J(y,x)$.
  These assumptions are labeled A.
- From this we conclude the right-hand side $\forall x.\ P(x) \longrightarrow G(x)$.
  Since it is a universal quantification we do it as follows:
  - We fix an arbitrary element, x.
  - Then we conclude $P(x) \longrightarrow G(x)$ as follows:
    - We assume $P(x)$ and label the assumption B.
    - Then we conclude $G(x)$ which follows by assumptions A and B.
      This is proved automatically with the tableau prover.

Figure 3: The declarative proof of Pelletier's problem 46 recast as a structured proof in natural language

$(P \longrightarrow Q) \Longrightarrow P \Longrightarrow Q$. It is convention to separate the assumptions from the conclusions of theorems and lemmas using $\Longrightarrow$ even though, at least logically, one might as well use $\longrightarrow$.

Isabelle/HOL also includes commands for defining types, defining functions and declaring theorems. We list these in Table 1.

In making our certified prover, we found the following tools of Isabelle/HOL essential:

- The structured proof language Isabelle/Isar [41], which offers ample features for writing declarative proofs, as well as proof methods such as simp, fastforce and metis, which can discharge proof goals [43].
- Sledgehammer [2], which can discharge proof goals by employing multiple automatic theorem provers (ATPs) as well as satisfiability-modulo-theories (SMT) solvers and proof reconstruction in Isabelle/Isar.
- Isabelle/ML [44], which is a way to use Standard ML (SML) inside the Isabelle environment. It can be embedded in Isabelle/Isar which means that it can be used side by side with Isabelle/HOL.
- Isabelle's code generation [10] and its code reflection is used to generate code from Isabelle/HOL definitions and load it into the Isabelle/ML environment.
- The Isabelle/jEdit Prover IDE (Integrated Development Environment) [42], which allows both for navigating, stating and checking formalizations in Isabelle/Isar and for programming and debugging in Isabelle/ML.

Isabelle/ML and in particular Isabelle's code generation have been most relevant for the integration of Isabelle/HOL and Standard ML code, and furthermore Isabelle/jEdit is used for

Command	Description
type_synonym	Defines a syntactic abbreviation of a type.
datatype	Defines an ML-style datatype.
definition	Defines a (non-recursive) function or constant.
abbreviation	Defines a syntactical abbreviation of a term.
primrec	Defines a primitive recursive function.
inductive	Defines an inductive predicate based on a set of introduction rules.
lemma	Declares a lemma and is followed by a proof.
theorem	Declares a theorem and is followed by a proof.
corollary	Declares a corollary and is followed by a proof.
code_reflect	Generates code that is reflected into the Isabelle/ML environment.

Table 1: A subset of Isabelle/HOL's commands

our declarative prover too. In addition, Sledgehammer was particularly useful for the starting point of our work, namely Alexander B. Jensen's thesis [16], since it allows Isabelle novices to prove theorems without having deep knowledge about Isabelle's library of theorems.

3 Architecture for Declarative Prover

Figure 4 shows the architecture of the entire development. The development consists of a single Isabelle theory file, which has an Isabelle/HOL part and an Isabelle/ML part. The two parts are connected with code reflection.

The part in Isabelle/HOL defines types for formulas and theorems, as well as functions for axioms and rules. These are then used inductively to define the proof system which is proved sound with respect to a semantics.

Hereafter, code reflection connects the Isabelle/HOL part with the Isabelle/ML part: Isabelle is instructed to generate code from the Isabelle/HOL definitions, and the code is then loaded into the Isabelle/ML environment. The loaded code consists of an ML module with a signature consisting of the type of formulas, the type of theorems, the axiom functions and the rule functions. It is the kernel of the declarative prover.

The part in Isabelle/ML defines the declarative prover. The declarative prover is a number of ML-functions that make calls into the kernel. These functions include derived rules, a tableau prover and various tactics.

The idea of the architecture is that we prove the axioms and rules sound in Isabelle/HOL. We load the axioms and rules into Isabelle/ML by using reflection, and the ML signature system then ensures that all values of the type for a theorem are built from the loaded axioms and rules. Thus these values represent theorems of the sound proof system.

Our entire development can run from a single file in a window in the Isabelle/jEdit IDE. As already mentioned the file is available and maintained in the Archive of Formal Proofs against the current release of Isabelle and the file includes both the Isabelle/HOL and the Isabelle/ML part [19]. It is possible to replace the code-reflection with a code generation tool that exports the kernel to a source code file in either OCaml, Haskell, Scala or SML.

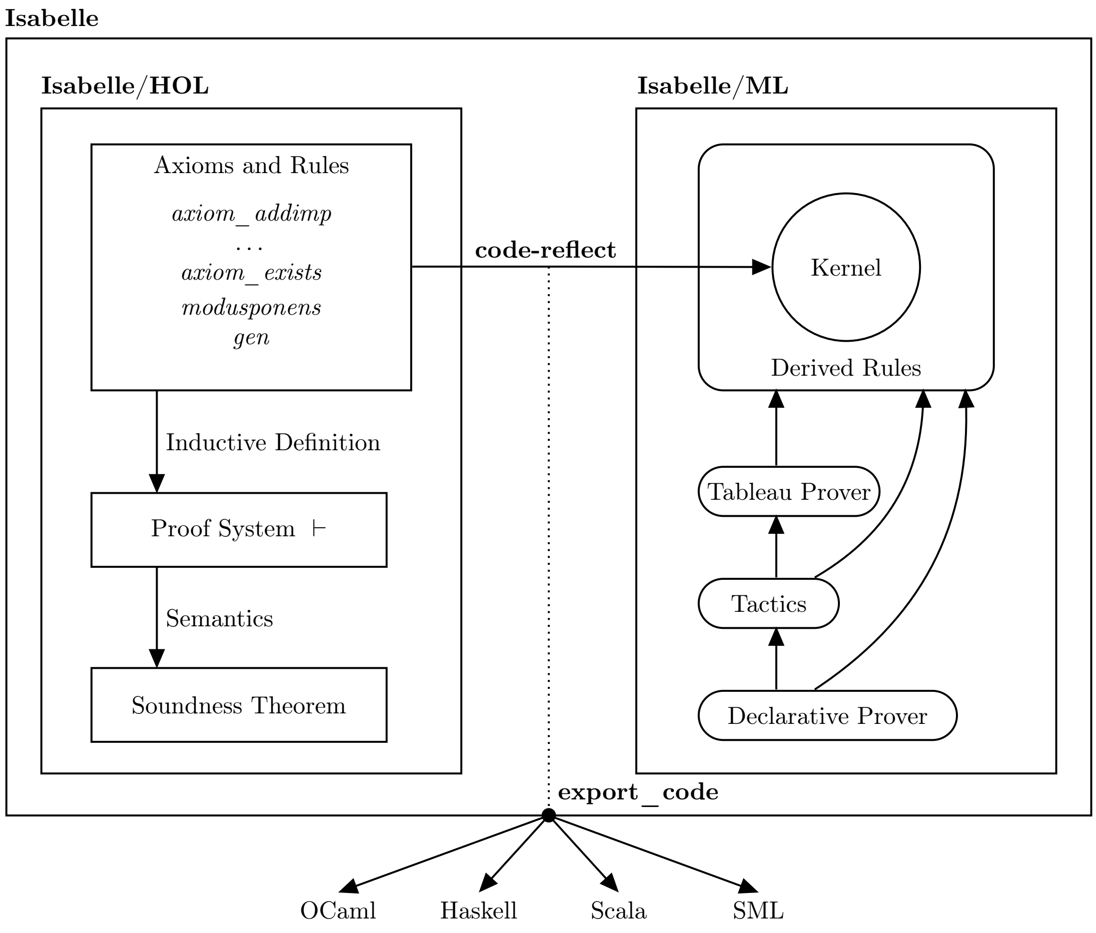


Figure 4: Architecture of the certified declarative prover in Isabelle.

4 Formalization of Terms and Formulas

We formalize formulas in the same style as Harrison's OCaml code which has the parameter *'a* for the type of atoms and variable identifiers represented as strings.

type_synonym *id* = *String.literal*

This way the atoms can be instantiated for either propositional logic or first-order logic.

datatype *'a fm* = *Truth* | *Falsity* | *Atom 'a* |
  Imp ⟨*'a fm*⟩ ⟨*'a fm*⟩ | *Iff* ⟨*'a fm*⟩ ⟨*'a fm*⟩ |
  And ⟨*'a fm*⟩ ⟨*'a fm*⟩ | *Or* ⟨*'a fm*⟩ ⟨*'a fm*⟩ |
  Not ⟨*'a fm*⟩ | *Exists id* ⟨*'a fm*⟩ | *Forall id* ⟨*'a fm*⟩

We similarly formalize the terms and first-order atoms. Function identifiers and predicate identifiers are also represented by strings:

datatype *tm* = *Var id* | *Fn id* ⟨*tm list*⟩

datatype *fol* = *Rl id* ⟨*tm list*⟩

Thus the first-order formulas are represented by the type *fol fm*.

5 Proof System

The entire axiomatic proof system can be seen in Figure 5 and comes from Harrison's textbook. The idea is that the validity of the rules and axioms should be evident and that they should be easy to implement. Harrison recognizes that substitution with named variable bindings is not entirely trivial. There are ways to alleviate these complications, for instance by using de Bruijn indices or nominals, but Harrison takes another approach. He takes inspiration from proof systems for first-order logic with equality by Tarski [39] and Monk [28] which avoid substitution entirely in their axioms and rules.

Harrison's rules and axioms in Figure 5 are structured as follows:

Inference rules (1-2) The inference rules are modus ponens (1) and generalization (2).

Propositional axioms (3-5) The propositional axioms, together with modus ponens (1), form a proof system of the propositional logic with $\longrightarrow$ and $\bot$ as the only operators. Harrison refers to the proof system P_0 by Church [7] which consists exactly of these three axioms and modus ponens.

First-order axioms (6-11) The first-order axioms, together with the propositional axioms and the inference rules, form a proof system for first-order logic with only the operators $\longrightarrow$, $\bot$, $\forall$ and the rest defined as abbreviations. Axioms 6-9 appear in the axiomatic systems by Tarski and Monk and so does an axiom similar to the congruence axioms 10-11.

Further operator axioms (12-19) These further operator axioms characterize $\longleftrightarrow$, $\top$, $\neg$, $\wedge$, $\vee$ and $\exists$ in terms of $\longrightarrow$, $\bot$ and $\forall$.

In addition to the advantage of leading to a simple kernel, the approach allows Harrison to present named variable bindings to the user without any conversion from an internal representation.

The same approach is used by the proof checker Metamath [25] which uses a similar set of axioms also inspired by Tarski [39].

6 Formalization of Axioms and Proof Rules

Since axioms and proof rules will be formalized as functions, they should be functions that return theorems. Therefore we introduce a datatype for the theorems, as well as a selector *concl*, such that *concl (Thm x) = x*.

datatype *"thm" = Thm (concl: ⟨fol fm⟩)*

We can then define the rules and axioms of the proof system as functions in Isabelle/HOL. Let us consider the simplest such function, namely *axiom_ addimp*.

```
definition axiom_addimp :: "fol fm ⇒ fol fm ⇒ thm"
where
 "axiom_addimp p q ≡ Thm (Imp p (Imp q p))"
```

This axiom simply implements the well-known axiom $p \longrightarrow (q \longrightarrow p)$. Notice also the type annotation. The axiom takes two formulas and returns a theorem.

We also consider a proof rule, namely *gen*, which is the generalization rule.

```
definition gen :: "id ⇒ thm ⇒ thm"
where
 "gen x s ≡ Thm (Forall x (concl s))"
```

This implements the rule $\frac{\vdash s}{\vdash \forall x.\ s}$. Notice that this function takes a theorem as input since it is a proof rule.

6.1 Side Conditions

The axioms *axiom_ impall* and *axiom_ existseq* have the side condition that x is not, respectively, free in p or occurs in t.

$$\frac{\neg \mathit{free\_in}\ x\ p}{p \longrightarrow (\forall x.\, p)} \qquad \frac{\neg \mathit{occurs\_in}\ x\ t}{\exists x.\, x = t}$$

Therefore we have to choose what the functions should return when the side conditions are not fulfilled.

Harrison chose to throw an exception but these are not available in Isabelle/HOL. We therefore considered several alternatives. One possibility would be to return *undefined*. Another possibility would be to return a *thm option* which would be *None* when the side conditions are not fulfilled.

We choose instead that the implementation returns *Thm Truth* when the side conditions are not fulfilled. This solution simplifies the code and the proofs. It clearly ensures soundness since, when things go wrong, we return a formula that is obviously valid.

abbreviation *(input) "fail_ thm ≡ Thm Truth"*

We define the following functions for terms and lists of terms:

```
primrec
  occurs_in :: "id ⇒ tm ⇒ bool"
and
  occurs_in_list :: "id ⇒ tm list ⇒ bool"
where
 "occurs_in i (Var x) = (i = x)" |
 "occurs_in i (Fn _ l) = occurs_in_list i l" |
 "occurs_in_list _ [] = False" |
 "occurs_in_list i (h # t) =
   (occurs_in i h ∨ occurs_in_list i t)"
```

1. modus ponens $$\frac{p \longrightarrow q \qquad p}{q}$$

2. generalization $$\frac{p}{\forall x.\, p}$$

3. axiom addimp $$\frac{}{p \longrightarrow q \longrightarrow p}$$

4. axiom distribimp $$\frac{}{(p \longrightarrow q \longrightarrow r) \longrightarrow (p \longrightarrow q) \longrightarrow p \longrightarrow r}$$

5. axiom doubleneg $$\frac{}{((p \longrightarrow \bot) \longrightarrow \bot) \longrightarrow p}$$

6. axiom allimp $$\frac{}{(\forall x.\, p \longrightarrow q) \longrightarrow (\forall x.\, p) \longrightarrow (\forall x.\, q)}$$

7. axiom impall $$\frac{\neg \mathit{free\_in}\ x\ p}{p \longrightarrow (\forall x.\, p)}$$

8. axiom existseq $$\frac{\neg \mathit{occurs\_in}\ x\ t}{\exists x.\, x = t}$$

9. axiom eqrefl $$\frac{}{t = t}$$

10. axiom funcong $$\frac{}{s_1 = t_1 \longrightarrow \cdots \longrightarrow s_n = t_n \longrightarrow f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n)}$$

11. axiom predcong $$\frac{}{s_1 = t_1 \longrightarrow \cdots \longrightarrow s_n = t_n \longrightarrow P(s_1, \ldots, s_n) \longrightarrow P(t_1, \ldots, t_n)}$$

12. axiom iffimp1 $$\frac{}{(p \longleftrightarrow q) \longrightarrow p \longrightarrow q}$$

13. axiom iffimp2 $$\frac{}{(p \longleftrightarrow q) \longrightarrow q \longrightarrow p}$$

14. axiom impiff $$\frac{}{(p \longrightarrow q) \longrightarrow (q \longrightarrow p) \longrightarrow (p \longleftrightarrow q)}$$

15. axiom true $$\frac{}{\top \longleftrightarrow (\bot \longrightarrow \bot)}$$

16. axiom not $$\frac{}{\neg p \longleftrightarrow (p \longrightarrow \bot)}$$

17. axiom and $$\frac{}{(p \land q) \longleftrightarrow ((p \longrightarrow q \longrightarrow \bot) \longrightarrow \bot)}$$

18. axiom or $$\frac{}{(p \lor q) \longleftrightarrow \neg(\neg p \land \neg q)}$$

19. axiom exists $$\frac{}{(\exists x.\, p) \longleftrightarrow \neg(\forall x.\, \neg p)}$$

Figure 5: The axiomatic proof system.

We define the following function for formulas:

```
primrec free_in :: "id ⇒ fol fm ⇒ bool"
where
 "free_in _ Truth = False" |
 "free_in _ Falsity = False" |
 "free_in i (Atom a) =
   (case a of Rl _ l ⇒ occurs_in_list i l)" |
 "free_in i (Imp p q) = (free_in i p ∨ free_in i q)" |
 "free_in i (Iff p q) = (free_in i p ∨ free_in i q)" |
 "free_in i (And p q) = (free_in i p ∨ free_in i q)" |
 "free_in i (Or p q) = (free_in i p ∨ free_in i q)" |
 "free_in i (Not p) = free_in i p" |
 "free_in i (Exists x p) = (i ≠ x ∧ free_in i p)" |
 "free_in i (Forall x p) = (i ≠ x ∧ free_in i p)"
```

```
definition axiom_impall :: "id ⇒ fol fm ⇒ thm"
where
 "axiom_impall x p ≡
   if ¬ free_in x p then Thm (Imp p (Forall x p))
   else fail_thm"
```

Axiom *axiom_existseq* is defined in the same way as axiom *axiom_impall*.

6.2 Congruence Axioms

The most complicated axioms are the congruence axioms, *axiom_funcong* and *axiom_predcong*.

$$\frac{}{\begin{array}{l} s_1 = t_1 \longrightarrow \cdots \longrightarrow s_n = t_n \longrightarrow \\ \qquad\qquad f(s_1, \cdots, s_n) = f(t_1, \cdots, t_n) \end{array}}$$

$$\frac{}{\begin{array}{l} s_1 = t_1 \longrightarrow \cdots \longrightarrow s_n = t_n \longrightarrow \\ \qquad\qquad P(s_1, \cdots, s_n) \longrightarrow P(t_1, \cdots, t_n) \end{array}}$$

For *axiom_funcong*, Harrison's implementation takes the two lists *lefts* $= [s_1, \ldots, s_n]$ and *rights* $= [t_1, \ldots, t_n]$ as input, and constructs the above nested implication.

```
let axiom_funcong f lefts rights =
      itlist2
        (fun s t p -> Imp (mk_eq s t,p)) lefts rights
          (mk_eq (Fn (f,lefts)) (Fn (f,rights)))
```

The function *itlist2* is defined as

```
let rec itlist2 f l1 l2 b =
  match (l1,l2) with
    ([],[]) -> b
  | (h1::t1,h2::t2) -> f h1 h2 (itlist2 f t1 t2 b)
  | _ -> failwith "itlist2";;
```

His idea is that we have a function which adds an equality of two terms as an antecedent to a formula. Then we can use that function and *itlist2* to iteratively add equalities of the terms in our lists as antecedents starting from the formula $f(t_1, \ldots, t_n) = f(s_1, \ldots, s_n)$.

Our formalization instead splits the functionality of *axiom_funcong* into two functions:

- *foldr Imp* takes a list of formulas $[F_1, \dots, F_n]$ and adds them as antecedents to a formula F to build a nested implication $F_1 \longrightarrow \cdots \longrightarrow F_n \longrightarrow F$.
- *zip_ eq* takes two lists of formulas, $[s_1, \dots, s_n]$, $[t_1, \dots, t_n]$ and builds the corresponding list of equalities $[s_1 = t_1, \dots, s_n = t_n]$.

```
definition zip_ eq :: "tm list ⇒ tm list ⇒ fol fm list"
where
 "zip_ eq l l′ ≡ map (λ(t, t′). Atom (Rl (STR ''='') [t, t′]))
                        (zip l l′) "
```

The idea of our approach is that we can separately reason about constructing equalities and adding antecedents, and this will make it easier to prove soundness. We now implement *axiom_funcong* as follows by first constructing the equalities, and then the nested implication.

```
definition axiom_funcong :: "id ⇒ tm list ⇒ tm list ⇒ thm"
where
 "axiom_funcong i l l′ ≡
   if equal_ length l l′ then
     Thm (foldr Imp (zip_ eq l l′)
              (Atom (Rl (STR ''='') [Fn i l, Fn i l′])))
   else fail_ thm"
```

We formalize *axiom_predcong* in a similar way.

```
definition axiom_predcong :: "id ⇒ tm list ⇒ tm list ⇒ thm"
where
 "axiom_predcong i l l′ ≡
   if equal_ length l l′ then
     Thm (foldr Imp (zip_ eq l l′)
              (Imp (Atom (Rl i l)) (Atom (Rl i l′))))
   else fail_ thm"
```

7 Formalization of Axiomatic Proof System

Since we want to prove the whole system sound, we need to characterize the theorems, which are built exclusively from the axioms and the rules. We therefore define the proof system as an inductive predicate *OK*. Writing ("⊢ _ " 0) we introduce the turnstile as syntax for the *OK* predicate where the underscore denotes that the formula follows the turnstile. The 0 denotes the precedence of the notation. After **where** follows each of the rules and axioms as introduction rules in the inductive predicate. The underscores there are dummy variables, that is, each one of them corresponds to a fresh Isabelle/HOL variable.

```
inductive OK :: "fol fm ⇒ bool" ("⊢ _ " 0)
where
```

```
modusponens:
    "⊢ concl s  ⟹
     ⊢ concl s' ⟹     ⊢ concl (modusponens s s')" |
gen:
    "⊢ concl s  ⟹     ⊢ concl (gen _ s)" |
axiom_addimp:          "⊢ concl (axiom_addimp _ _)" |
axiom_distribimp:      "⊢ concl (axiom_distribimp _ _ _)" |
axiom_doubleneg:       "⊢ concl (axiom_doubleneg _)" |
axiom_allimp:          "⊢ concl (axiom_allimp _ _ _)" |
axiom_impall:          "⊢ concl (axiom_impall _ _)" |
axiom_existseq:        "⊢ concl (axiom_existseq _ _)" |
axiom_eqrefl:          "⊢ concl (axiom_eqrefl _)" |
axiom_funcong:         "⊢ concl (axiom_funcong _ _ _)" |
axiom_predcong:        "⊢ concl (axiom_predcong _ _ _)" |
axiom_iffimp1:         "⊢ concl (axiom_iffimp1 _ _)" |
axiom_iffimp2:         "⊢ concl (axiom_iffimp2 _ _)" |
axiom_impiff:          "⊢ concl (axiom_impiff _ _)" |
axiom_true:            "⊢ concl axiom_true" |
axiom_not:             "⊢ concl (axiom_not _)" |
axiom_and:             "⊢ concl (axiom_and _ _)" |
axiom_or:              "⊢ concl (axiom_or _ _)" |
axiom_exists:          "⊢ concl (axiom_exists _ _)"
```

8 Semantics

To prove the rules sound, we of course need a semantics of terms and formulas. The formalization is mostly straightforward. We represent universes as types, and therefore the semantics refers to the universe by a type variable $'a$. A noteworthy case of the semantics is the one for the atoms, where we interpret the = predicate applied to two terms as an equality. This is done by evaluating the terms and seeing if their values are equal.

```
primrec — Semantics of terms
  semantics_term ::
    "(id ⇒ 'a) ⇒ (id ⇒ 'a list ⇒ 'a) ⇒ tm ⇒ 'a"
and
  semantics_list ::
    "(id ⇒ 'a) ⇒ (id ⇒ 'a list ⇒ 'a) ⇒ tm list ⇒ 'a list"
where
  "semantics_term e _ (Var x) = e x" |
  "semantics_term e f (Fn i l) = f i (semantics_list e f l)" |
  "semantics_list _ _ [] = []" |
  "semantics_list e f (t # l) =
    semantics_term e f t # semantics_list e f l"
```

```
primrec — Semantics of formulas
  semantics
    :: "(id ⇒ 'a) ⇒ (id ⇒ 'a list ⇒ 'a) ⇒
                (id ⇒ 'a list ⇒ bool) ⇒ fol fm ⇒ bool"
where
  "semantics _ _ _ Truth = True" |
  "semantics _ _ _ Falsity = False" |
  "semantics e f g (Atom a) =
    (case a of Rl i l ⇒
      if i = STR ''='' ∧ length2 l then
        (semantics_term e f (hd l) =
          semantics_term e f (hd (tl l)))
      else g i (semantics_list e f l))'' |
  ''semantics e f g (Imp p q) =
    (semantics e f g p ⟶ semantics e f g q)'' |
  ''semantics e f g (Iff p q) =
    (semantics e f g p ⟷ semantics e f g q)'' |
  ''semantics e f g (And p q) =
    (semantics e f g p ∧ semantics e f g q)'' |
  ''semantics e f g (Or p q) =
    (semantics e f g p ∨ semantics e f g q)'' |
  ''semantics e f g (Not p) = (¬ semantics e f g p)'' |
  ''semantics e f g (Exists x p) =
    (∃ v. semantics (e(x := v)) f g p)'' |
  ''semantics e f g (Forall x p) =
    (∀ v. semantics (e(x := v)) f g p)''
```

9 Soundness of the Proof System

Harrison only presents a very high-level soundness proof which leaves most of the exercise up to the reader. Furthermore, his proof is about the proof system, not its implementation. Our approach is therefore to device a proof ourselves, using Isabelle/jEdit to explore proofs and to help us reveal the necessary lemmas.

We prove the soundness by rule induction on the proof system, and thus need to prove one case for each axiom and rule. Cases for the axioms without side conditions and the *gen* rule are proved sound using only the automation of Isabelle/HOL.

Cases for the axioms with side conditions are not as easy to prove. Here, we need to come up with appropriate lemmas to prove them sound. We present and explain these lemmas.

The *modus_ponens* case is proved with a short declarative proof that also relies on automation.

9.1 Axioms with Non-Free or Non-Occurring Variables

The first challenge in the soundness proof is the two axioms, *axiom_impall* and *axiom_existseq*, that require a variable to be respectively non-free or non-occurring in an expression. For *axiom_existseq* it is clear that the formula holds if we assign the value of x to t. By inspecting the semantics of the existential quantifier, we realize that the variable x must not occur in t. It is however not clear to Isabelle that this problem is avoided when x does not occur in t. The lemma *map'* explicitly states that if x does not occur in t, then the semantic value of t is the same for all values of x. The statement is extended to hold for lists of terms due to the inductive definition of terms.

```
lemma map':
  ''¬ occurs_in x t ⟹
      semantics_term e f t = semantics_term (e(x := v)) f t''
  ''¬ occurs_in_list x l ⟹
      semantics_list e f l = semantics_list (e(x := v)) f l''
```

The lemma *map* is similar, but states that if the variable x does not occur freely in p (or if it does not occur at all) then the truth value of p is the same for all values of x.

lemma *map*:
$''\neg$ *free_in x p* $\Longrightarrow$
    semantics e f g p $\longleftrightarrow$ *semantics* $(e(x := v))$ *f g p''*

By inspecting the semantics of the universal quantifier, we see that this exactly states that the semantics of p is the same as $\forall x.\, p$. This fact is an even stronger result than what we need to prove *axiom_impall* valid. We are now ready to prove *axiom_impall* using *map* and *axiom_existseq* using *map'*.

9.2 Congruence Axioms

The next challenge is to prove the congruence axioms, *axiom_funcong* and *axiom_predcong* sound. We now take advantage of the *foldr Imp* function we introduced earlier, and prove a lemma explaining its semantics. The lemma states that a nested implication is true exactly when the truth of its antecedents implies the truth of its conclusion.

lemma *imp_chain_equiv*:
''semantics e f g (*foldr Imp l p*) $\longleftrightarrow$
  ($\forall\, q \in$ *set l. semantics e f g q*) $\longrightarrow$ *semantics e f g p''*

We then also state a lemma which explains the semantics of *foldr Imp* (*zip_eq l l'*) *p*. The lemma states that it holds exactly when the semantical equality between l and l' implies the truth of p.

lemma *imp_chain_zip_eq*:
''equal_length l l' $\Longrightarrow$
    semantics e f g (*foldr Imp* (*zip_eq l l'*) *p*) $\longleftrightarrow$
        semantics_list e f l = *semantics_list e f l'* $\longrightarrow$
            semantics e f g p''

With this we prove the congruence axioms sound using automation in lemmas *funcong* and *predcong*. It is easy to see that the lemma *funcong* proves a theorem in the first-order logic, since its conclusion is encapsulated by *semantics*. The formula is the main content of the Isabelle definition of *axiom_funcong*.

lemma *funcong*:
''equal_length l l' $\Longrightarrow$
    semantics e f g (*foldr Imp* (*zip_eq l l'*)
                    (*Atom* (*Rl* (*STR ''=''*) [*Fn i l*, *Fn i l'*])))*''*

Likewise in the lemma *predcong*, we see a theorem in the first-order logic. The formula is the main content of the Isabelle definition of *axiom_predcong*.

lemma *predcong*:
''equal_length l l' $\Longrightarrow$
    semantics e f g (*foldr Imp* (*zip_eq l l'*)
                    (*Imp* (*Atom* (*Rl i l*)) (*Atom* (*Rl i l'*))))*''*

9.3 Soundness Theorem

We then prove soundness. Often, soundness is expressed as provability implying validity. Therefore we would like a HOL predicate expressing validity. That is unfortunately not possible, because of our choice of representing universes as types, which one cannot quantify over inside HOL.

Instead, we express soundness as follows:

theorem *soundness*:
"⊢ p ⟹ semantics e f g p"

This also expresses soundness, since it states that the provability of any formula p implies its truth, for any environment e, function denotation f and predicate denotation g. The proof is by rule-induction on the proof system as described.

From our main theorem we immediately obtain a consistency corollary which states that there is a formula that we cannot prove:

corollary *"¬ (⊢ Falsity)"*
using *soundness*
by *fastforce*

10 Prover

Since we have defined all the necessary datatypes for our logic as well as the axioms and rules used to construct theorems, we are ready to expose them to the Isabelle/ML environment using code-reflection. To do this we use the Isabelle **code_reflect** command which takes a structure name as well as a list of datatypes and their constructors, as well as a list of functions. It then generates a signature and structure based on them and exposes it to the Isabelle/ML environment. In particular, we tell Isabelle that the datatypes *fm*, *tm* and *fol* should be in the signature along with their respective constructors. Likewise we tell Isabelle that the signature should include functions *modusponens, gen, . . . , concl*.

```
code_reflect
  Proven
datatypes
  fm = Falsity | Truth | Atom | Imp | Iff |
       And | Or | Not | Exists | Forall
and
  tm = Var | Fn
and
  fol = Rl
functions
  modusponens gen axiom_addimp axiom_distribimp
  axiom_doubleneg axiom_allimp axiom_impall
  axiom_existseq axiom_eqrefl axiom_funcong
  axiom_predcong axiom_iffimp1 axiom_iffimp2
  axiom_impiff axiom_true axiom_not axiom_and
  axiom_or axiom_exists concl
```

Let us inspect the signature of the generated module:

```
structure Proven:
  sig
    val axiom_addimp: fol fm -> fol fm -> thm
    val axiom_allimp: string -> fol fm -> fol fm -> thm
    val axiom_and: fol fm -> fol fm -> thm
    val axiom_distribimp: fol fm -> fol fm -> fol fm -> thm
    val axiom_doubleneg: fol fm -> thm
    val axiom_eqrefl: tm -> thm
    val axiom_exists: string -> fol fm -> thm
    val axiom_existseq: string -> tm -> thm
    val axiom_funcong: string -> tm list -> tm list -> thm
    val axiom_iffimp1: fol fm -> fol fm -> thm
    val axiom_iffimp2: fol fm -> fol fm -> thm
    val axiom_impall: string -> fol fm -> thm
    val axiom_impiff: fol fm -> fol fm -> thm
    val axiom_not: fol fm -> thm
    val axiom_or: fol fm -> fol fm -> thm
    val axiom_predcong: string -> tm list -> tm list -> thm
    val axiom_true: thm
    val concl: thm -> fol fm
    datatype 'a fm =
        And of 'a fm * 'a fm
      | Atom of 'a
      | Exists of string * 'a fm
      | Falsity
      | Forall of string * 'a fm
      | Iff of 'a fm * 'a fm
      | Imp of 'a fm * 'a fm
      | Not of 'a fm
      | Or of 'a fm * 'a fm
      | Truth
    datatype fol = Rl of string * tm list
    val gen: string -> thm -> thm
    val modusponens: thm -> thm -> thm
    type num
    type thm
    datatype tm = Fn of string * tm list | Var of string
  end
```

By inspecting the signature of the reflected module we see that the only functions that return values of type *thm* are the axioms and rules. This fact and the soundness proof certify the soundness of the kernel assuming that we trust ML's type-system and Isabelle's code generator.

Notice that a user cannot write e.g. *Thm Falsity* in ML since the *Thm* value constructor is not exposed in the signature and thus unavailable to the user.

11 Declarative Prover

The signature above fits with the one in Harrison's prover. We translate his prover from OCaml to SML such that we can run the prover inside of Isabelle in the Isabelle/ML environment. The translation was not too difficult, but there were some challenges arising from the differences between SML (as defined in the revised standard from 1997 [27]) and OCaml.

- In SML there is no built-in polymorphic ordering and hashing. Therefore we, when needed, define orderings and hash functions explicitly for each datatype.
- In SML there is no shallow/pointer comparison. All places it is used in the OCaml version we can fortunately replace it with structural equality.
- In SML one cannot put guards on case-expressions. Therefore we use if-then-else instead in these cases.
- OCaml has widely used preprocessors (camlp4 and camlp5). Harrison uses them when parsing formulas. We choose not to use a preprocessor. One unfortunate consequence of

this is that when we want to use formulas as input, they are strings, and thus /\ needs to written as /\\ in order to escape the backslash.

Harrison's OCaml code contains many examples that are run when executing his code. We have collected these examples and translated them to SML. We have systematically tested that both versions produce the same output. The results are available online [37].

Let us take a look at how (our translation of) Harrison's proof assistant works and how it plugs into our generated kernel.

11.1 Derived Rules

The rules and axioms are functions that return theorems. By combining them Harrison obtains new such functions, i.e. derived rules. For instance the rule which takes a theorem $\vdash q$ and produces $\vdash p \longrightarrow q$ where p is some formula. The ML-implementation of this looks as follows:

```
fun add_assum p th =
    modusponens (axiom_addimp (concl th) p) th;
```

The idea is that the above code implements the following proof where we think of q as *concl th*:

1. $\vdash q$
2. $\vdash q \longrightarrow p \longrightarrow q$ (axiom_addimp)
3. $\vdash p \longrightarrow q$ (modusponens 1 2)

We can also inspect the type of *add_assum* and see that it indeed takes a formula and a theorem and returns a theorem:

```
fm -> thm -> thm
```

11.2 Tableau Prover

Harrison implements a tableau prover for first-order logic on top of the kernel. It is implemented as code and thus calls into the kernel. The prover implements a tableau system with unification — see e.g. Wikipedia [45] or Hähnle's chapter in the *Handbook of Automated Reasoning* [11]. The tableau is expanded in a preorder-fashion. Whenever a pair of complementary literals is found the resulting unifier will be applied to an environment that is passed on to the next node to be expanded. In the code, branching on a disjunctive formula is handled by working on the left branch immediately, and delaying the work on the right branch by building a continuation function.

The tableau prover takes as parameter a number n indicating how many times universal quantifiers are allowed to introduce fresh variables. An outer function tries to build tableaux with larger and larger n until it, if the formula can be refuted, succeeds.

Harrison proves informally that the tableau prover is complete for first-order logic without equality.

11.3 Tactics

Tactics are a way to implement backwards reasoning in a proof assistant. When a user wants to use tactics he first states the goal he wants to prove. He can then use a tactic to reduce the goal to a number of subgoals from which the goal follows. The subgoals can likewise be reduced with tactics until they become trivial and then the proof is done.

A state in this process is represented by a datatype *goals* which looks as follows in the ML-implementation:

datatype *goals* =
  Goals **of** ((*string* ∗ *fol fm*) *list* ∗ *fol fm*)*list* ∗
    (*thm list* −> *thm*);

Each (*string* ∗ *fol fm*) *list* ∗ *fol fm* represents a subgoal with a number of assumptions. More precisely, the subgoal value $([p_1, \ldots, p_i], q)$ represents the implication $p_1 \wedge ... \wedge p_i \longrightarrow q$.

The ((*string* ∗ *fol fm*) *list* ∗ *fol fm*) *list* represents a list of subgoals:

$$\begin{array}{lcl} p_{11} \wedge \ldots \wedge p_{1i_1} & \longrightarrow & q_1 \\ \vdots & & \\ p_{n1} \wedge \ldots \wedge p_{ni_n} & \longrightarrow & q_n \end{array}$$

The *string* in the type allows us to label the assumptions.

The (*thm list* −> *thm*) is called a justification function and represents a rule which will bring us from the subgoals to the goal P we ultimately want to prove. It should thus represent a rule on the following form:

$$\left(\begin{array}{lcl} \vdash p_{11} \wedge \ldots \wedge p_{1i_1} & \longrightarrow & q_1 \\ \vdots & & \\ \vdash p_{n1} \wedge \ldots \wedge p_{ni_n} & \longrightarrow & q_n \end{array}\right) \Longrightarrow \vdash P$$

Let's consider a simple example of a *goals*. Say we want to prove $\top \wedge \top$. A *goals* for this could be a list with the single subgoal

$$\top \wedge \top$$

together with a justification:

$$(\vdash \top \wedge \top) \Longrightarrow (\vdash \top \wedge \top)$$

In ML this could be the value ([([],*And*(*Truth*,*Truth*))],*hd*) where *hd* gives the head of a list, and so indeed when it is given $[\vdash \top \wedge \top]$ it will return $\vdash \top \wedge \top$.

A tactic is then simply a function of the type *goals* -> *goals* that should reduce the subgoals to something simpler and change the justification function accordingly. This is similar to how it was done in LCF [9, 26].

For instance we could apply a conjunction introduction tactic to our current example which would then produce the subgoals

$$\begin{array}{c} \top \\ \top \end{array}$$

together with a justification:

$$\left(\begin{array}{l} \vdash \top \\ \vdash \top \end{array}\right) \Longrightarrow \vdash \top \wedge \top$$

A simple example of a tactic is the conjunction introduction tactic which replaces a subgoal of the form $a \longrightarrow p \wedge q$ with two subgoals $a \longrightarrow p$ and $a \longrightarrow q$.

Let us look at how to program the conjunction introduction tactic. In general, the tactic is supposed to go from a *goals* with the following justification (and a corresponding list of subgoals)

$$\left(\begin{array}{lcl} \vdash p_{11} \wedge \ldots \wedge p_{1i_1} & \longrightarrow & a \wedge b \\ \vdash p_{21} \wedge \ldots \wedge p_{2i_2} & \longrightarrow & q_2 \\ \vdots & & \\ \vdash p_{n1} \wedge \ldots \wedge p_{ni_n} & \longrightarrow & q_n \end{array}\right) \Longrightarrow \vdash P$$

to the following justification (and a corresponding list of subgoals)

$$\begin{pmatrix} \vdash p_{11} \wedge \ldots \wedge p_{1i_1} & \longrightarrow & a \\ \vdash p_{11} \wedge \ldots \wedge p_{1i_1} & \longrightarrow & b \\ \vdash p_{21} \wedge \ldots \wedge p_{2i_2} & \longrightarrow & q_2 \\ \vdots & & \\ \vdash p_{n1} \wedge \ldots \wedge p_{ni_n} & \longrightarrow & q_n \end{pmatrix} \Longrightarrow \vdash P$$

The tactic is implemented as follows. The call it makes to *imp_trans_chain* and *and_pair* is described below, but for brevity we leave out the function definitions.

```
fun conj_intro_tac (Goals((asl,And(a,b))::gls,jfn)) =
  let fun jfn' (tha::thb::ths) =
    jfn(imp_trans_chain [tha, thb] (and_pair a b)::ths) in
  Goals((asl,a)::(asl,b)::gls,jfn')
  end;
```

The subgoals are changed as described – namely from *(asl,And(a,b))::gls* to *(asl,a)::(asl,b)::gls*. There is also a new justification function *jfn'*. In its definition the function call *imp_trans_chain [tha, thb] (and_pair a b)* takes theorems $\vdash asl \longrightarrow a$ and $\vdash asl \longrightarrow b$ and from these produces $\vdash asl \longrightarrow a \wedge b$ by calls into the kernel. When this is done we have the list of theorems that the original justification function expected and we can then simply apply it to produce the theorem we finally want.

Harrison implements several functions that are, or return, tactics:

- *conj_intro_tac* – conjunction introduction
- *forall_intro_tac* – forall introduction
- *exists_intro_tac* – existential introduction
- *imp_intro_tac* – implication introduction
- *auto_tac* – tableau prover
- *lemma_tac* – adding a new assumption
- *exists_elim_tac* – existential elimination
- *disj_elim_tac* – disjunction elimination

11.4 Declarative Proofs

Harrison builds the deductive prover on top of the tactics. The first step is to define a function *prove* which takes a formula and a list of tactics. It then sets up a *goals* with that formula and applies the tactics in the list one after another. In the end it returns the formula as a theorem if the tactics were successful in proving it.

We use this function in Fig. 6 to conduct a proof. Notice that the proof is actually an SML-expression directly calling *prove*. Because proofs are SML-expressions it is easy to extend the prover's syntax by writing new functions. The proof has a nested structure with some subproofs introduced by the function *proof* that are processed in a similar way. Some of these proofs use the *have* function to state intermediate steps towards proving the final goal. Let us first look at how this works if we for instance write *have p using [q]* in some goals *g*. Here *p* is some formula and

q is some theorem. The *have* function calls *lemma_tac* with p and *using [q]*, and the following happens:

The *goals* g has a first subgoal of the form $asl \longrightarrow w$. This subgoal is replaced with $p \longrightarrow asl \longrightarrow w$. Furthermore the justification function is changed. The new one calls *using [q]* which constructs the theorem $\vdash asl \longrightarrow univeral\_closure\ q$. Hereafter it will use the tableau prover to construct $\vdash universal\_closure\ q \longrightarrow p$ if possible. From this follows $\vdash asl \longrightarrow p$. The new justification function expects as input $\vdash p \longrightarrow asl \longrightarrow w$ and therefore it can now construct $\vdash asl \longrightarrow w$. This is what the old justification function expected as input and thus it is applied.

As we saw, *have* and *using* can be used to prove an intermediate step with a previously proved theorem. Likewise, *have* and *by* can be used to refer to a previously established fact in the proof. In the implementation this can be done by ensuring that the steps that introduce facts put them in the assumption list of the goals – as we saw *have* did. Then *by* can simply find them there by their name. Combining *have* and $proof$ allows subproofs to prove intermediate steps in a similar manner.

Other tactics to be used in declarative proofs are

- *note* – similar to *have*, but the intermediate step is named.
- *fix* – which is simply a forall-introduction rule.
- *assume* – which does implication introduction.
- *consider* – which does existential elimination and introduces an appropriate variable.
- *so* – which modifies e.g. the *have* tactic to use the previous fact to prove its intermediate step.
- *conclude* – indicates that we prove a subgoal.
- *qed* – indicates the end of the proof.

The declarative prover is able to give the user a rudimentary form of feedback when developing the proof:

- The type system of SML will tell the user if she enters a proof wrongly on the highest level
- On the lower level, the tactics will throw exceptions if they are applied on a *goals* they did not expect.

We have tried several workflows for building proofs:

- It is possible, but arguably a bit tedious, to build the proof from scratch with the help from the SML type system and the exceptions.
- Another possibility is to start with a formula that can be proved with the tableau prover, and then expand the proof more and more to the desired granularity, each time filling in the next part to be expanded with a call to the tableau prover. This of course only works on formulas that the prover can prove.
- A third possibility is to write the proof first by manually applying tactics to goals and printing the resulting goals until one has a proof. Hereafter the proof can be reconstructed in the declarative style.

Isabelle/jEdit presents both type errors and exceptions directly in its output panel, which is updated live while the user is writing her proof. However, there is definitely room for improvement when it comes to the usability of the prover.

For declarative proofs it is a huge advantage to have powerful proof automation that can take care of some of the simpler steps. As a rather challenging example, consider the following formalization with predicate r for *rich* and function f for *father* [23, page 128]:

> *If every person that is not rich has a rich father, then some rich person must have a rich grandfather.*

$$\forall x(\neg r(x) \to r(f(x))) \to \exists x(r(x) \wedge r(f(f(x))))$$

The tableau prover can in fact find the proof automatically and almost instantaneously. We can easily use the tableau prover and/or the declarative prover as a stand-alone program as follows. In Isabelle, we can introduce a Standard ML function *auto* and test it on some examples including the above one (of course a more advanced version of the function *auto* is possible, also using some helper functions):

```
ML {*
 fun auto s = prove (<!s!>) [our thesis at once, qed]
*}

ML_val {* auto "A ==> A" *}

ML_val {* auto "exists x. D(x) ==> forall x. D(x)" *}

ML_val {* auto "(forall x. ~R(x) ==> R(f(x)))
                         ==> exists x. R(x) /\\ R(f(f(x)))" *}
```

Using Isabelle's code generator we obtain a stand-alone Standard ML program *auto* for the certified automated theorem prover. We have then used the tool SMLtoJs ("SML toys") to translate the Standard ML code to JavaScript such that we can use it in our NaDeA system [40] running in a browser (here *auto* is run for just a fraction of a second and if necessary terminated).

12 Evaluation of the Declarative Prover

We first evaluate the usability of the declarative prover and then we evaluate the adequacy of the soundness proof.

12.1 Usability of the Declarative Prover

Recall Figures 1 and 2 with Pelletier's problem 46. With this example we have already shown that we can prove a challenging theorem in the prover with the declarative style.

We now wish to further evaluate the prover by using it to prove a mathematical theorem. We therefore consider Pelletier's problem 43. The problem defines from a relation P another relation Q as follows:

$$Q(x,y) \longleftrightarrow (\forall z.\ P(z,x) \longleftrightarrow P(z,y))$$

The problem then claims that Q is symmetric.

Additionally, we to want construct a declarative proof with a stronger resemblance to our understanding of thorough paper proofs as chains of sentences. Thus, the proof should break down the structure of the formula to an appropriate level, and on that level resemble a thorough paper proof consisting of a chain of sentences.

Figure 6 shows such a proof in the declarative proof language. We also discuss this and some alternative proofs in Alexander B. Jensen's thesis [16].

The first step of the proof is to show $\forall x\, y.\, Q(x, y) \longleftrightarrow Q(y, x)$ assuming $(\forall x\, y.\, Q(x, y) \longleftrightarrow \forall z.\, P(z, x) \longleftrightarrow P(z, y))$. We assume the left-hand side of the implication in the main formula with the *assume* command and give it the name A. The command is in many ways similar to its Isabelle counterpart. We then fix variables x and y using *fix x, fix y* to eliminate the quantifiers. We further break down the problem and show the conjunction of both directions of the bi-implication in $Q(x, y) \longleftrightarrow Q(y, x)$. We show the formula using the command *have* which is similar to *conclude* except that it does not have to directly solve a sub-goal. The sub-goal $\forall z.\, P(z, y) \longleftrightarrow P(z, x)$ for the $\longrightarrow$ direction is solved by using the assumption A which is achieved by *so have* (< !"*forall z. P(z,x) <=> P(z,y)*"!>) *by* ["*A*"]. In the following sub-goal, where the left-hand side and right-hand side are swapped, we use only the previous fact and no assumptions. The command *at once* can be used when the goal can be solved by pure first-order reasoning from the previous fact. From the conjunction of implications, we show that it is equivalent to the bi-implication using *so our thesis at once* and thus finish the proof.

To show that this proof is comparable to the declarative proofs in Isabelle/Isar and Isabelle/HOL, we present in Figure 7 a similar proof in that system. The correspondence is clear.

12.2 Adequacy of the Soundness Proof

Let us also evaluate the soundness proof. In order to believe it, we need to convince ourselves that the $\vdash$ predicate indeed represents the ML-type *thm*. In order to do this we need to check that all the axioms and rules for which we generate code, indeed appear in the definition of $\vdash$. The process is easy but allows for mistakes due to human error. Imagine for instance that someone writes the following axiom, and generates code for it, but forgets to add it to $\vdash$ – thus bypassing the soundness proof.

definition *axiom_false* :: *"thm"* **where**
  "axiom_false ≡ Thm Falsity"

It is not a catastrophe, since his peers can spot his mistake by inspection of $\vdash$, but is none the less undesirable.

One way to remedy this problem is to disallow, also in Isabelle/HOL, constructions such as *Thm Falsity*. One way to do this is to define the axioms, rules and $\vdash$ on formulas instead of theorems. Hereafter one can define the type of theorems as the set of formulas derivable with $\vdash$ using Isabelle's *typedef* command. The lifting package of Isabelle can then lift the axioms and rules to work on this new type. The *concl* function will then be defined to convert theorems back to formulas.

With such a definition we can express soundness:

theorem *soundness*: *"semantics e f g (concl p)"*

And consistency:

theorem *consistency*: *"concl p ≠ Falsity"*

Another, similar, way to remedy the problem is to define a predicate characterizing the valid formulas and then define the theorems as the valid formulas using *typedef*. Unfortunately, as we noticed in Section 9.3, this is not possible. One way to overcome this is to introduce a new type

```
prove
    (<!"(forall x y. Q(x,y) <=> forall z. P(z,x) <=> P(z,y)) ==> forall x y. Q(x,y) <=> Q(y,x)"!>)
    [
        assume [("A", <!"forall x y. Q(x,y) <=> forall z. P(z,x) <=> P(z,y)"!>)],
        conclude (<!"forall x y. Q(x,y) <=> Q(y,x)"!>) proof
        [
            fix "x", fix "y",
            conclude (<!"Q(x,y) <=> Q(y,x)"!>) proof
            [
                have (<!"(Q(x,y) ==> Q(y,x)) /\\ (Q(y,x) ==> Q(x,y))"!>) proof
                [
                    conclude (<!"Q(x,y) ==> Q(y,x)"!>) proof
                    [
                        assume [("", <!"Q(x,y)"!>)],
                        so have (<!"forall z. P(z,x) <=> P(z,y)"!>) by ["A"],
                        so have (<!"forall z. P(z,y) <=> P(z,x)"!>) at once,
                        so conclude (<!"Q(y,x)"!>) by ["A"],
                        qed
                    ],
                    conclude (<!"Q(y,x) ==> Q(x,y)"!>) proof
                    [
                        assume [("", <!"Q(y,x)"!>)],
                        so have (<!"forall z. P(z,y) <=> P(z,x)"!>) by ["A"],
                        so have (<!"forall z. P(z,x) <=> P(z,y)"!>) at once,
                        so conclude (<!"Q(x,y)"!>) by ["A"],
                        qed
                    ],
                    qed
                ],
                so our thesis at once,
                qed
            ],
            qed
        ],
        qed
    ]
```

Figure 6: A detailed proof of Pelletier's problem 43 in the declarative prover.

```
lemma "(∀x y. Q(x,y) ⟷ (∀z. P(z,x) ⟷ P(z,y))) ⟶ (∀x y. Q(x,y) ⟷ Q(y,x))"
proof
    assume A: "∀x y. Q(x,y) ⟷ (∀z. P(z,x) ⟷ P(z,y))"
    show "∀x y. Q(x,y) ⟷ Q(y,x)"
    proof (rule, rule)
        fix x y
        show "Q(x,y) ⟷ Q(y,x)"
        proof -
            have "(Q(x,y) ⟶ Q(y,x)) ∧ (Q(y,x) ⟶ Q(x,y))"
            proof
                show "Q(x,y) ⟶ Q(y,x)"
                proof
                    assume "Q(x,y)"
                    then have "∀z. P(z,x) ⟷ P(z,y)" using A by blast
                    then have "∀z. P(z,y) ⟷ P(z,x)" by blast
                    then show "Q(y,x)" using A by blast
                qed
            next
                show "Q(y,x) ⟶ Q(x,y)"
                proof
                    assume "Q(y,x)"
                    then have "∀z. P(z,y) ⟷ P(z,x)" using A by blast
                    then have "∀z. P(z,x) ⟷ P(z,y)" by blast
                    then show "Q(x,y)" using A by blast
                qed
            qed
            then show "Q(x,y) ⟷ Q(y,x)" by blast
        qed
    qed
qed
```

Figure 7: A detailed proof of Pelletier's problem 43 in Isabelle/HOL.

U, using Isabelle's *typedecl* command, and then assume absolutely nothing about it. Then if a formula evaluates to true for all environments and interpretations over this universe, we can, informally, argue that it must be valid, since U is completely arbitrary. Again we can then define the rules and axioms on the type of formulas, and then lift them to work on theorems. With this approach soundness is captured in the types – any theorem value or function that returns a theorem is valid. Consistency can be expressed and proved in the same way as when we lifted $\vdash$. One can, however, argue that defining the theorems as the valid formulas goes against the meaning of the word theorem – theorem being a syntactic notion and validity being a semantic notion. In first-order logic the two words capture the same meaning for sound and complete proof systems, but for other logics such as ZFC there are no sound and complete proof systems with respect to their usual semantics, and thus the words have very distinct meanings.

We have implemented both approaches of having *thm* as a type. Their code is available online [18]. We, however, prefer our current approach because we feel that for teaching purposes there are already enough concepts to talk about and adding lifting to the mix might confuse more than help.

13 Related Work

The literature contains several other formalizations of logic and contains also declarative provers. Let us first look at some other formalizations of logic.

Harrison [12] formalized, in HOL Light, soundness and consistency proofs for the HOL of HOL Light without definitions. More precisely he considered three different logics: HOL, HOL $+ I$ and HOL $- \infty$. HOL $+ I$ is HOL extended with an axiom claiming the existence of a very large cardinal, and HOL $- \infty$ is HOL where the axiom of infinity is removed. His results are to prove in HOL $+ I$ that HOL is sound and consistent, and to prove in HOL that HOL $- \infty$ is sound and consistent. Kumar et al. [20] extended Harrison's work by proving, in HOL4, that HOL with definitions is sound and consistent. Their proofs rely on assuming a specification of a set-theory. Our work differs from this by using the meta-logic of Isabelle/HOL and the object logic of FOL. Using Isabelle/HOL on the meta-level has at least two advantages seen from a teaching perspective. Firstly, Isabelle/HOL provides a complete integrated package of proof assistant, prover integrated development environment and code-generation. This enables students to load the entire development directly in Isabelle including verification, code-reflection and the execution of the prover. Secondly, having FOL on the object level has pedagogical advantages, since it is a logic that students are often familiar with and thus we can assume they have some understanding of its syntax and semantics. Thus, we see our development as a pedagogical stepping stone students can take towards the self-verifications of Harrison and Kumar et al.

Other provers based on verified proof systems for first-order logics are our NaDeA system [40] and Breitner's The Incredible Proof Machine [6]. They offer, by design, only limited automation and the connection between the verification and the implementation is, furthermore, entirely informal. Margetson and Ridge's automatic prover for first-order logic in negation normal form without first-order terms [24, 34, 35] makes the connection explicit, opting for execution within Isabelle/HOL's rewrite engine. Our prover stands out from these in two ways. Firstly, it is an interactive theorem prover where users can employ techniques of declarative proving, automation, tactics, etc. as they wish. Secondly, the connection between the verification and the system is made explicit using code-generation.

There are many other formalizations of logic such as e.g. Persson's constructive completeness of intuitionistic predicate logic [33], Braselmann, Koepke and Schlöder's sequent calculus for uncountable languages [4, 5, 38], Berghofer's natural deduction [1], Ilik's constructive completeness results for classical and intuitionistic logic [15], Blanchette, Popescu and Traytel's abstract completeness library [3], Schlichtkrull's resolution calculus [36], Peltier's superposition calculus [32], and Paulson's proof of Gödel's incompleteness theorems [30]. These formalizations, however, do not formalize provers.

Let us now look at some other declarative provers. Geuvers [8] studies the history, ideas and future of proof assistants. For instance, he emphasizes the advantage of having declarative provers since they allow proofs in proof assistants to look like the texts that mathematicians write and understand. Furthermore he emphasizes that declarative proofs are easier to adapt when a definition is changed, since they explicitly document in each step which facts are supposed to hold there. He also gives an overview of declarative proofs in the proof assistants Mizar, Isabelle/Isar, Coq/C-Zar, and HOL Light (Mizar mode).

Our prover stands out from these provers in that it relies on a verified kernel. Furthermore, it is not meant as an advanced production scale proof assistant, but instead as a smaller program that is easy to understand and whose inner workings can be taught. None the less, the prover still has the advantages of being declarative that Geuvers described.

14 Conclusion

We have in Isabelle/HOL certified the soundness of the underlying axiomatic proof system of the declarative first-order prover. Using code reflection, we obtain from the proof system a certified kernel that is loaded into the Isabelle/ML environment. The declarative prover uses the certified kernel, and thus we also consider the soundness of the prover certified.

Declarative proofs mention explicitly the intermediate proof states, in contrast to procedural proofs that merely explain what method is used to go to the next state. We have given example proofs using the prover in Isabelle. Due to the compactness and transparent approach we think that the certified declarative prover is useful as a tool for teaching logic.

Many well-known theorems can be proved by full automation using the tableau prover, e.g. Pelletier's problems 1–46 except for problems 34 (also known as Andrews's challenge), 43 and 46.

For problem 43 and 46 that could not be proved automatically in reasonable time with the current tableau prover, we have shown how the proofs can be written as declarative proofs that resemble paper proofs and combine the declarative language with a high level of automation. It would be interesting to improve the tableau prover or to add, say, a resolution prover, which would be certified by using the certified kernel. We have not considered the tricky problem 34 yet.

Our declarative prover follows the LCF-style of having a trusted kernel on which other components are built. In a single Isabelle theory file we certify the soundness of the kernel and use code reflection to obtain a simple yet quite powerful interactive theorem prover. Our combination of derived rules, a tableau prover, tactics and a declarative prover opens up for easy experimentation with reliable combinations of automatic and interactive proof techniques — and such techniques are in high demand as the following quote suggests:

> *In view of the practical limitations of pure automation, it seems today that, whether one likes it or not, interactive proof is likely to be the only way to formalize most non-trivial theorems in mathematics or computer system correctness.* [14]

Learning the declarative style is of course beneficial for a computer science student who wants to use one of the aforementioned provers. Even for those who will never use a proof assistant again, it can be a helpful learning experience. Lamport recommends a structured style even for paper proofs [21,22]. His experience is that this style helps reveal mistakes and cope with details. He also suggests using this style for teaching because it allows for additional explanation and has a clear logical structure that is easy to learn from. The concrete style he uses resembles very much that of our declarative prover. Furthermore, the style is implemented in the TLAPS prover [22]. We conjecture that a good way to learn this structured style is by studying and understanding a concrete prover. Our prover emphasizes the connection between the logical systems, its semantics and the prover that implements them. A student can study all these aspects in the package we provide.

Acknowledgements We would like to thank Jasmin Christian Blanchette for valuable feedback on the workshop paper [17] on which parts of this paper are based. We would also like to thank Andrei Popescu for helpful discussions, and finally, we would like to thank Andreas Halkjær From and the anonymous referees for constructive comments on drafts of the paper.

References

[1] S. Berghofer. First-order logic according to Fitting. *Archive of Formal Proofs*, Aug. 2007. `http://isa-afp.org/entries/FOL-Fitting.shtml`, Formal proof development.

[2] J. C. Blanchette, S. Böhme, and L. C. Paulson. Extending Sledgehammer with SMT solvers. In N. Bjørner and V. Sofronie-Stokkermans, editors, *CADE-23*, pages 116–130. Springer, 2011.

[3] J. C. Blanchette, A. Popescu, and D. Traytel. Soundness and completeness proofs by coinductive methods. *Journal of Automated Reasoning*, 58(1):149–179, 2017.

[4] P. Braselmann and P. Koepke. Gödel's completeness theorem. *Formalized Mathematics*, 13(1):49–53, 2005.

[5] P. Braselmann and P. Koepke. A sequent calculus for first-order logic. *Formalized Mathematics*, 13(1):33–39, 2005.

[6] J. Breitner. Visual theorem proving with the Incredible Proof Machine. In J. C. Blanchette and S. Merz, editors, *International Conference on Interactive Theorem Proving*, volume 9807 of *LNCS*, pages 123–139. Springer, 2016.

[7] A. Church. *Introduction to Mathematical Logic.* Princeton: Princeton University Press, 1956.

[8] H. Geuvers. Proof assistants: History, ideas and future. *Sadhana*, 34(1):3–25, 2009.

[9] M. J. Gordon, A. J. Milner, and C. P. Wadsworth. *Edinburgh LCF – A Mechanised Logic of Computation*, volume 78 of *LNCS*. Springer, 1979.

[10] F. Haftmann and T. Nipkow. Code generation via higher-order rewrite systems. In M. Blume, N. Kobayashi, and G. Vidal, editors, *Functional and Logic Programming (FLOPS 2010)*, volume 6009 of *LNCS*, pages 103–117. Springer, 2010.

[11] R. Hähnle. Tableaux and related methods. *Handbook of Automated Reasoning*, 1(101-176):4, 2001.

[12] J. Harrison. Towards self-verification of HOL Light. In U. Furbach and N. Shankar, editors, *IJCAR 2006*, volume 4130 of *LNCS*, pages 177–191. Springer, 2006.

[13] J. Harrison. *Handbook of Practical Logic and Automated Reasoning.* Cambridge University Press, 2009.

[14] J. Harrison, J. Urban, and F. Wiedijk. History of interactive theorem proving. In J. Siekmann, editor, *Handbook of the History of Logic vol. 9 (Computational Logic)*, pages 135–214. Elsevier, 2014.

[15] D. Ilik. *Constructive Completeness Proofs and Delimited Control.* PhD thesis, École Polytechnique, 2010.

[16] A. B. Jensen. Development and verification of a proof assistant. Master's thesis, Technical University of Denmark, 2016.

[17] A. B. Jensen, A. Schlichtkrull, and J. Villadsen. Verification of an LCF-style first-order prover with equality. *Isabelle Workshop*, 2016.

[18] A. B. Jensen, A. Schlichtkrull, and J. Villadsen. `https://bitbucket.org/isafol/isafol/src/master/FOL_Harrison/`, 2017. IsaFoL Entry – First-Order Logic According to Harrison.

[19] A. B. Jensen, A. Schlichtkrull, and J. Villadsen. First-order logic according to Harrison. *Archive of Formal Proofs*, Jan. 2017. `http://isa-afp.org/entries/FOL_Harrison.shtml`,Formal proof development.

[20] R. Kumar, R. Arthan, M. O. Myreen, and S. Owens. Self-formalisation of higher-order logic: Semantics, soundness, and a verified implementation. *Journal of Automated Reasoning*, 56(3):221–259, 2016.

[21] L. Lamport. How to write a proof. *Global Analysis in Modern Mathematics*, pages 311–321, 1993. Also published in *American Mathematical Monthly*, 102(7):600-608, August-September 1995.

[22] L. Lamport. How to write a 21st century proof. *Journal of fixed point theory and applications*, 11(1):43–63, 2012.

[23] R. Letz. First-order tableau methods. In M. D'Agostino, D. M. Gabbay, and R. Hähnle, editors, *Handbook of Tableau Methods*, pages 125–196. Kluwer Academic Publishers, 1999.

[24] J. Margetson and T. Ridge. Completeness theorem. *Archive of Formal Proofs*, Sept. 2004. `http://isa-afp.org/entries/Completeness.shtml`, Formal proof development.

[25] N. D. Megill. *Metamath: A Computer Language for Pure Mathematics.* Lulu Press, Morrisville, North Carolina, 2007. `http://us.metamath.org/downloads/metamath.pdf`.

[26] R. Milner. LCF: A way of doing proofs with a machine. In J. Bečvář, editor, *Mathematical Foundations of Computer Science*, volume 74 of *LNCS*, pages 146–159. Springer, 1979.

[27] R. Milner, M. Tofte, and D. Macqueen. *The Definition of Standard ML.* MIT Press, Cambridge, MA, USA, 1997.

[28] J. Monk. *Mathematical Logic.* Graduate Texts in Mathematics. Springer New York, 1976.

[29] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS.* Springer, 2002.

[30] L. C. Paulson. A mechanised proof of Gödel's incompleteness theorems using Nominal Isabelle. *Journal of Automated Reasoning*, 55(1):1–37, 2015.

[31] F. J. Pelletier. Seventy-five problems for testing automatic theorem provers. *Journal of Automated Reasoning*, 2(2):191–216, 1986.

[32] N. Peltier. A variant of the superposition calculus. *Archive of Formal Proofs*, Sept. 2016. `http://isa-afp.org/entries/SuperCalc.shtml`, Formal proof development.

[33] H. Persson. *Constructive completeness of intuitionistic predicate logic*. PhD thesis, Chalmers University of Technology, 1996.

[34] T. Ridge. A mechanically verified, efficient, sound and complete theorem prover for first order logic. *Archive of Formal Proofs*, Sept. 2004. `http://isa-afp.org/entries/Verified-Prover.shtml`, Formal proof development.

[35] T. Ridge and J. Margetson. A mechanically verified, sound and complete theorem prover for first order logic. In J. Hurd and T. Melham, editors, *TPHOL's 2005*, volume 3603 of *LNCS*, pages 294–309. Springer, 2005.

[36] A. Schlichtkrull. Formalization of the resolution calculus for first-order logic. In J. C. Blanchette and S. Merz, editors, *International Conference on Interactive Theorem Proving*, volume 9807 of *LNCS*, pages 341–357. Springer, 2016.

[37] A. Schlichtkrull and J. Villadsen. `https://github.com/logic-tools/sml-handbook/tree/master/code`, 2017. README: SML version of code for John Harrison's "Handbook of Practical Logic and Automated Reasoning".

[38] J. J. Schlöder and P. Koepke. The Gödel completeness theorem for uncountable languages. *Formalized Mathematics*, 20(3):199–203, 2012.

[39] A. Tarski. A simplified formalization of predicate logic with identity. *Journal of Symbolic Logic*, 39(3):602–603, 1974.

[40] J. Villadsen, A. B. Jensen, and A. Schlichtkrull. NaDeA: A natural deduction assistant with a formalization in Isabelle. *IfCoLog Journal of Logics and their Applications*, 4(1):55–82, 2017.

[41] M. Wenzel. Isar - A generic interpretative approach to readable formal proof documents. In Y. Bertot, G. Dowek, L. Théry, A. Hirschowitz, and C. Paulin, editors, *Theorem Proving in Higher Order Logics*, volume 1690 of *LNCS*, pages 167–184, 1999.

[42] M. Wenzel. System description: Isabelle/jEdit in 2014. In *Proceedings Eleventh Workshop on User Interfaces for Theorem Provers, UITP 2014, Vienna, Austria, 17th July 2014*, pages 84–94, 2014.

[43] M. Wenzel. The Isabelle/Isar reference manual, 2016. `http://isabelle.in.tum.de/dist/doc/isar-ref.pdf`.

[44] M. Wenzel. Isabelle/jEdit, 2016. `http://isabelle.in.tum.de/dist/doc/jedit.pdf`.

[45] Wikipedia. Method of analytic tableaux, 2017. `https://en.wikipedia.org/wiki/Method_of_analytic_tableaux`.

Formalized Meta-Theory of a Paraconsistent Logic

Anders Schlichtkrull

DTU Compute, Technical University of Denmark, Kongens Lyngby, Denmark

Abstract

Classical logics are explosive, meaning that everything follows from a contradiction. Paraconsistent logics are logics that are not explosive. This chapter presents the meta-theory of a paraconsistent infinite-valued logic, in particular showing that while the question of validity for a given formula can be reduced to a consideration of only finitely many truth values, this does not mean that the logic collapses to a finite-valued logic. All definitions and theorems are formalized in the Isabelle/HOL proof assistant.

1 Introduction

The previous chapters in the thesis considered classical logics, which have many applications including reasoning about mathematics, computer science and engineering. Classical logics are by design *explosive* – everything follows from a contradiction. This is mostly uncontroversial, but it seems problematic for certain kinds of reasoning. In paraconsistent logics, everything does not follow from a contradiction. Non-classical logics should also enjoy the benefits of formalization, and therefore this chapter presents a formalization of a paraconsistent infinite-valued propositional logic.

The entry on paraconsistent logic in the Stanford Encyclopedia of Philosophy [10] thoroughly motivates paraconsistent logics by arguing that some domains do contain inconsistencies, but this should not make meaningful reasoning impossible. An example from computer science is that in large knowledge bases an inconsistency can easily occur if just one data point is entered wrong. A reasoning system based on such a database needs a meaningful way to deal with the inconsistency. Many other examples are mentioned from philosophy, linguistics, automated reasoning and mathematics. A recent book [1] looks at paraconsistency in the domain of engineering. There is no one paraconsistent logic to rule them all – there are many logics which can be used in different contexts. The encyclopedia gives a taxonomy of paraconsistent logics consisting of discussive logics, non-adjunctive systems, preservationism, adaptive logics, logics of formal inconsistency, relevant logics and many-valued logics.

The logic considered here is the propositional fragment of a paraconsistent infinite-valued higher-order logic by Villadsen [15, 16, 17, 18] and more recently Jensen and Villadsen in an extended abstract [7]. The propositional logic, here called $\mathbb{V}$, has a semantics with the two classical truth values and countably infinitely many non-classical truth values. When U is a subset of the non-classical truth values, $\mathbb{V}_U$ is the logic defined the same as $\mathbb{V}$ except for the restriction that its non-classical truth values are only those in U. In $\mathbb{V}_U$ with a finite U, one can find out whether a formula p is valid by enumerating enough interpretations that they cover all possible assignments of the propositional symbols in p. This approach does not work in $\mathbb{V}$ since there are infinitely many such interpretations. This chapter shows that it is enough to consider the models in $\mathbb{V}_U$ for a finite U, but that the size of U depends on the formula considered.

The contents of this chapter are as follows:

- Section 2 defines and formalizes $\mathbb{V}$. It gives an example of paraconsistency in the logic.
- Section 3 defines and formalizes $\mathbb{V}_U$.
- Section 4 proves and formalizes that for any formula p there is a finite U such that if p is valid in $\mathbb{V}_U$, it is also valid in $\mathbb{V}$. This allows the question of validity in $\mathbb{V}$ to be solved by a finite enumeration of interpretations.
- Section 5 proves and formalizes that if $|U| = |W|$, then $\mathbb{V}_U$ and $\mathbb{V}_W$ consider the same formulas valid.
- Section 6 shows that, to answer the question of validity in $\mathbb{V}$, one cannot fix a finite valued $\mathbb{V}_U$ once and for all because there exists a formula $\pi_{|U|}$ that is valid in this logic but not in $\mathbb{V}$. In other words, despite the result in Section 4, $\mathbb{V}$ is a different logic than any finite valued $\mathbb{V}_U$.

The formalization in Sections 2 and 3 was written as part of my PhD studies and previously presented in a book chapter [19] and a paper [20] by Villadsen and myself. The result in Section 4 had already been conjectured by Jensen and Villadsen [7], but was first proved and formalized as part of my PhD studies as presented in the mentioned book chapter [19] and paper [20]. Likewise, the result in Section 6 was conjectured by Villadsen and myself [20] and was first proved and formalized as part of my PhD studies. It is presented in writing here for the first time except for a brief mention in the abstract of a talk [11]. Thus, there are no previous informal proofs to refer to for these results, and this chapter will therefore both present the formalization of these results and their informal proofs. The full formalization is available online – 1500 lines of code are already in an *Archive of Formal Proofs* entry by Villadsen and myself [13], and the 800 lines corresponding to Sections 5 and 6 [12] will be added later. To make the chapter easier to read, its notation is slightly different from the formalization.

2 A Paraconsistent Infinite-Valued Logic

The paraconsistent infinite-valued propositional logic $\mathbb{V}$ has two classical truth values, namely true ($\bullet$) and false ($\circ$). These are called the determinate truth values. It also has countably many different non-classical truth values (ı, ıı, ııı, . . .) [7]. These are called the inderterminate truth values. This is represented as a datatype *tv*.

datatype *tv* = *Det bool* | *Indet nat*

Det True and *Det False* represent $\bullet$ and $\circ$ respectively, and constructor *Indet* maps each natural number (0, 1, 2, . . .) to the corresponding indeterminate truth value (ı, ıı, ııı, . . .).

The propositional symbols of $\mathbb{V}$ are strings of a finite alphabet. Here, the symbols are denoted as $\mathsf{p}, \mathsf{q}, \mathsf{r}, \ldots$. Interpretations are functions from propositional symbols into truth values. The formulas of the logic are built from the propositional symbols and operators $\boldsymbol{\neg}$, $\boldsymbol{\wedge}$, $\boldsymbol{\Leftrightarrow}$ and $\boldsymbol{\leftrightarrow}$ as well as a symbol for truth $\boldsymbol{\top}$. To make them distinguishable, the logical operators in the paraconsistent logic are bold, while Isabelle/HOL's logical operators are not (e.g. $\neg$, $\wedge$, $\vee$, $\longleftrightarrow$). $\boldsymbol{\Leftrightarrow}$ represents equality whereas $\boldsymbol{\neg}$, $\boldsymbol{\wedge}$ and $\boldsymbol{\leftrightarrow}$ are designed to be generalizations of their classical counterparts. In the Isabelle/HOL formalization, the formulas are defined by a datatype *fm*, with a constructor for atomic formulas consisting of propositional symbols and with constructors for each of the operators.

In the semantics, Villadsen motivated the different cases by equalities of classical logic that also hold in $\mathbb{V}$ [15]. These motivating equalities are shown to the right of their case:

$$\begin{aligned} \mathit{eval}\ i\ x &= i\ x \text{ if } x \text{ is a propositional symbol} \\ \mathit{eval}\ i\ \top &= \bullet \end{aligned}$$

$$\mathit{eval}\ i\ (\neg\, p) = \begin{cases} \bullet & \text{if } \mathit{eval}\ i\ p = \circ \qquad \top \Leftrightarrow \neg\, \bot \\ \circ & \text{if } \mathit{eval}\ i\ p = \bullet \qquad \bot \Leftrightarrow \neg\, \top \\ \mathit{eval}\ i\ p & \text{otherwise} \end{cases}$$

$$\mathit{eval}\ i\ (p \wedge q) = \begin{cases} \mathit{eval}\ i\ p & \text{if } \mathit{eval}\ i\ p = \mathit{eval}\ i\ q \qquad p \Leftrightarrow p \wedge p \\ \mathit{eval}\ i\ q & \text{if } \mathit{eval}\ i\ p = \bullet \qquad q \Leftrightarrow \top \wedge q \\ \mathit{eval}\ i\ p & \text{if } \mathit{eval}\ i\ q = \bullet \qquad p \Leftrightarrow p \wedge \top \\ \circ & \text{otherwise} \end{cases}$$

$$\mathit{eval}\ i\ (p \Leftrightarrow q) = \begin{cases} \bullet & \text{if } \mathit{eval}\ i\ p = \mathit{eval}\ i\ q \\ \circ & \text{otherwise} \end{cases}$$

$$\mathit{eval}\ i\ (p \leftrightarrow q) = \begin{cases} \bullet & \text{if } \mathit{eval}\ i\ p = \mathit{eval}\ i\ q \qquad \top \Leftrightarrow p \leftrightarrow p \\ \mathit{eval}\ i\ q & \text{if } \mathit{eval}\ i\ p = \bullet \qquad q \Leftrightarrow \top \leftrightarrow q \\ \mathit{eval}\ i\ p & \text{if } \mathit{eval}\ i\ q = \bullet \qquad p \Leftrightarrow p \leftrightarrow \top \\ \mathit{eval}\ i\ (\neg\, q) & \text{if } \mathit{eval}\ i\ p = \circ \qquad \neg q \Leftrightarrow \bot \leftrightarrow q \\ \mathit{eval}\ i\ (\neg\, p) & \text{if } \mathit{eval}\ i\ q = \circ \qquad \neg\, p \Leftrightarrow p \leftrightarrow \bot \\ \circ & \text{otherwise} \end{cases}$$

Additionally, a number of derived operators are defined:

$$\begin{array}{ll} \bot \equiv \neg\, \top & \neg\!\!\neg\, p \equiv \Box\, (\neg\, p) \\ p \vee q \equiv \neg\, (\neg\, p \wedge \neg\, q) & p \mathbin{\wedge\!\!\!\wedge} q \equiv \Box\, (p \wedge q) \\ p \Rightarrow q \equiv p \Leftrightarrow (p \wedge q) & p \mathbin{\vee\!\!\!\vee} q \equiv \Box\, (p \vee q) \\ p \rightarrow q \equiv p \leftrightarrow (p \wedge q) & \boldsymbol{\Delta}\, p \equiv (\Box\, p) \mathbin{\vee\!\!\!\vee} (p \Leftrightarrow \bot) \\ \Box\, p \equiv p \Leftrightarrow \top & \boldsymbol{\nabla}\, p \equiv \neg\!\!\neg\, (\boldsymbol{\Delta}\, p) \end{array}$$

Of special interest are $\Box$, $\boldsymbol{\Delta}$ and $\boldsymbol{\nabla}$. $\Box$ maps $\bullet$ to $\bullet$ and any other value to $\circ$. In other words $\Box\, p$ states "p is true". Similarly $\boldsymbol{\Delta}\, p$ states "p is determinate" and $\boldsymbol{\nabla}\, p$ states "p is indeterminate".

The other operators can be divided in two groups – general operators ($\neg$, $\wedge$, $\vee$ and $\leftrightarrow$) and purely determinate operators ($\neg\!\!\neg$, $\wedge\!\!\!\wedge$, $\vee\!\!\!\vee$ and $\Leftrightarrow$). The general operators behave as expected on determinate values, and this behavior is generalized to indeterminate values. Consider for example the truth table for $\vee$:

∨	●	○	∣	∥
●	●	●	●	●
○	●	○	∣	∥
∣	●	∣	∣	●
∥	●	∥	●	∥

The purely determinate operators also behave as expected on determinate values, and their behavior generalizes to indeterminate values – however this time in such a way that they always return a determinate truth value. Consider for example the truth table for ⩔:

⩔	●	○	∣	∥
●	●	●	●	●
○	●	○	○	○
∣	●	○	○	●
∥	●	○	●	○

Validity is defined in the usual way, i.e. a formula is valid if it is true in all interpretations.

definition *valid* :: *"fm ⇒ bool"*
where
  "valid p ≡ ∀ i. eval i p = ●"

As a simple example of paraconsistency, consider the formula (p ∧ (¬ p)) ⇒ q. This formula is not valid since it has e.g. the counter-model mapping p to ∣ and q to ○.

3 Paraconsistent Finite-Valued Logics

For any set U of indeterminate truth values, the logic $\mathbb{V}_U$ is defined as follows: $\mathbb{V}_U$ is defined in the same way as $\mathbb{V}$, except that it has a different notion of interpretations. An interpretation in $\mathbb{V}_U$ is a function from propositional symbols to the set $\{\bullet, \circ\} \cup U$ instead of to the type of all truth values.

A function *domain* constructs $\{\bullet, \circ\} \cup U$ from a set of natural numbers:

definition *domain* :: *"nat set ⇒ tv set"*
where
  "domain U ≡ {Det True, Det False} ∪ Indet ` U"

Here, *Indet ` U* denotes the image of *Indet* on U. Notice that in the formalization, U is a set of natural numbers rather than a set of indeterminate values. This is only because it is less tedious to write $\{0, 1, 2\}$ than $\{\mathit{Indet}\ 0, \mathit{Indet}\ 1, \mathit{Indet}\ 2\}$ and because being able to write *domain* $\{\mathit{Indet}\ 0, \bullet\}$ is rather pointless since $\bullet$ is added by *domain* anyway. For the same reasons, I will from now on also write e.g. $\mathbb{V}_{\{0,1,2\}}$ rather than $\mathbb{V}_{\{\mathit{Indet}\ 0,\ \mathit{Indet}\ 1,\ \mathit{Indet}\ 2\}}$. The function is called *domain* because in the higher-order version of $\mathbb{V}$ one can use the truth values as the domain of discourse.

The notion of being valid in $\mathbb{V}_U$ is formalized. The expression *range i* denotes the function range of i.

definition *valid_in* :: *"nat set ⇒ fm ⇒ bool"*
where
  "valid_in U p ≡ ∀ i. range i ⊆ domain U ⟶ eval i p = ●"

It is clear that validity in $\mathbb{V}$ implies validity in any $\mathbb{V}_U$.

theorem *valid_valid_in*: **assumes** *"valid p"* **shows** *"valid_in U p"*

Proof. If p is valid in $\mathbb{V}$, it is true in all interpretations and thus in particular those with the desired range. Therefore p is valid in $\mathbb{V}_U$. □

The set U can be finite or infinite. The former case in particular will be of interest in the following sections.

4 A Reduction from Validity in $\mathbb{V}$ to Validity in $\mathbb{V}_U$

When U is finite, one can find out if a formula is valid by considering all the different cases of what an interpretation might map the formula's propositional symbols to. As an example, consider the formula $(\mathsf{p} \wedge (\neg\, \mathsf{p})) \longrightarrow \mathsf{q}$ in the logic $\mathbb{V}_\emptyset$, which corresponds to classical propositional logic.

```
proposition "valid_in ∅ ((p ∧ (¬ p)) ⟶ q)"
  unfolding valid_in_def
proof (rule; rule)
  fix i :: "id ⇒ tv"
  assume "range i ⊆ domain ∅"
  then have
      "i p ∈ {•, ∘}"
      "i q ∈ {•, ∘}"
    unfolding domain_def
    by auto
  then show "eval i ((p ∧ (¬ p)) ⟶ q) = •"
    by (cases "i p"; cases "i q") auto
qed
```

For $\mathbb{V}$ this approach does not work, since there are infinitely many truth values. This section overcomes the problem by showing that there exists a finite subset of the interpretations in $\mathbb{V}_U$ that it is enough to enumerate. The idea is that looking at the semantics of $\mathbb{V}$ reveals that there is a lot of symmetry between the indeterminate truth values $\mathsf{I}, \mathsf{II}, \mathsf{III}, \ldots$. Specifically, the indeterminate values are all different and can be told apart using $\Leftrightarrow$, but none of them play any special role compared with the others. Intuitively, this means that one just needs to consider enough interpretations to ensure that one has considered all different possibilities of interpreting the different pairs of propositional symbols as either different or equal indeterminate truth values. Therefore it is only necessary to consider enough truth values to ensure that this is possible and thus, for any formula p, it should be sufficient to consider all the interpretations in the logic $\mathbb{V}_U$, where $|U|$ is at least the number of propositional symbols in p.

The first step towards proving this is to prove that interpretations that agree on the propositional symbols occurring in a formula also evaluate the formula to the same result.

lemma *relevant_props*: **assumes** *"$\forall s \in props\ p.\ i_1\ s = i_2\ s$"* **shows** *"$eval\ i_1\ p = eval\ i_2\ p$"*

Proof. Follows by induction on the formula and the definitions of *props* and *eval*. □

The next step is to consider an interpretation i in $\mathbb{V}$ and see that it behaves the same as a corresponding interpretation in $\mathbb{V}_U$. The idea is that i can be changed to an interpretation in $\mathbb{V}_U$ by applying a function from *nat* into U to the indeterminate values that the interpretation returns.

Given a function f of type $nat \Rightarrow nat$ and an interpretation, its application $f\ x$ to a truth value x is defined as

$$f\ x = \begin{cases} x & \text{if } x \text{ is determinate} \\ \mathit{Indet}\ (f\ n) & \text{if } x = \mathit{Indet}\ n \end{cases}$$

A function can also be applied to an interpretation:

$$f\ i = \lambda s.\ f\ (i\ s)$$

If f is an injection, then applying f to the result or to the interpretation gives the same result when evaluating a formula.

lemma *eval_change*: **assumes** *"inj f"* **shows** *"eval (f i) p = f (eval i p)"*

Proof. The proof is by induction on p. For each logical symbol, consider the different cases of what the subformulas evaluate to under i as specified in the semantics. Consider for instance the semantics' "otherwise"-case for $p \leftrightarrow q$. Here, it is the case that $\mathit{eval}\ i\ p \neq \mathit{eval}\ i\ q$ and that there exists a natural number n such that $\mathit{eval}\ i\ p = \mathit{Indet}\ n$ and some m such that $\mathit{eval}\ i\ q = \mathit{Indet}\ m$. Hence $\mathit{Indet}\ n \neq \mathit{Indet}\ m$ and therefore $n \neq m$. Since f is injective, also $f\ n \neq f\ m$ and $\mathit{Indet}\ (f\ n) \neq \mathit{Indet}\ (f\ m)$. The induction hypotheses are $\mathit{eval}\ (f\ i)\ p = f\ (\mathit{eval}\ i\ p)$ and $\mathit{eval}\ (f\ i)\ q = f\ (\mathit{eval}\ i\ q)$. Consider the first one. Here it is the case that $\mathit{eval}\ (f\ i)\ p = f\ (\mathit{eval}\ i\ p) = f\ (\mathit{Indet}\ n) = \mathit{Indet}\ (f\ n)$. Likewise from the second it follows that $\mathit{eval}\ (f\ i)\ q = f\ (\mathit{eval}\ i\ q) = f\ (\mathit{Indet}\ m) = \mathit{Indet}\ (f\ m)$. This implies that $\mathit{eval}\ (f\ i)\ p \neq \mathit{eval}\ (f\ i)\ q$. Then, by the semantics, it follows that $\mathit{eval}\ (f\ i)\ (p \leftrightarrow q) = \circ = f\ \circ = f\ (\mathit{eval}\ i\ (p \leftrightarrow q))$. And this part of the proof is done. This was just one out of the 17 cases in the semantics. For the rest I refer to the formalization. □

Writing out all 17 cases would be tedious and checking all of them by hand requires discipline. Therefore, there is always the danger of overlooking a needed argument, because one case looked similar to another but really was not. Formalization enforces this discipline.

Now it is time to prove that if there are at least as many indeterminate truth values in U as the number of propositional symbols in p, then the validity of p in $\mathbb{V}_U$ implies the validity of p in $\mathbb{V}$. The lemma is expressed using Isabelle/HOL's *card* function, which for finite sets returns their cardinality and for infinite sets returns 0.

theorem *valid_in_valid*: **assumes** *"card U ≥ card (props p)"* **and** *"valid_in U p"* **shows** *"valid p"*

Proof. p is proved valid by fixing an arbitrary interpretation i: First, obtain an injection f of type $nat \Rightarrow nat$ such that f maps any value in i ‘ $(\mathit{props}\ p)$ to a value in $\mathit{domain}\ U$. This is possible because $|\mathit{domain}\ U| \geq |\mathit{props}\ p|$.

Now define the following interpretation:

$$i'\ s = \begin{cases} (f\ i)\ s & \text{if } s \in \mathit{props}\ p \\ \bullet & \text{otherwise} \end{cases}$$

From the properties of f and definition of i' it follows that $\mathit{range}\ i' \subseteq \mathit{domain}\ U$ and then by the validity of p in U it follows that $\mathit{eval}\ i'\ p = \bullet$. Furthermore, i' and $f\ i$ coincide on all symbols in p, and therefore, by the lemma *relevant_props*, it also follows that $\mathit{eval}\ (f\ i)\ p = \bullet$. Now from *eval_change* follows that $f\ (\mathit{eval}\ i\ p) = \bullet$. By definition of the application of a $nat \Rightarrow nat$ to a truth-value it is the case that $\mathit{eval}\ i\ p = \bullet$. Thus any interpretation evaluates to $\bullet$ and therefore the formula is valid. □

theorem *valid_iff_valid_in*:
  assumes *"card U ≥ card (props p)"*
  shows *"valid p ⟷ valid_in U p"*

Proof. Follows from *valid_valid_in* and *valid_in_valid*. □

5 Sets of Equal Cardinality Define the Same Logic

Recall that while the indeterminate values are all different and can be told apart using $\Leftrightarrow$, none of them play any special role compared to the others. Therefore one would expect $\mathbb{V}_U$ and $\mathbb{V}_W$ to be the same when U and W have the same cardinality. In the same way, consider what happens when $|U| < |W|$. In this case one can think of $\mathbb{V}_U$ as being $\mathbb{V}_W$ with some truth values, and thus interpretations, removed. Removing interpretations only makes it easier for a formula to be valid and thus any formula that is valid in $\mathbb{V}_W$ should also be valid in $\mathbb{V}_U$.

Isabelle/HOL defines *inj_on* such that *inj_on f A* expresses that f is an injection from A into the return type of f. In order to be able to talk about one set having smaller cardinality than another, it is useful to also define the notion of an injection from a set into another set.

definition *inj_from_to* :: *"('a ⇒ 'b) ⇒ 'a set ⇒ 'b set ⇒ bool"* **where**
  "inj_from_to f X Y ≡ inj_on f X ∧ f ‘ X ⊆ Y"

The lemma *eval_change* is generalized from the type *nat* to sets of *nat*s.

lemma *eval_change_inj_on*:
  assumes *"inj_on f U"*
  assumes *"range i ⊆ domain U"*
  shows *"eval (f i) p = f (eval i p)"*

Proof. The proof is analogous to that of *eval_change*. □

This is enough to prove the following lemma:

lemma *inj_from_to_valid_in*:
  assumes *"inj_from_to f W U"*
  assumes *"valid_in U p"*
  shows *"valid_in W p"*

Proof. The plan is to fix an arbitrary interpretation in $\mathbb{V}_W$ and prove that it makes p true. First, realize that *range* $(f\ i) \subseteq$ *domain* U; this follows from the fact that for any x it is the case that $(f\ i)\ x = f\ (i\ x)$ and here the application of i will give an element in *domain* W and then the application of f will give an element in *domain* U. Therefore *eval* $(f\ i)\ p = \bullet$ by the validity of p in $\mathbb{V}_U$. Then use *eval_change_inj_on* to get that f (*eval* $i\ p) = \bullet$ and then from the definition of the application of f to a truth value that *eval* $i\ p = \bullet$. □

It is now time to prove that if U and W have equal cardinality, they define the same logic.

lemma *bij_betw_valid_in*:
  assumes *"bij_betw f U W"*
  shows *"valid_in U p ⟷ valid_in W p"*

Proof. f is an injection from U into W. f^- is an injection from W into U. The lemma therefore follows from *inj_from_to_valid_in*. □

6 The Difference Between $\mathbb{V}$ and $\mathbb{V}_U$ for a Finite U

Section 4 showed that the question of the validity of p in $\mathbb{V}$ can be reduced to the question of its validity in $\mathbb{V}_{\{0..<|prop\ p|\}}$, where $\{n..<m\} = \{k \mid n \leq k < m\}$ for any n and m. This section shows that this does not mean that $\mathbb{V}$ collapses to a finite valued $\mathbb{V}_U$. The approach is to demonstrate a formula that is true in $\mathbb{V}_{0..n}$ but false in $\mathbb{V}$. The formula is denoted the pigeonhole formula. For $n = 3$ the pigeonhole formula π_3 is

$$\pi_3 = \boldsymbol{\nabla}\mathsf{x}_0 ⩕ \boldsymbol{\nabla}\mathsf{x}_1 ⩕ \boldsymbol{\nabla}\mathsf{x}_2 \Rightarrow (\mathsf{x}_0 \Leftrightarrow \mathsf{x}_1) ⩖ (\mathsf{x}_0 \Leftrightarrow \mathsf{x}_2) ⩖ (\mathsf{x}_0 \Leftrightarrow \mathsf{x}_1).$$

I.e. it states that, assuming that x_0, x_1 and x_2 refer to indeterminate values, two of them will be the same. This is of course not true in an interpretation where they map to three different values, but if one only considers two indeterminate values there are no such interpretations. Therefore the formula is not valid in general but it is valid in $\mathbb{V}_{\{\mathsf{I},\,\mathsf{II}\}}$. Propositions x_0 and x_1 and x_2 can be thought of as pigeons and the values I and II as pigeon holes.

In order to define the formula for any n, first define the conjunction and disjunction of a list of formulas:

$$[⩕]_{p_1,\ldots,p_n} = p_1 ⩕ \cdots ⩕ p_n$$

$$[⩖]_{p_1,\ldots,p_n} = p_1 ⩖ \cdots ⩖ p_n$$

Extend $\boldsymbol{\nabla}$to a symbol that characterizes lists of indeterminate values:

$$[\boldsymbol{\nabla}]_{p_1,\ldots,p_n} = [⩕]_{\boldsymbol{\nabla}p_1,\ldots,\boldsymbol{\nabla}p_n}$$

Given two sets S_1 and S_2, the concept of their cartesian product $S_1 \times S_2$ is well known. Their *off-diagonal product* is defined as

$$S_1 \times_{\text{off-diag}} S_2 = \{(s_1, s_2) \in S_1 \times S_2 \mid s_1 \neq s_2\}$$

Isabelle/HOL offers the function *List.product* of type $'a\ list \Rightarrow\ 'a\ list \Rightarrow (\,'a \times\, 'a)\ list$, which implements the cartesian product on lists representing sets. From this the *list off-diagonal product* is defined:

$$L_1 \times_{\text{off-diag}} L_2 = \mathit{filter}\ (\lambda(x, y).\ x \neq y)\ (\mathit{List.product}\ L_1 L_2)$$

The list off-diagonal product is used to introduce equivalence existence, which given a list of formulas expresses that two of the formulas in the list are equivalent.

$$[\exists =]_{p_1,\ldots,p_n} = [⩖][=]((p_1, \ldots, p_n) \times_{\text{off-diag}} (p_1, \ldots, p_n))$$

where

$$[=]_{(p_{11},p_{12}),\ldots,(p_{n1},p_{n2})} = p_{11} \Leftrightarrow p_{12}, \ldots, p_{n1} \Leftrightarrow p_{n2}$$

Let $\mathsf{x}_0, \mathsf{x}_1, \mathsf{x}_2, \ldots$ be a sequence of different variables. These will form the pigeon holes. Implication, $\boldsymbol{\nabla}$, equivalence existence and the pigeon holes are combined to form the pigeonhole formula:

$$\pi_n = [\boldsymbol{\nabla}]_{\mathsf{x}_0,\cdots,\mathsf{x}_{n-1}} \Rightarrow [\exists =]_{\mathsf{x}_0,\cdots,\mathsf{x}_{n-1}}$$

6.1 π_n is not valid in $\mathbb{V}$

In order to prove that the pigeonhole formula is not valid, a counter-model for it is demonstrated. This counter-model is in $\mathbb{V}_{\{0..<n\}}$ and is thus also a counter-model for the validity of the pigeonhole formula in $\mathbb{V}_{\{0..<n\}}$. The counter-model for pigeonhole formula number n is

$$c_n(y) = \begin{cases} \mathit{Indet}\ i & \text{if } y = \mathsf{x}_i \text{ and } i < n \\ \bullet & \text{otherwise} \end{cases}$$

In order to prove that it indeed is a counter-model of the pigeonhole formula, a number of lemmas are introduced that characterize the semantics of the formula's components:

lemma *cla_false_Imp*:
assumes *"eval i a = •"*
assumes *"eval i b = ∘"*
shows *"eval i (a ⇒ b) = ∘"*

Proof. Follows directly from the involved definitions. □

lemma *eval_CON*:
"eval i ([⩕] ps) = Det (∀ p ∈ set ps. eval i p = •)"

Proof. Note that *set ps* denotes the set of members in the list *ps*. The lemma follows by induction on *ps* from the involved definitions. □

lemma *eval_DIS*:
"eval i ([⩖] ps) = Det (∃ p ∈ set ps. eval i p = •)"

Proof. Follows by induction on *ps* from the involved definitions. □

lemma *eval_ExiEql*:
"eval i ([∃ =] ps) = Det (∃ (p_1, p_2)∈(set ps $\times_{\textit{off-diag}}$ set ps). eval i p_1 = eval i p_2)"

Proof. Follows from the definition of [∃ =], the definition of $\times_{\text{off-diag}}$ and *eval_DIS*. □

is_indet t is defined to be true iff *t* is indeterminate. Likewise *is_det t* is true iff *t* is determinate.

lemma *eval_Nab*: *"eval i (∇ p) = Det (is_indet (eval i p))"*

Proof. Follows directly from the involved definitions. □

lemma *eval_NAB*:
"eval i ([∇] ps) = Det (∀ p ∈ set ps. is_indet (eval i p))"

Proof. Follows from the definition of [∇], *eval_CON* and *eval_Nab*. □

With this one can prove that the pigeonhole formula is false under the c_n counter-model.

lemma *interp_of_id_pigeonhole_fm_False*: *"eval c_n π_n = ∘"*

Proof. The lemma *cla_false_Imp* states that an implication can be proved false by proving its antecedent true and conclusion false. Start by proving the antecedent true: The antecedent is $[\nabla]_{\mathsf{x}_0,\ldots,\mathsf{x}_{n-1}}$, and this means that all the variables in $\mathsf{x}_0, \ldots, \mathsf{x}_{n-1}$ should refer to indeterminate values, which indeed they do by the definition of c_n. The conclusion $[\exists =]_{\mathsf{x}_0,\ldots,\mathsf{x}_{n-1}}$ is proved false using *eval_ExiEql*, which reduces the problem to proving that no pair of different symbols among $\mathsf{x}_0, \ldots, \mathsf{x}_{n-1}$ evaluate to the same. That follows from how c_n is defined. □

From this follows that the pigeonhole formula is not valid:

theorem *not_valid_pigeonhole_fm*: "$\neg$ *valid* π_n"

Proof. Follows from *interp_of_id_pigeonhole_fm_False*. □

It follows that the pigeonhole formula is not valid in $U_{\{0..<n\}}$:

theorem *not_valid_in_n_pigeonhole_fm*: "$\neg$ *valid_in* $\{0..<n\}$ π_n"

Proof. From c_n's definition follows that *range* $c_n \subseteq$ *domain* $\{0..<n\}$. It follows that π_n is not valid in $U_{\{0..<n\}}$ by *interp_of_id_pigeonhole_fm_False* and the definition of validity in $U_{\{0..<n\}}$ □

6.2 π_n is valid in $\mathbb{V}_{\{0..<m\}}$ for $m < n$

In order to prove that π_n is valid in $\mathbb{V}_{\{0..<m\}}$ for $m < n$, a new lemma on the semantics of an implication is needed:

lemma *cla_imp_I*:
  assumes "*is_det* (*eval i a*)"
  assumes "*is_det* (*eval i b*)"
  assumes "*eval i a* = • $\Longrightarrow$ *eval i b* = •"
  shows "*eval i* ($a \Rightarrow b$) = •"

Proof. Not surprisingly, it follows directly from the involved definitions. □

∇ and [∃ =] returning determinate values is also needed.

lemma *is_det_NAB*: "*is_det* (*eval i* ([**∇**] *ps*))"

Proof. The lemma follows from *eval_NAB*. □

lemma *is_det_ExiEql*: "*is_det* (*eval i* ([∃ =] *ps*))"

Proof. The lemma follows from *eval_ExiEql*. □

Moreover the pigeonhole principle is needed. This theorem is part of the Isabelle libraries in the following formulation:

lemma *pigeonhole*: "*card A* > *card* (*f* ‘ *A*) $\Longrightarrow$ $\neg$ *inj_on f A*"

It states that if the image of f on A is of smaller cardinality than A, then f cannot be an injection. From this follows a more specific formulation of the principle, which will be applied:

lemma *pigeon_hole_nat_set*:
  assumes "*f* ‘ $\{0..<n\} \subseteq \{0..<m\}$"
  assumes "$m < (n :: nat)$"
  shows "$\exists j_1 \in \{0..<n\}.\ \exists j_2 \in \{0..<n\}.\ j_1 \neq j_2 \wedge f\, j_1 = f\, j_2$"

Proof. From the assumptions follows that *card* $\{0..<n\}$ > *card* $\{0..<m\} \geq$ *card* (*f* ‘ $\{0..<n\}$). Therefore *pigeonhole* is applicable and the conclusion follows immediately. □

The pigeonhole formula will evaluate to true in any interpretation with truth values in $\mathbb{V}_{\{0..m\}}$ where $m < n - 1$:

lemma *eval_true_in_lt_n_pigeonhole_fm*:
  assumes "$m < n$"
  assumes "*range i* $\subseteq$ *domain* $\{0..<m\}$"
  shows "*eval i* $\pi_n = \bullet$"

Proof. Apply *cla_imp_I* to break down the conclusion. The two first assumptions of *cla_imp_I* follow from *is_det_NAB* and *is_det_ExiEql*, and then what remains is to prove that the antecedent of π_n implies the conclusion of π_n. Therefore, assume that the antecedent, $[\boldsymbol{\nabla}]_{\mathsf{x}_0,\ldots,\mathsf{x}_{n-1}}$, evaluates to true. From this and *eval_NAB* follows that $\mathsf{x}_0, \ldots, \mathsf{x}_{n-1}$ all evaluate to indeterminate values. This, together with the fact that the range of i is *domain* $\{0..<m\}$, means that i must map any x_l where $l \in \{0..<n\}$ to *Indet* k for some $k \in \{0..<m\}$. Therefore, by *pigeonhole_nat_set* there are $j_1 < n$ and $j_2 < n$ such that x_{j_1} and x_{j_2} are different but i evaluates them to the same value. This is by *eval_ExiEql* exactly what is required for the conclusion $[\exists =]_{\mathsf{x}_0,\ldots,\mathsf{x}_{n-1}}$ to evaluate to true. □

Therefore the pigeonhole formula must be valid in $\mathbb{V}_{\{0..<m\}}$.

theorem *valid_in_lt_n_pigeonhole_fm*:
  assumes "$m<n$"
  shows "*valid_in* $\{0..<m\}$ (*pigeonhole_fm n*)"

Proof. Follows immediately from *eval_true_in_lt_n_pigeonhole_fm*. □

There are many other finite sets than $\{0..<m\}$. It is therefore desirable to extend the theorem to claim that π_n is valid in any V_U where $|U| < n$. This can be done using the result from Section 5:

theorem *valid_in_pigeonhole_fm_n_gt_card*:
  assumes "*finite U*"
  assumes "*card U* $< n$"
  shows "*valid_in U* (*pigeonhole_fm n*)"

Proof. Follows from *valid_in_lt_n_pigeonhole_fm* and *bij_betw_valid_in* □

6.3 $\mathbb{V}$ is different from $\mathbb{V}_U$ where U is finite

The previous subsection demonstrated that π_n is valid in e.g. $\mathbb{V}_U$ where $|U| = n$ but not in $\mathbb{V}$. Therefore the logics are different:

theorem *extend*: "*valid* $\neq$ *valid_in U*" **if** "*finite U*"

Proof. Follows from *valid_in_pigeonhole_fm_n_gt_card* and *not_valid_pigeonhole_fm*. □

This can be seen as a justification of the infinitely many values in the logic – they cannot once and for all be replaced by a finite subset. The reduction in Section 4 only worked because there the size of U depended on the considered formula.

7 Discussion and Related Work

My previous paper with Villadsen [20] contains a thorough discussion of related work giving an overview of various many-valued logics that have been formalized in Isabelle/HOL. I will refrain from repeating the section here and mention again only the most pertinent works namely by Marcos [9] and Steen and Benzmüller [14]. Marcos developed an ML program that can generate proof tactics; these tactics implement tableaux that can prove theorems in various finitely many-valued logics. Steen and Benzmüller defined a shallow embedding of the many-valued SIXTEEN logic into classical HOL. This means that one can take a formula in SIXTEEN, translate it to classical HOL using their embedding and then try to prove it using a theorem prover for HOL. Benzmüller and Woltzenlogel Paleo [5] used the same approach to embed several higher-order modal logics and also showed the approach applied to a sketch of a paraconsistent logic. Several other logics have been embedded in HOL in this way, including conditional logics by Benzmüller, Gabbay, Genovese and Rispoli [2], quantified multimodal logics by Benzmüller and Paulson [3], first-order nominal logic by Steen and Wisniewski [21] and free logic by Benzmüller and Scott [4]. In contrast, the formalization in this chapter is a deep embedding of a logic and formalizes semantics rather than defining a tableau or a translation.

A noteworthy characteristic of the present formalization is that all proofs were built from the ground up in the proof assistant – it was not based on any preexisting proofs. Proof assistants make it very clear when a proof is finished, and one does not have to reread it over and over to see if everything adds up. Furthermore, in the development I tried out different definitions of the implication used in the pigeon-formula and the proof assistant was very helpful in checking that the changes did not break any proofs. Proof assistants of course ensure correctness of proofs. Many times I stated lemmas and proved them directly in the proof assistant. Other times the insurance of correctness was a hindrance in that on the way to a correct proof it was helpful to state lemmas that were "mostly correct" and whose expressions "mostly type checked", i.e. I abstracted away from some of the details. This was often better done on a piece of paper than in the proof assistant. However, after this process was done, it was definitely worth returning to the proof assistant to see if the "mostly correct" proof held up to the challenge of being formalized and thus turned into a correct proof.

The propositional fragment of a paraconsistent infinite-valued higher-order logic has now been formalized. An obvious next step would be to formalize the whole paraconsistent higher-order logic. The basis of such an endeavor could be the formalizations of HOL Light in HOL Light and HOL4 by respectively Harrison [6] and Kumar et al. [8]. The challenge is to give a semantics to the language. In the formalization in HOL4 this is done by abstractly specifying set theory in HOL. The same specification could be used for giving a semantics to the paraconsistent higher-order logic.

8 Conclusion

This chapter formalizes Villadsen's paraconsistent infinite-valued logic $\mathbb{V}$ and the $|U|$-valued logics $\mathbb{V}_U$ as well as proves and formalizes several meta-theorems of the logic. One meta-theorem shows that, for any formula, the question of its validity in $\mathbb{V}$ can be reduced to the question of its validity in $\mathbb{V}_U$ for a large enough finite U. The other meta-theorems, not previously presented, characterize how the number of truth-values affect truths of the logic. One of them shows that when $|U| = |W|$ then $\mathbb{V}_U$ has the same truths as $\mathbb{V}_W$. Another shows that for any finite U it is the case that $\mathbb{V}$ and $\mathbb{V}_U$ are different logics. The theory was developed in parallel with its formalization. This illustrates that proof assistants can be used as tools, not only for formalizing established results, but also for developing new results – in this case the meta-theory of a logic.

Acknowledgements

Jørgen Villadsen, Jasmin Christian Blanchette and John Bruntse Larsen provided valuable feedback on the chapter and the formalization. Thanks to Freek Wiedijk for discussions.

References

[1] S. Akama, editor. *Towards Paraconsistent Engineering*, volume 110 of *Intelligent Systems Reference Library*. Springer, 2016.

[2] C. Benzmüller, D. Gabbay, V. Genovese, and D. Rispoli. Embedding and automating conditional logics in classical higher-order logic. *Annals of Mathematics and Artificial Intelligence*, 66(1):257–271, 2012.

[3] C. Benzmüller and L. C. Paulson. Quantified multimodal logics in simple type theory. *Logica Universalis*, 7(1):7–20, 2013.

[4] C. Benzmüller and D. Scott. Automating free logic in Isabelle/HOL. In G.-M. Greuel, T. Koch, P. Paule, and A. Sommese, editors, *International Conference on Mathematical Software (ICMS)*, volume 9725 of *LNCS*, pages 43–50. Springer, 2016.

[5] C. Benzmüller and B. Woltzenlogel Paleo. Higher-order modal logics: Automation and applications. In W. Faber and A. Paschke, editors, *Reasoning Web (RW)*, volume 9203 of *LNCS*, pages 32–74. Springer, 2015.

[6] J. Harrison. Towards self-verification of HOL Light. In U. Furbach and N. Shankar, editors, *International Joint Conference on Automated Reasoning (IJCAR)*, volume 4130 of *LNCS*, pages 177–191. Springer, 2006.

[7] A. S. Jensen and J. Villadsen. Paraconsistent computational logic. In P. Blackburn, K. F. Jørgensen, N. Jones, and E. Palmgren, editors, *8th Scandinavian Logic Symposium: Abstracts*, pages 59–61. Roskilde University, 2012.

[8] R. Kumar, R. Arthan, M. O. Myreen, and S. Owens. Self-formalisation of higher-order logic: Semantics, soundness, and a verified implementation. *Journal of Automated Reasoning*, 56(3):221–259, 2016.

[9] J. Marcos. Automatic generation of proof tactics for finite-valued logics. In I. Mackie and A. Martins Moreira, editors, *Tenth International Workshop on Rule-Based Programming, Proceedings*, volume 21 of *Electronic Proceedings in Theoretical Computer Science*, pages 91–98. Open Publishing Association, 2010.

[10] G. Priest, K. Tanaka, and Z. Weber. Paraconsistent logic. In E. N. Zalta, editor, *Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, summer 2018 edition, 2018. `https://plato.stanford.edu/archives/sum2018/entries/logic-paraconsistent/`.

[11] A. Schlichtkrull. Formalization of a paraconsistent infinite-valued logic (short abstract). *Automated Reasoning in Quantified Non-Classical Logics – 3rd International Workshop – Program*, 2018. `https://easychair.org/smart-program/FLoC2018/ARQNL-program.html`.

[12] A. Schlichtkrull and J. Villadsen. IsaFoL: Paraconsistency. `https://bitbucket.org/isafol/isafol/src/master/Paraconsistency/`.

[13] A. Schlichtkrull and J. Villadsen. Paraconsistency. *Archive of Formal Proofs*, Dec. 2016. `http://isa-afp.org/entries/Paraconsistency.html`, Formal proof development.

[14] A. Steen and C. Benzmüller. Sweet SIXTEEN: Automation via embedding into classical higher-order logic. *Logic and Logical Philosophy*, 25(4):535–554, 2016.

[15] J. Villadsen. Combinators for paraconsistent attitudes. In P. de Groote, G. Morrill, and C. Retoré, editors, *Logical Aspects of Computational Linguistics (LACL)*, volume 2099 of *LNCS*, pages 261–278. Springer, 2001.

[16] J. Villadsen. Paraconsistent assertions. In G. Lindemann, J. Denzinger, I. J. Timm, and R. Unland, editors, *Multi-Agent System Technologies (MATES)*, volume 3187 of *LNCS*, pages 99–113, 2004.

[17] J. Villadsen. A paraconsistent higher order logic. In B. Buchberger and J. A. Campbell, editors, *Artificial Intelligence and Symbolic Computation (AISC)*, volume 3249 of *LNCS*, pages 38–51. Springer, 2004.

[18] J. Villadsen. Supra-logic: Using transfinite type theory with type variables for paraconsistency. *Logical Approaches to Paraconsistency, Journal of Applied Non-Classical Logics*, 15(1):45–58, 2005.

[19] J. Villadsen and A. Schlichtkrull. Formalization of Many-Valued Logics. In H. Christiansen, M. Jiménez-López, R. Loukanova, and L. Moss, editors, *Partiality and Underspecification in Information, Languages, and Knowledge*, chapter 7. Cambridge Scholars Publishing, 2017.

[20] J. Villadsen and A. Schlichtkrull. Formalizing a paraconsistent logic in the Isabelle proof assistant. In A. Hameurlain, J. Küng, R. Wagner, and H. Decker, editors, *Transactions on Large-Scale Data- and Knowledge-Centered Systems (TLDKS)*, volume 10620 of *LNCS*, pages 92–122. Springer, 2017.

[21] M. Wisniewski and A. Steen. Embedding of quantified higher-order nominal modal logic into classical higher-order logic. In C. Benzmüller and J. Otten, editors, *ARQNL 2014. Automated Reasoning in Quantified Non-Classical Logics*, volume 33 of *EPiC Series in Computing*, pages 59–64. EasyChair, 2015.

Thesis Appendix: Changes to Published Papers

Chapter 1:
Formalization of the Resolution Calculus for First-Order Logic

The chapter was published as a paper by me in a special issue of “Journal of Automated Reasoning” on “Milestones in Interactive Theorem Proving” [1]. The references have been put in alphabetic order.

Chapter 2:
Formalizing Bachmair and Ganzinger’s Ordered Resolution Prover

The chapter’s content is the technical report extending the paper “Formalizing Bachmair and Ganzinger’s Ordered Resolution Prover” by Blanchette, Traytel, Waldmann and myself that was published in the Proceedings of the 9th International Joint Conference on Automated Reasoning (IJCAR 2018) [2]. The extension consists of more thorough explanations and an appendix listing, for reference, the errors and imprecisions our formalization revealed in the chapter on “Resolution Theorem Proving” by Bachmair and Ganzinger.

Chapter 3:
A Verified Automatic Prover Based on Ordered Resolution

The chapter is a draft paper written by Blanchette, Traytel and myself [3]. The paper is submitted, but not published as of the submission of this thesis.

Chapter 4:
NaDeA: A Natural Deduction Assistant with a Formalization in Isabelle

The chapter was published as a paper by Jensen, Villadsen and myself in a special issue of the “IfCoLog Journal of Logics and their Applications” on “Tools for Teaching Logic” [4].

The following changes were made:

- The expression “ $\exists p.\ OK\ p\ []$ ***for*** p” is corrected to “ $\exists p.\ OK\ p\ []$” in section 5.6.
- The name “Andreas Halkjær” is corrected to “Andreas Halkjær From” in the references.

Chapter 5:
Programming and Verifying a Declarative First-Order Prover in Isabelle/HOL

The chapter was published as a paper by Jensen, Larsen, Villadsen and myself in a special issue of “AI Communications” on “Automated Reasoning” [5]. No changes have been made.

Chapter 6:
Formalized Meta-Theory of a Paraconsistent Logic

The chapter is a draft paper written by me [6]. The paper is submitted, but not published as of the submission of this thesis.

Changes Made in November 2018

Introduction

On page 8 the occurrence of the word “uncountably” is corrected to “countably”.

Chapter 1:
Formalization of the Resolution Calculus for First-Order Logic

On page 38 the occurrence of the word “uncountably” is corrected to “countably”.

Chapter 3:
A Verified Automatic Prover Based on Ordered Resolution

On page 87, an occurrence of the word “not” is corrected to “nor”.

On page 92, in the definition of wit, the expression $\mathsf{LCons}\ x\ (\mathsf{pick}\ R\ x\ y \mathbin{+\!\!+} \mathsf{wit}\ R\ xs)$ is corrected to $\mathsf{LCons}\ x\ (\mathsf{pick}\ R\ x\ y \mathbin{+\!\!+} \mathsf{wit}\ R\ (\mathsf{LCons}\ y\ ys))$.

References

[1] A. Schlichtkrull. Formalization of the resolution calculus for first-order logic. *Journal of Automated Reasoning*, 61(4):455–484, 2018.

[2] A. Schlichtkrull, J. C. Blanchette, D. Traytel, and U. Waldmann. Formalizing Bachmair and Ganzinger’s ordered resolution prover. In D. Galmiche, S. Schulz, and R. Sebastiani, editors, *International Joint Conference on Automated Reasoning (IJCAR)*, pages 89–107. Springer, 2018. Extended in technical report: `http://matryoshka.gforge.inria.fr/pubs/rp_report.pdf`.

[3] A. Schlichtkrull, J. C. Blanchette, and D. Traytel. A verified automatic prover based on ordered resolution. 2018. Submitted.

[4] J. Villadsen, A. B. Jensen, and A. Schlichtkrull. NaDeA: A natural deduction assistant with a formalization in Isabelle. *IfCoLog Journal of Logics and their Applications*, 4(1):55–82, 2017.

[5] A. B. Jensen, J. B. Larsen, A. Schlichtkrull, and J. Villadsen. Programming and verifying a declarative first-order prover in Isabelle/HOL. *AI Communications*, 31(3):281–299, 2018.

[6] A. Schlichtkrull. Formalized meta-theory of a paraconsistent logic. 2018. Submitted.