

User-Friendly Formal Methods

Towards user-friendly proof mechanization

PhD Thesis

Frederik Krogsdal Jacobsen



User-Friendly Formal Methods

Towards user-friendly proof mechanization

PhD Thesis

June, 2024

By

Frederik Krogsdal Jacobsen

Copyright: Reproduction of this publication in whole or in part must include the customary bibliographic citation, including author attribution, report title, etc.

Cover photo: Ulrike Leone, 2017. Detail of the anastylosed Temple E at Selinus, Sicily.

Published by: Technical University of Denmark,
Department of Applied Mathematics and Computer Science,
Richard Petersens Plads, Building 324, 2800 Kgs. Lyngby, Denmark
www.compute.dtu.dk

ISSN: 0909-3192

Abstract

Formal methods for software engineering make it possible to verify the correctness of software using logical and mathematical techniques. Unfortunately, formal methods have a reputation for being difficult to understand and use. This thesis investigates approaches to improving the ease of use and usefulness of formal methods.

The thesis has a focus on proof assistants and mechanization of proofs about concurrent calculi, and investigates three research questions:

- What are effective approaches to learning proof competence with computer assistance?
- How can we improve the helpfulness of proof assistants and related tools?
- What is the state of the art in efficiently mechanizing proofs about concurrent calculi, and how can we improve it?

Towards answering these questions, this thesis contains:

- A sequent calculus verifier for learning proof competence, leveraging the proof assistant Isabelle to check proofs.
- An overview of the computer science curriculum at the Technical University of Denmark, contextualizing the situations in which students learn proof competence.
- A study providing preliminary evidence for the efficacy of learning logic and proof competence via functional programming in a proof assistant.
- A tutorial-style investigation of approaches to testing learning outcomes with proof assistants in the context of written exams.
- An instrumented proof assistant for conducting studies of the efficacy of using proof assistants to facilitate learning of proof competence.
- An automated sequent calculus theorem prover for first-order logic with functions, with a formally verified soundness and completeness proof that considers the search strategy of the prover.
- A set of benchmark challenges addressing three key issues for formal reasoning about concurrent calculi.

The thesis thus provides a broad foundational contribution towards rebuilding the reputation of formal methods in software engineering.

Resumé

Formelle metoder til softwareudvikling gør det muligt at verificere at software fungerer korrekt ved hjælp af logiske og matematiske teknikker. Desværre har formelle metoder ry for at være svære at forstå og bruge. Denne afhandling undersøger tilgange til at gøre formelle metoder mere brugbare og lettere at bruge.

Afhandlingen fokuserer på bevisassistenter og mekanisering af beviser om parallelle kalkuler og undersøger tre forskningsspørgsmål:

- Hvad er effektive tilgange til at lære beviskompetencer med computerassistance?
- Hvordan kan vi gøre bevisassistenter og relaterede værktøjer mere hjælpsomme?
- Hvad er state-of-the-art inden for effektiv mekanisering af beviser om parallelle kalkuler, og hvordan kan den forbedres?

Som et bidrag til besvarelsen af disse spørgsmål indeholder afhandlingen:

- En sekventkalkuleefterprøver, som kan bruges til at lære beviskompetencer, og som benytter bevisassistenten Isabelle til efterprøvning af beviser.
- Et overblik over det datalogiske pensum på Danmark Tekniske Universitet som kontekstualiserer de situationer hvori studerende lærer beviskompetencer.
- En forundersøgelse, der giver et foreløbigt indblik i effekten af at bruge funktionsprogrammering med en bevisassistent til at lære logik og beviskompetencer.
- En undersøgelse af metoder til måling af om læringsmål er opfyldt, i form af en guide til at bruge bevisassistenter i skriftlige eksamener.
- En instrumenteret bevisassistent som kan bruges til at undersøge effekten af at bruge bevisassistenter til at facilitere læring af beviskompetencer.
- En automatisk sekventkalkulebevisfører for førsteordenslogik med funktioner, med et formelt verificeret bevis for sundhed og kompletthed, der tager højde for bevisførerens søgestrategi.
- En samling udfordrende opgaver, der kan fungere som sammenligningsgrundlag for tre kerneproblemer inden for formel bevisførelse om parallelle kalkuler.

Afhandlingen repræsenterer dermed et bredt fundamentalt bidrag til genopbygningen af formelle metoders ry i softwareudvikling.

Preface

I wrote this thesis at the Section for Algorithms, Logic and Graphs at the Department of Applied Mathematics and Computer Science at the Technical University of Denmark (DTU) in partial fulfilment of the requirements for the PhD degree. My three years of PhD studies started on July 1, 2021 and ended on June 30, 2024. My studies were funded by a PhD fellowship from the Department of Applied Mathematics and Computer Science at DTU. My main supervisor was Jørgen Villadsen and I was co-supervised by Alceste Scalas.

Throughout the text, I assume that the reader has basic knowledge of computer science and logic. Please note that the existing publications included in this thesis are reproduced as published, including their original page numbers and formatting. Look in the lower outside corners of each page to find the page numbers relevant to this thesis.

I'm deeply indebted to my advisors, Jørgen Villadsen and Alceste Scalas, for their invaluable guidance and trust in me during my studies. I must also thank all of my collaborators for their hard work and for always contributing to a pleasant work environment; in alphabetical order: Marco Carbone, David Castro-Perez, Kim Jana Eiken, Francisco Ferreira, Asta Halkjær From, Lorenzo Gheri, Nadine Karsten, Simon Tobias Lund, Christoph Math-eja, Alberto Momigliano, Uwe Nestmann, Luca Padovani, Alceste Scalas, Dawit Tirore, Martin Vassor, Jørgen Villadsen, Nobuko Yoshida, and Daniel Zackon.

I am grateful to everyone at the Section for Algorithms, Logic and Graphs for their enjoyable company and all of the fun we had along the way. I would also like to thank everyone in the Mobility Research Group at the University of Oxford and at the Department of Computer Science at Royal Holloway, University of London for welcoming me during my visits. My deepest gratitude to Nobuko Yoshida and Francisco Ferreira for hosting me during my research visits.

I would like to thank the Department of Applied Mathematics and Computer Science at DTU for funding my studies, and Otto Mønsted Fonden for supporting my external research visit and my participation in the Federated Logic Conference and the Federated Conference on Distributed Computing Techniques.

This work would not have been possible without the unwavering support of my friends and family. Above all, I would like to thank Anna for her love and constant support. Thank you for the late nights and the early mornings. Thank you for keeping me sane and for your patience of my rambles. Most of all, thank you for being my best friend.

Frederik Krogsdal Jacobsen
Copenhagen, June 2024

Contents

Abstract	iii
Resumé	v
Preface	vii
1 Introduction	1
2 SeCaV: A Sequent Calculus Verifier in Isabelle/HOL	9
3 Using Isabelle in Two Courses on Logic and Automated Reasoning	27
4 Teaching Functional Programmers Logic and Metatheory	43
5 On Exams with the Isabelle Proof Assistant	63
6 ProofBuddy: A Proof Assistant for Learning and Monitoring	77
7 Verifying a Sequent Calculus Prover for First-Order Logic with Functions in Isabelle/HOL	99
8 The Concurrent Calculi Formalisation Benchmark	129
List of Publications Reproduced in this Thesis	139

1 Introduction

The title of my PhD project is User-Friendly Formal Methods. While defining the term “formal methods” is not without complications (I will attempt to do so shortly), “user-friendly” is an especially nebulous term. In short, user-friendliness refers to the aspects of a system or technique that make it easy to learn, understand, deal with, use, obtain, access, operate, and so on. The vagueness and subjectiveness of these aspects means that it is impossible to have the final word on whether any given system is user-friendly or not. We can, however, attempt to improve the situation by making systems *more* user-friendly.

In this thesis, the systems in question are tools and techniques for implementing formal methods in software engineering. Much software today is complex and difficult to understand even for the engineers implementing it. It is not surprising, then, that thinking or talking informally about software does not often lead to correct implementations. Formal methods are techniques for specifying, developing, analysing and verifying properties of computer systems in a mathematically rigorous way. In this context, computer systems can consist of hardware, software, or both, but in this thesis I restrict the scope to software. Formal methods is thus an umbrella term for any technique or tool that can provide mathematical evidence towards the correctness of software. Such evidence is possible to obtain both by manual techniques and by fully automated tools.

This thesis is primarily concerned with a class of formal methods called proof assistants. Proof assistants are interactive programs which help users write mathematically rigorous proofs. They do this by verifying that the steps of reasoning employed in a proof are actually correct. Most proof assistants also provide facilities for automatically writing the easy parts of proofs and for automatically searching for proofs of the complex parts. We thus often say that proofs carried out using a proof assistant are “mechanized” proofs. I work primarily in the proof assistant Isabelle, which allows me to write proofs in the expressive language of higher-order logic. This language looks similar to ordinary mathematics, but is fully precise and rigorous.

It is not easy to determine the exact line between formal methods and “normal” software engineering. For instance, property-based testing [1] is sometimes considered a formal method [2]. While property-based testing *can* provide mathematical guarantees about correctness, the testing is rarely exhaustive in practice. Model checking is essentially the same technique, but with the guarantee that the testing is exhaustive. Model checking is thus certainly a formal method, since the technique provides a mathematical guarantee of correctness.

Some formal methods are now so common that most software engineers take them for granted as part of their day-to-day work. These include program analysis tools and type checkers, which are now integrated in almost all software development environments. As an example, the mainstream programming language Rust has a so-called borrow checker directly integrated in the standard compiler [3]. Most software engineers would probably still refer to constraint-based non-lexical lifetime analysis as a formal method, despite the fact that it is simply another name for the same concept [4]. It seems that formal methods are only called so until they enter the mainstream.

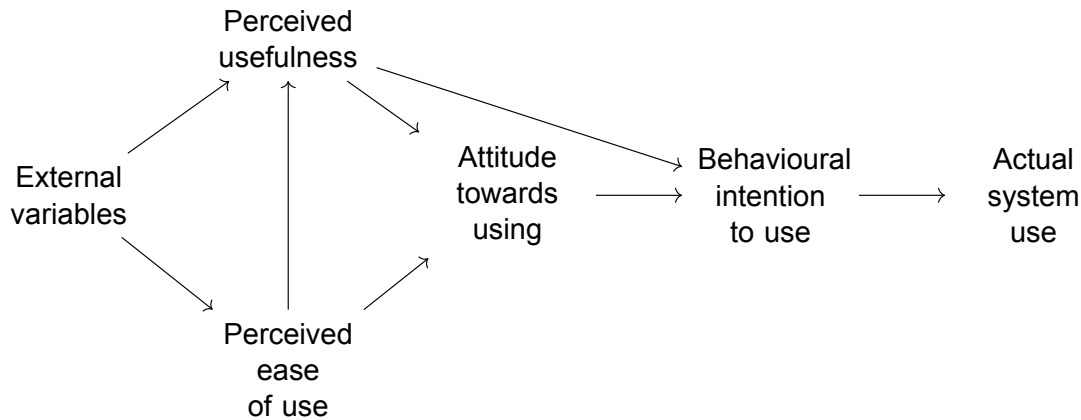


Figure 1.1: The Technology Acceptance Model.

1.1 Why are formal methods not mainstream?

Despite decades of research, formal methods are still mainly used in academia and in highly regulated or safety-critical industries. One approach to understanding why formal methods have not entered the mainstream is to investigate how software engineers come to accept and use new technologies such as formal methods. One of the most influential models in this space is the Technology Acceptance Model [5], pictured in fig. 1.1.

While there have been extensions to the Technology Acceptance Model to account for factors affecting specific industries and situations, its generality and simplicity make it a good fit for this general overview. The main factors in the model are the perceived usefulness and the perceived ease of use of a technology. The term “user-friendly” is generally used to mean some combination of properties relating to these two factors. I will adopt this convention: a system is user-friendly if it is both useful and easy to use.

Perceived usefulness is the degree to which a person believes that using a particular technology would enhance their job performance. Most software today is complex and there is scarcely any without implementation bugs. Most software engineers would thus be able to enhance their job performance (i.e. write software with fewer bugs) if they were able to use formal methods without expending any effort. For formal methods, the main problem regarding usefulness is expressivity: before a software engineer can believe that a formal method is useful, they must first believe that it is possible to express the properties they wish to verify about their software using that method. Here there is a distinction between what is theoretically possible and what is possible in practice. For instance, it is theoretically possible to use the formal system introduced by Bourbaki to reason about mathematics. In practice, however, Mathias has shown that encoding the number 1 requires 4,523,659,424,929 symbols [6]. Similarly, formal tools which are so slow that software engineers cannot use them to verify realistic systems are not useful within this terminology. There is also a distinction between what is possible in practice and what users believe to be possible. For example, software engineers may falsely believe that formal methods are not able express properties that are in fact easy to express. The main research problem for usefulness is thus to find efficient ways to encode properties and to communicate that these exist to software engineers.

Perceived ease of use is the degree to which a person believes that using a particular technology would be free from effort. Compared to usefulness, this is a larger problem for formal methods, since formal methods are generally not regarded to be easy to use.

Any formal method that requires users of the technique or tool to manually write proofs requires that users have the proof competence to do so. This is especially the case for proof assistants, where writing proofs is the main activity involved in using the tool. Fully automatic formal tools do not require proof competence, but do require software engineers using them to have a strong understanding of the performance characteristics of the properties they wish to verify. Such an understanding has unfortunately turned out to be difficult to develop, and even researchers in this field often base their results on experimental data rather than a theoretical understanding of performance characteristics [7, 8]. As a result, users of fully automatic tools must know “the tricks of the trade” that improve performance in non-intuitive ways. In some cases, this leads users to essentially reinvent proof assistants in their automatic tools [9]. An important aspect that can make systems easier to use is helpfulness. A helpful system has properties such as comprehensive documentation for both beginners and experts, warnings about potential pitfalls, and sub-systems to aid learners. Helpful systems can alleviate both the problem of lacking proof competence and the problem of having to know performance “hacks” by allowing users to learn faster. Unfortunately, most formal methods systems are not very helpful.

There are of course also external variables which can impact the perceived usefulness and ease of use of a technology. For instance, trends in the industry can socially influence engineers to try formal methods, and structural imperatives such as regulations and standards can force adoption. Conversely, factors such as cost and a lack of perceived self-efficacy with formal methods can dissuade adoption.

In this thesis, I will primarily focus on investigating and improving the ease of use of formal methods, returning to the topic of usefulness in the final chapters.

1.2 Research questions

In the previous sections, we have seen that formal methods are not yet user-friendly. This thesis will particularly focus on proof assistants, which, while useful for verifying properties of many types of software, are not easy to use. The objective of this thesis is thus to investigate how we can make it easier to leverage the benefits of proof assistants.

As discussed in the previous section, there are three primary problems for the user-friendliness of proof assistants:

1. Lack of proof competence in software engineers
2. Lack of helpfulness in tools
3. Lack of efficient encodings for properties of certain types of software

These problems are not equally large, with the last two being most severe. All of the problems, however, are large enough that there is no hope of solving them in general in a single project. For that reason, I will focus on proof assistants and mechanization of proofs about concurrent calculi in particular. The rest of this thesis will thus investigate the following more specialized research questions, as described in more detail in the next section:

1. What are effective approaches to learning proof competence with computer assistance?
2. How can we improve the helpfulness of proof assistants and related tools?
3. What is the state of the art in efficiently mechanizing proofs about concurrent calculi, and how can we improve it?

1.3 Synopsis of the Following Chapters

In this section, I will give a brief overview of the main points of each of the following chapters. My aim is to elucidate their interrelationships and contextualize them in relation to the objectives of the overall project. All of these chapters are reproductions of existing peer-reviewed articles. Page 139 contains a list of the reproduced publications and references to the associated versions of record.

1.3.1 SeCaV: A Sequent Calculus Verifier in Isabelle/HOL

Chapter 2 describes SeCaV: a sequent calculus verifier leveraging the proof assistant Isabelle/HOL to check proofs. The verifier is an inductively defined sequent calculus proof system with a few auxiliary functions implemented in Isabelle/HOL. The main use of SeCaV is as a tool for learning how to write sequent calculus proofs in a controlled and simple environment with instant feedback. Learners can use the verifier directly inside Isabelle/HOL or with the SeCaV Unshortener web application, which has a more lightweight syntax. In either case, the verifier allows learners to follow a concrete and controlled structure when writing proofs, thus clarifying the options available at any given proof step. The soundness and completeness of the verifier is also proven in Isabelle/HOL, and learners can use the simple mechanization as a way to gain basic insight on how to use Isabelle/HOL.

SeCaV will appear as a basic tool for learning proof competence in most of the following chapters, and as part of an approach to improving the helpfulness of automated theorem provers in chapter 7.

1.3.2 Using Isabelle in Two Courses on Logic and Automated Reasoning

Chapter 3 is an overview of the teaching context at the Technical University of Denmark (DTU), as referred to in some of the other chapters. It describes our Bachelor level course on logical systems and logic programming and our Master level course on automated reasoning. The main purpose of the chapter is to contextualize the other chapters on proof competence learning by describing our courses and their place in the overall curriculum.

The chapter also includes an overview of the tools we use to facilitate proof competence learning for beginners and the progression towards learning to use Isabelle/HOL. These tools include the point-and-click natural deduction tool NaDeA and the sequent calculus tool SeCaV, which act as starting points. Within these tools, it is difficult to accidentally write proofs that make no sense, since the controlled environments constrain the options of the user. In the Master level course, we then use a series of simple proof systems implemented directly in Isabelle/Pure to move from the controlled environment to the full power of Isabelle/HOL. Unlike the “real” Isabelle/HOL, these proof systems are entirely implemented in a single file, and it is thus still possible for learners to get an overview of their options at any given point. The systems also have no automation, which forces students to learn how to use basic proof rules before progressing to systems with automation.

1.3.3 Teaching Functional Programmers Logic and Metatheory

Chapter 4 is a study of the efficacy of learning logic and proof competence via functional programming. The overall hypothesis is that learners who already know some functional programming will benefit from seeing and experimenting with functional implementations of logical concepts when learning how to write formal logical proofs. We implemented the logical concepts, and mechanized standard properties about them, in Isabelle/HOL. This allows learners to interactively check examples, run auxiliary functions to check their understanding, and modify definitions while the proof assistant immediately tells them if and where these modifications break the proofs. This chapter is thus a preliminary study

of the efficacy of using proof assistants as a teaching tool for proof competence. To my knowledge, this is the only such study considering graduate students.

This chapter provides evidence that concrete implementations aid understanding of concepts in logic and confirms that students do experiment with these implementations to gain understanding. It also shows that prior experience with functional programming is useful for self-perceived confidence with our course and that students also gain confidence in functional programming. Due to technical and methodological limitations, the study remains preliminary. I will return to efforts towards improving the situation in chapter 6.

1.3.4 On Exams with the Isabelle Proof Assistant

Automation of simple proof cases and automatic proof search procedures are generally a major benefit of working within a proof assistant. When learning proof competence, however, this automation can make it difficult to test student learning outcomes since students can “cheat” using the automation. The main issue is designing test problems that are sufficiently easy for students to solve while avoiding problems that are trivial to solve with automation. In a written exam situation, the issue is especially severe, since there is only limited time, meaning that the test problems must be small.

Chapter 5 is a tutorial-style investigation of approaches we have used to prevent these issues. One solution is to simply not use proof assistants to test learning outcomes, but this means losing the scalability advantage of almost automatic grading of proofs, and additionally that proficiency with the proof assistant itself is not tested. We instead describe how to use the proof assistant in ways that prevent students from using automation, and discuss our experiences with the approach.

1.3.5 ProofBuddy: A Proof Assistant for Learning and Monitoring

We based the data in chapter 4 on perceived aptitude and attitudes towards the approach of learning proof competence with a proof assistant. This means that we must trust these rating-based perceptions, which are known to be unreliable [10], and that we could not investigate the details of student interaction with the tools.

Chapter 6 describes ProofBuddy, which is a tool for conducting studies of the efficacy of using proof assistants to facilitate learning of proof competence. ProofBuddy is essentially an instrumented frontend for the Isabelle proof assistant, giving it the ability to collect data about how students interact with the proof assistant. The frontend is configurable to allow instructors to e.g. restrict the syntax and automation that students can use. ProofBuddy also provides facilities to give adaptive and individual feedback to students as they solve exercises.

ProofBuddy is now in use at Technische Universität Berlin (TUB), where it underpins the studies involved in the curriculum reform project ProveIT [11]. This project aims to establish proof competence as an explicit subject in the BSc Computer Science programme at TUB. Concretely, the project investigates how to utilize learning software such as ProofBuddy to improve learning outcomes, while also improving the feedback and usability of the ProofBuddy tool itself.

1.3.6 Verifying a Sequent Calculus Prover for First-Order Logic with Functions in Isabelle/HOL

Chapter 7 describes the design, implementation and verification of an automated theorem prover for first-order logic with functions. We based the proof strategy of the prover on the SeCaV proof system, and the prover can generate proof certificates in the SeCaV Unshortener format. This approach allows the prover to be more helpful than most

automated theorem provers in the sense that it can generate proofs that are both human-readable and machine-verifiable. We implemented the prover and verified its soundness and completeness in Isabelle/HOL.

We prove completeness of the prover analytically, i.e. by building a countermodel from any failed proof attempt. Since we are proving completeness of a concrete proof strategy, we must prove that the strategy is sufficient to prove all valid formulas. Additionally, we must build our countermodel only over those terms that are actually used in the proof attempt. We solve this by introducing the notion of a bounded semantics where the domain is explicit. By proving that this notion is sound with respect to the original SeCaV semantics, we show that the prover will find a proof for any formula derivable in SeCaV.

1.3.7 The Concurrent Calculi Formalisation Benchmark

Chapter 8 considers both the usefulness and the helpfulness of proof assistants from the perspective of concurrent calculi. Concurrent calculi have three key features that make it difficult to formally prove properties about them: linear handling of communication channels, scope extrusion of names, and coinductive reasoning about infinite behaviours.

By creating a set of benchmark challenges for formally reasoning about concurrent calculi, we hope to:

- elucidate the current state of the art
- set the foundation for developing and sharing best practices and tutorials relating to these techniques
- discover potential improvements to proof assistants that would make them more useful and/or helpful when reasoning about concurrent calculi

Several research groups are working on solutions to the challenges in this chapter, and their mechanizations and experience reports are continuously added to the benchmark website as they are published.

1.4 Summary of Work Not Included in this Thesis

Besides working on the published articles reproduced in chapters 2 to 8, I have also contributed to projects that have not yet resulted in a published peer-reviewed article, or which have been incorporated in articles in different forms. These projects are briefly described here.

1.4.1 Lessons of Teaching Formal Methods with Isabelle

This paper was accepted at the 2022 Isabelle Workshop, but only informally published. It describes more recent experiences in the vein of chapter 3, focusing on lessons learned that are relevant to the Isabelle community.

1.4.2 Teaching Logic for Computer Science Students: Proof Assistants and Related Tools

This short position paper was accepted at the Why and how to teach Logic for CS undergraduates? workshop in 2022, but never published. It advocates a focus on logic as a tool when teaching computer science to undergraduates. Additionally it discusses the potential benefits of using proof assistants in logic education. The paper is in this sense a precursor to chapters 4 and 6.

1.4.3 Ongoing Work

With Lorenzo Gheri, Alceste Scalas and Nobuko Yoshida, I am developing a theory for compositional verification of multiparty session types. Existing compositional verification

results are based only on global types, but we are developing results for a generalised theory which supports, but does not require, global types. Additionally, our approach is compatible with existing verification approaches for liveness and other model-checkable properties.

With Jørgen Villadsen, I am considering a concise formalization of a sequent calculus and an automatic theorem prover for classical implicational logic with a focus on the fine points of the mechanized proofs of correctness. The formalization is available from the Archive of Formal Proofs [12].

With Francisco Ferreira, I have worked on an embedding of second-order abstract syntax into higher-order logic, namely Isabelle/HOL. Second-order abstract syntax can encode any reasonable syntax with binders, but cannot encode arbitrary judgments. By embedding a second-order syntactic framework into higher-order logic, we obtain the ability to define judgments in the meta-logic. This is also possible with specialized proof assistants for higher-order abstract syntax such as Beluga, but our approach has different trade-offs. Restricting our framework to second-order abstract syntax allows us to take advantage of the rich library ecosystem and automation capabilities of Isabelle/HOL when reasoning about judgments, at the cost of having to prove substitution lemmas for each judgment instead of getting them “for free”.

References

- [1] Koen Claessen and John Hughes. “QuickCheck: a lightweight tool for random testing of Haskell programs”. In: *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*. ICFP '00. New York, NY, USA: Association for Computing Machinery, 2000, pp. 268–279. DOI: 10.1145/351240.351266.
- [2] Harrison Goldstein et al. “Property-Based Testing in Practice”. In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. ICSE '24. Lisbon, Portugal: Association for Computing Machinery, 2024, 187. DOI: 10.1145/3597503.3639581.
- [3] David J. Pearce. “A Lightweight Formalism for Reference Lifetimes and Borrowing in Rust”. In: *ACM Trans. Program. Lang. Syst.* 43.1, 3 (Apr. 2021). ISSN: 0164-0925. DOI: 10.1145/3443420.
- [4] Niko Matsakis. *Non-lexical lifetimes*. Rust RFC 2094. 2017. URL: <https://rust-lang.github.io/rfcs/2094-nll.html>.
- [5] Younghwa Lee, Kenneth A. Kozar, and Kai R. T. Larsen. “The Technology Acceptance Model: Past, Present, and Future”. In: *Communications of the Association for Information Systems* 12, 50 (2003), pp. 752–780. DOI: 10.17705/1CAIS.01250.
- [6] A. R. D. Mathias. “A Term of Length 4 523 659 424 929”. In: *Synthese* 133 (2002), pp. 75–86. DOI: 10.1023/A:1020827725055.
- [7] Yue Li et al. “Scalability-first pointer analysis with self-tuning context-sensitivity”. In: *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ES-EC/FSE 2018. Lake Buena Vista, FL, USA: Association for Computing Machinery, 2018, pp. 129–140. DOI: 10.1145/3236024.3236041.
- [8] Minseok Jeon et al. “A Machine-Learning Algorithm with Disjunctive Model for Data-Driven Program Analysis”. In: *ACM Trans. Program. Lang. Syst.* 41.2, 13 (June 2019). ISSN: 0164-0925. DOI: 10.1145/3293607.
- [9] K. Rustan M. Leino. *Program Proofs*. MIT Press, 2023. ISBN: 9780262546232.

- [10] Georgios N. Yannakakis and Héctor P. Martínez. “Ratings are overrated!” In: *Frontiers in ICT* 2, 13 (2015). DOI: 10.3389/fict.2015.00013.
- [11] Nadine Karsten. *Proof Assistant for Teaching*. Research Profile. 2022. URL: <https://www.tu.berlin/en/mtv/research/research-profile/proof-assistants-for-teaching>.
- [12] Asta Halkjær From and Jørgen Villadsen. *Soundness and Completeness of Implicational Logic*. Formal proof development. 2022. URL: https://www.isa-afp.org/sessions/implicational_logic/#Implicational_Logic_Sequent_Calculus.

SeCaV: A Sequent Calculus Verifier in Isabelle/HOL

Asta Halkjær From Frederik Krogsdal Jacobsen Jørgen Villadsen

DTU Compute - Department of Applied Mathematics and Computer Science - Technical University of Denmark

We describe SeCaV, a sequent calculus verifier for first-order logic in Isabelle/HOL, and the SeCaV Unshortener, an online tool that expands succinct derivations into the full SeCaV syntax. We leverage the power of Isabelle/HOL as a proof checker for our SeCaV derivations. The interactive features of Isabelle/HOL make our system transparent. For instance, the user can simply click on a side condition to see its exact definition. Our formalized soundness and completeness proofs pertain exactly to the calculus as exposed to the user and not just to some model of our tool. Users can also write their derivations in the SeCaV Unshortener, which provides a lighter syntax, and expand them for later verification. We have used both tools in our teaching.

1 Introduction

Classical first-order logic plays an important role in mathematical logic and often occupies a central part in textbooks and courses on the subject. The sequent calculus is used to exemplify formal deduction and to show theoretical results in proof theory. It is instructive to write out concrete derivations in the calculus to get a feel for the rules and the method of reasoning. While such derivations can be done with pen and paper and checked for mistakes by human eyes, we argue that there is benefit in computer assistance.

To this end, we introduce SeCaV, a sequent calculus verifier built on top of Isabelle/HOL. SeCaV presents everything within the same unified system: the syntax of formulas, the proof rules, their side conditions, and the way derivations are written. Moreover, it provides immediate feedback to the user on the correctness of their derivations. This empowers the users when learning to write derivations and gives them an independence that is harder to achieve without computer assistance. We recall Nipkow [13] on the analogy between proof assistants and video games and especially the benefits of immediate feedback:

This is in contrast to the usual system of homework that is graded by a teaching assistant and returned a week later, long after the student struggled with it, and at a time when the course has moved on. This delay significantly reduces the impact that any feedback scribbled on the homework may have.

In this paper we include the teaching and learning aspects only as background motivation, since this is a system description and we have discussed the teaching and learning aspects elsewhere [7, 8]. A detailed description of the soundness and completeness results, which does not overlap with the present paper, can be found in a separate paper by From, Schlichtkrull and Villadsen [10].

Our main focus, then, is on the definition of the system itself and especially the benefits of building it on top of Isabelle/HOL. While many tools are implemented independently and perhaps modeled in a proof assistant, we aim to show the benefits of working entirely within Isabelle/HOL.

A completely novel development is the SeCaV Unshortener, a web application that allows experienced users to forgo full verification and in return write more succinct derivations. Such a derivation is automatically expanded into the full SeCaV syntax which can then be verified for correctness.

The screenshot shows the SeCaV Unshortener interface. At the top, there are buttons for "Sequent Calculus Verifier", "Help and Input Examples", "1:1", "Copy Output to Clipboard", and "SeCaV Unshortener 1.0". The main area is split into two panels. The left panel shows a compact proof:

```
(* A shortened proof *)
Imp p p
AlphaImp
  Neg p
  p
Ext
  p
  Neg p
Basic
```

The right panel shows the expanded proof:

```
proposition <p → p> by metis
text <
  Predicate numbers
  0 = p
  >

lemma <⊢
[
  Imp (Pre 0 []) (Pre 0 [])
]
>
proof -
  from AlphaImp have ?thesis if <⊢
  [
    Neg (Pre 0 []),
    Pre 0 []
  ]
  >
  using that by simp
  with Ext have ?thesis if <⊢
  [
    Pre 0 [],
    Neg (Pre 0 [])
  ]
  >
  using that by simp
  with Basic show ?thesis
  by simp
qed
```

Figure 1: The SeCaV Unshortener generating the example in Figure 6.

The SeCaV Unshortener, shown in Figure 1, allows proofs to be written in a much more compact syntax, cf. Section 5.

We used the SeCaV system in our BSc course “Logical Systems and Logic Programming” in the fall of 2020. 71 students took the 2-hour exam where the exercises in SeCaV were worth 20% of the grade. We also used the SeCaV system in our MSc course “Automated Reasoning” in the spring of 2021, mainly in order to bridge the gap between our micro provers for propositional logic and our Natural Deduction Assistant (NaDeA), cf. Section 2. Here, the students were also introduced to the recent SeCaV Unshortener. 34 students took the 2-hour exam where the exercises in SeCaV were worth 25% of the grade. Our experiences teaching these courses are detailed in a separate paper by Villadsen and Jacobsen [19].

The SeCaV system is available online (tested with Isabelle2020 and Isabelle2021):

<https://github.com/logic-tools/secav>

The two relevant files are `SeCaV.thy`, which defines the sequent calculus and proves soundness, and `Sequent_Calculus_Verifier.thy`, which builds on our existing work [9] to prove completeness (cf. Section 4).

The SeCaV Unshortener 1.0 is available online (tested using the Chrome, Edge, Firefox and Safari browsers):

<https://secav.compute.dtu.dk/>

Version 1.0 is fully functional and has an online tutorial with examples. The online tutorial can be used to learn how to actually use the SeCaV system, while the present paper is a description of the system.

We continue by discussing existing work (Section 2) before introducing our system and explaining a few design choices via a number of examples (Section 3). We then introduce SeCaV formally, explain our design considerations further, outline the soundness and completeness results, and emphasize the benefits of the Isabelle/HOL integration (Section 4). Next, we give an overview of the SeCaV Unshortener (Section 5) before concluding (Section 6).

We hope to convince the reader of the utility of SeCaV and its unshortener. In combination, we have both an online system where students can easily practice derivations in sequent calculus and a

transparent implementation of a proof checker for such derivations. The transparency means that the full implementation is accessible, from the specification of the syntax as simple datatypes to the definition of the side conditions as functional programs. Experienced students can even explore topics like soundness and completeness because Isabelle/HOL allows us to formalize semantics as well. In summary, we believe that SeCaV, its implementation in Isabelle/HOL and its unshortener provide an excellent set of tools for introducing the topic of sequent calculus to students.

2 Related Work

There are many tools for sequent calculus, both online and offline. We shall see that, while SeCaV defines one logic and one calculus, the transparency of the system makes the idea of extending or deviating from the system tangible. An exploration of how proof assistants in general make the different layers and elements of logic easier to distinguish has previously been conducted by From, Villadsen and Blackburn [8].

The web application Logitext (<http://logitext.mit.edu>) allows users to derive a sequent by clicking the connective they want to apply a rule to. As such, the rules are almost hidden away from the user who simply sees the appearance of new sub-derivations. Sequoia [14] allows users to input their own rules in a \LaTeX format and build derivations from them. It also checks certain meta-theoretical properties of the stated calculus. The online application was unavailable at the time of writing. The `Carnap.io` site [11] allows users to specify their own logic as well as proof system, but in Haskell, which is then compiled to a web application. The offline Sequent Calculus Trainer [6] guides the user away from dead ends by alerting them if the current sequent is determined to be unprovable. AXolotl [4, 5] is an Android app that supports sequent calculus derivations in a classical notation. It is designed to facilitate self-study. Unlike SeCaV, none of these tools provide any formal guarantees of their correctness. Each of them is a bespoke application in a regular programming language.

The Incredible Proof Machine [2, 3] is “an interactive visual theorem prover which represents proofs as port graphs.” It distinguishes itself by having a model of this proof representation formalized in Isabelle/HOL and shown to be as strong as natural deduction. Unlike SeCaV, the formalized metatheoretical results only apply to a model of the system instead of to the actual implementation.

Our Natural Deduction Assistant (NaDeA) [18] presents natural deduction in a more traditional style. Its metatheory is formalized in Isabelle/HOL and the web application supports exporting proofs that can be verified in Isabelle/HOL, alleviating the problem of potential bugs in the online tool.

Our Students’ Proof Assistant [15] exists entirely inside Isabelle/HOL, where it defines a proof assistant within the proof assistant. This helps make proof assistants and their design concrete, but makes the proving experience less natural than using the outer proof assistant directly as done in SeCaV.

Finally, we mention our micro provers for propositional logic [17] whose formalized soundness and completeness results take up only a few dozen lines of Isabelle/HOL. They are based on sequent calculus and can work as a first example in a course, before the full power of first-order logic and SeCaV is introduced.

3 Design Choices and Examples

Before we go into the details of our proof system, we will explain some of our design choices by simple examples, which will hopefully also give some intuition about how our system works and how it is used in practice.

We use a simple programming-like syntax for formulas in SeCaV and abbreviate it further in the SeCaV Unshortener. Both the syntax for SeCaV formulas and the abbreviated version are plain text formats, which makes them easy to write and avoids conflicts with logical operators in Isabelle/HOL.

Figure 2 gives an example derivation of the formula $p(a,b) \vee \neg p(a,b)$. The formula is stated on line 3 as the sole member of the one-sided sequent spanning lines 2–4. Recall that such a sequent is understood as a disjunction of formulas. On line 3, the disjunction \vee is written using the constructor *Dis* applied to two arguments separated by a space. For predicates, the constructor *Pre* takes a list of terms as arguments. Here we use *Fun 0* [] and *Fun 1* [], two function symbols taking no arguments, to represent the constants informally called *a* and *b*. We make this syntax precise in Section 4. Note that both predicates and functions take their arguments as a list, which makes it easy to write simple functions that manipulate them by pattern matching.

The first rule application in Figure 2 occurs on line 7. We will explain our proof rules later, but the interested reader may already now consult Figure 7 to see their definitions. We apply the *AlphaDis* rule backwards, stating that our goal follows from the sequent listed on lines 9–10. This sequent fits the shape of our *Basic* axiom since it starts with a formula that also occurs negated. Lines 14–15 finish the derivation based on this. Note that our proof rules work only on the first element of each sequent, which makes the definition of the deductive system by pattern matching on the list representing the sequent simple. The only exception is the structural rule *Ext*, which works on the entire sequent. We have chosen this representation over the common choice of representing sequents as sets to ensure that our definitions of e.g. side-conditions can be simple functional programs which users can actually execute to examine in detail why each step of their proof holds.

In the above we focused on the things essential to a human reader: the goal, the rules and their resulting sequents. The remaining lines and keywords are for the benefit of Isabelle/HOL: they fit our derivations into the Isar syntax [20] giving us all the verification benefits of Isabelle/HOL. If a rule is applied in a wrong manner, the editor tells us!

Our calculus is designed such that this boilerplate is predictable. With the exception of two rules that require clarification when more than one variable is in play, we have yet to encounter a rule application that cannot be justified by Isabelle/HOL's simplifier. This predictability means that we can write down only the essential parts of the derivation and then generate the boilerplate with the SeCaV Unshortener. Figure 3 contains an example of this, namely the same derivation as Figure 2. It starts with the goal formula on line 1, then the first rule application on line 3, followed by the resulting sequent, which spans lines 4–5, and finally line 6 finishes the derivation. In fact, the SeCaV Unshortener produced the Isabelle/HOL code in Figure 2 automatically from this representation. For brevity, we will generally favor the short representation.

At this point one might ask why we have not used Isabelle as a logical framework to implement SeCaV such that we can write proofs more directly instead of having to embed our deductive system within Isabelle/HOL. First and foremost, doing so would prevent us from formalizing soundness and completeness of the SeCaV system directly, since we would then no longer have a formal metalanguage to work in. Additionally, the implementation of such a system would be much more complex and include much more code to interface with the existing Isabelle system, obscuring the simplicity of the SeCaV system. We would like students to see that implementing a basic formal deductive system and proving it sound and complete is not as daunting as it may seem when looking at full-fledged, and thus very complicated, proof assistants such as Isabelle/HOL or Coq. This is of course a trade-off between ease of use for people who exclusively use the system to prove formulas and ease of understanding for people who would also like to understand why and how the system works. Since our students are using the system for both purposes, we cannot stray too far towards either side of this balance.

```

1 lemma ‹⊢
2   [
3     Dis (Pre 0 [Fun 0 [], Fun 1 []]) (Neg (Pre 0 [Fun 0 [], Fun 1 []]))
4   ]
5 ›
6 proof —
7   from AlphaDis have ?thesis if ‹⊢
8     [
9       Pre 0 [Fun 0 [], Fun 1 []],
10      Neg (Pre 0 [Fun 0 [], Fun 1 []])
11    ]
12 ›
13   using that by simp
14   with Basic show ?thesis
15   by simp
16 qed

```

Figure 2: A sample SeCaV derivation in Isabelle/HOL.

```

1 Dis p[a, b] (Neg p[a, b])
2
3 AlphaDis
4   p[a, b]
5   Neg p[a, b]
6 Basic

```

Figure 3: The sample SeCaV derivation in Figure 2 written in the syntax of the SeCaV Unshortener.

3.1 Instantiating Quantifiers

Consider the additional example in Figure 4, which contains a derivation of:

$$(\forall x. \forall y. p(x, y)) \rightarrow p(a, a)$$

We write the formulas using de Bruijn indices to match the Isabelle/HOL formalization. In the example, the variable 1 is bound by the outermost quantifier and 0 by the innermost. The rule application on line 3 is propositional and straightforward: the implication holds if either the antecedent does not or the consequent does. Consider instead line 6 where several things occur. First, the *GammaUni* rule allows us to derive a negated, universally quantified formula from an example. Applied backwards, we can insert any term for the bound variable while eliminating the quantifier. We do so, replacing the bound variable with the term *a*. Second, the notation *[a]* becomes a hint to Isabelle/HOL that *a* is the term used to replace the bound variable. This ensures that the simplifier can verify the correctness and is necessary when more than one variable occurs in the term (we omit it on line 9). Line 12 applies the *Ext* rule that rearranges the sequent such that the *Basic* rule applies on line 15. We insist that the entire sequent is written down after each rule application so it is possible to read each application without referring back to previous ones.

```

1 Imp (Uni (Uni (p[1, 0])) p[a, a]
2
3 AlphaImp
4   Neg (Uni (Uni p[1, 0]))
5   p[a, a]
6 GammaUni[a]
7   Neg (Uni p[a, 0])
8   p[a, a]
9 GammaUni
10  Neg p[a, a]
11  p[a, a]
12 Ext
13  p[a, a]
14  Neg p[a, a]
15 Basic

```

Figure 4: SeCaV Unshortener example with instantiation of quantifiers.

3.2 Branching Derivations

As a final example consider the longer Figure 5, which includes branching rules. Lines 1–24 proceed using rules similar to those we have already seen. We cover them in detail in Section 4. Line 25 applies the *BetaImp* rule, which relies on two sub-derivations. The first sequent that needs to be derived is given on lines 26–28 and the second sequent on lines 30–32, with a plus symbol (+) separating the two. The application of *Basic* on line 33 closes the first branch and the rest of the derivation concerns only the second one.

The order of the two branches is not important for the Isabelle/HOL verification and the subsequent rules can be applied to either of the branches or even both at the same time. For the sake of human-readability, however, we suggest working on the first branch.

Figure 5 displays another feature of our calculus that is worth pointing out. On line 37, the *Ext* rule is used not just to rearrange the sequent, but also to drop a formula that was only necessary on one branch. As such, it can be used to tidy up sequents during a derivation.

4 SeCaV

Figure 6 shows a SeCaV derivation as it appears in the Isabelle/jEdit editor. For brevity, we have removed a number of line breaks. The SeCaV Unshortener allows the same proof to be written in a much more compact syntax, cf. Figure 1.

In this section we formally describe the SeCaV system: its syntax and semantics, proof rules, metatheory and Isabelle/HOL integration.

4.1 Syntax and Semantics

The syntax of terms and formulas in SeCaV is formally defined by the following two Isabelle/HOL datatype declarations:


```

1  Imp (Uni (Imp p[0] q[0])) (Imp (Exi p[0]) (Exi q[0]))
2
3  AlphaImp
4    Neg (Uni (Imp p[0] q[0]))
5    Imp (Exi p[0]) (Exi q[0])
6  Ext
7    Imp (Exi p[0]) (Exi q[0])
8    Neg (Uni (Imp p[0] q[0]))
9  AlphaImp
10   Neg (Exi p[0])
11   Exi q[0]
12   Neg (Uni (Imp p[0] q[0]))
13 DeltaExi
14   Neg p[a]
15   Exi q[0]
16   Neg (Uni (Imp p[0] q[0]))
17 Ext
18   Neg (Uni (Imp p[0] q[0]))
19   Neg p[a]
20   Exi q[0]
21 GammaUni
22   Neg (Imp p[a] q[a])
23   Neg p[a]
24   Exi q[0]
25 BetaImp
26   p[a]
27   Neg p[a]
28   Exi q[0]
29 +
30   Neg q[a]
31   Neg p[a]
32   Exi q[0]
33 Basic
34   Neg q[a]
35   Neg p[a]
36   Exi q[0]
37 Ext
38   Exi q[0]
39   Neg q[a]
40 GammaExi
41   q[a]
42   Neg q[a]
43 Basic

```

Figure 5: SeCaV Unshortener example with a branching derivation.

```

lemma <⊢ [ Imp (Pre 0 []) (Pre 0 []) ] >
proof -
  from AlphaImp have ?thesis if <⊢ [ Neg (Pre 0 []), Pre 0 [] ] >
    using that by simp
  with Ext have ?thesis if <⊢ [ Pre 0 [], Neg (Pre 0 []) ] >
    using that by simp
  with Basic show ?thesis
    by simp
qed

```

Proof state Auto update Update Search: 100%

```

proof (chain)
picking this:
  ⊢ Neg ?p # ?q # ?z ⇒ ⊢ Imp ?p ?q # ?z

```

Figure 6: A compressed SeCaV derivation in the Isabelle/jEdit editor. The “Output” panel at the bottom shows the rule under the cursor (*AlphaImp*). Isabelle automatically instantiates each variable (*?p*, *?q*, *?z*) appropriately.

datatype *tm* = *Fun nat* (*tm list*) | *Var nat*

datatype *fm* = *Pre nat* (*tm list*) | *Imp fm fm* | *Dis fm fm* | *Con fm fm* | *Exi fm* | *Uni fm* | *Neg fm*

Terms are either functions identified by a natural number and applied to a list of terms, or variables identified by de Bruijn indices. Formulas are either predicates, also identified by a natural number and applied to a list of terms, a connective applied to an appropriate number of formulas, or a quantifier. This use of natural numbers as identifiers was chosen for simplicity, but we could also make the datatypes generic over the type of identifiers. In the Unshortener (cf. 5) we use strings of letters instead of numbers.

The embedding of SeCaV into Isabelle/HOL means that we do not need to write a parser for this syntax. As seen in Figure 6 we can write down a formula immediately. We use a simple programming-like syntax here, but it is also possible to define a more regular infix syntax with various precedences and associativity.

To formalize metatheory, like the soundness and completeness of our proof system, it is essential to assign a meaning to our formulas. We can do this because of our deep embedding of the syntax as a datatype. While we could also use Isabelle as the generic proof assistant it is, define our logic in that style and still have it check our proofs, doing so would prevent us from formalizing our metatheory, and we would not even be able to prove soundness.

The following functions interpret terms and formulas into Isabelle/HOL’s higher-order logic, given a variable assignment *e*, a function denotation *f* and a predicate denotation *g*:

primrec *semantics-term* and *semantics-list* where

⟨ *semantics-term e f* (*Var n*) = *e n* ⟩ |

⟨ *semantics-term e f* (*Fun i l*) = *f i* (*semantics-list e f l*) ⟩ |

⟨ *semantics-list e f* [] = [] ⟩ |

⟨ *semantics-list e f* (*t # l*) = *semantics-term e f t* # *semantics-list e f l* ⟩

primrec semantics where

$$\begin{aligned}
\langle \text{semantics } efg \text{ (Pre } i \ l) &= g \ i \ (\text{semantics-list } e \ f \ l) \rangle | \\
\langle \text{semantics } efg \text{ (Imp } p \ q) &= (\text{semantics } efg \ p \longrightarrow \text{semantics } efg \ q) \rangle | \\
\langle \text{semantics } efg \text{ (Dis } p \ q) &= (\text{semantics } efg \ p \vee \text{semantics } efg \ q) \rangle | \\
\langle \text{semantics } efg \text{ (Con } p \ q) &= (\text{semantics } efg \ p \wedge \text{semantics } efg \ q) \rangle | \\
\langle \text{semantics } efg \text{ (Exi } p) &= (\exists x. \text{semantics } (\text{shift } e \ 0 \ x) \ f \ g \ p) \rangle | \\
\langle \text{semantics } efg \text{ (Uni } p) &= (\forall x. \text{semantics } (\text{shift } e \ 0 \ x) \ f \ g \ p) \rangle | \\
\langle \text{semantics } efg \text{ (Neg } p) &= (\neg \text{semantics } efg \ p) \rangle
\end{aligned}$$

We use the given connectives and quantifiers from Isabelle/HOL’s higher-order logic to interpret our own, similarly to how semantics given using pen and paper uses natural language like “and” or “there exists.” The two quantifier clauses make the interpretation of de Bruijn indices explicit. When interpreting a quantifier, the function *shift* adjusts the variable assignment *e* so index 0 points at the newly quantified variable. Its general definition is:

definition $\langle \text{shift } e \ v \ x \equiv \lambda n. \text{if } n < v \text{ then } e \ n \text{ else if } n = v \text{ then } x \text{ else } e \ (n - 1) \rangle$

This use matches the intuition that variable 0 is bound by the “nearest” quantifier. Similarly, the existing indices are shifted by one since they appear one scope further out. Keeping this definition explicit makes some of the proofs needed for soundness more manageable.

Both here and later we will prefer to define explicit but simple recursive functions instead of using higher-order library functions such as *map* and *list.all* in our definitions. We find that, while these library functions make the definitions more succinct and understandable to experienced users of Isabelle/HOL, their use can easily confuse students who are not very good functional programmers. Another issue specific to Isabelle/HOL is that the definition of e.g. the *map* library function for lists is very cryptic, consisting only of the definition of the *list* datatype and a specialization of a more general function. Understanding how this works requires a deep dive into the internals of Isabelle/HOL, and since SeCaV is oriented towards beginners, we would like to keep this complexity out of sight.

4.2 Substitution

Returning to the topic of de Bruijn indices we now cover how substitution is formalized, as we need it to specify our proof rules. We include the definitions of such helper functions to make our presentation self-contained. Our substitution function on formulas, *sub*, is designed to be used whenever we instantiate a quantifier. The application *sub v s p* substitutes variable *v* for the term *s* in formula *p*. During this substitution, we ensure that no variable in *s* gets bound by a quantifier in *p*. We define the function by structural recursion:

primrec sub where

$$\begin{aligned}
\langle \text{sub } v \ s \ (\text{Pre } i \ l) &= \text{Pre } i \ (\text{sub-list } v \ s \ l) \rangle | \\
\langle \text{sub } v \ s \ (\text{Imp } p \ q) &= \text{Imp } (\text{sub } v \ s \ p) \ (\text{sub } v \ s \ q) \rangle | \\
\langle \text{sub } v \ s \ (\text{Dis } p \ q) &= \text{Dis } (\text{sub } v \ s \ p) \ (\text{sub } v \ s \ q) \rangle | \\
\langle \text{sub } v \ s \ (\text{Con } p \ q) &= \text{Con } (\text{sub } v \ s \ p) \ (\text{sub } v \ s \ q) \rangle | \\
\langle \text{sub } v \ s \ (\text{Exi } p) &= \text{Exi } (\text{sub } (v + 1) \ (\text{inc-term } s) \ p) \rangle | \\
\langle \text{sub } v \ s \ (\text{Uni } p) &= \text{Uni } (\text{sub } (v + 1) \ (\text{inc-term } s) \ p) \rangle | \\
\langle \text{sub } v \ s \ (\text{Neg } p) &= \text{Neg } (\text{sub } v \ s \ p) \rangle
\end{aligned}$$

Only the predicate and quantifier cases are interesting; the rest simply apply the substitution to the sub-formulas. In the predicate case we use the function *sub-list* to apply the substitution across the list of argument terms. It is defined mutually with *sub-term*:

	inductive sequent-calculus ($\langle \vdash - \rangle 0$) where
<i>Basic</i>	$\langle \vdash p \# z \rangle \mathbf{if} \langle member (Neg p) z \rangle $
<i>AlphaDis</i>	$\langle \vdash Dis p q \# z \rangle \mathbf{if} \langle \vdash p \# q \# z \rangle $
<i>AlphaImp</i>	$\langle \vdash Imp p q \# z \rangle \mathbf{if} \langle \vdash Neg p \# q \# z \rangle $
<i>AlphaCon</i>	$\langle \vdash Neg (Con p q) \# z \rangle \mathbf{if} \langle \vdash Neg p \# Neg q \# z \rangle $
<i>BetaCon</i>	$\langle \vdash Con p q \# z \rangle \mathbf{if} \langle \vdash p \# z \rangle \mathbf{and} \langle \vdash q \# z \rangle $
<i>BetaImp</i>	$\langle \vdash Neg (Imp p q) \# z \rangle \mathbf{if} \langle \vdash p \# z \rangle \mathbf{and} \langle \vdash Neg q \# z \rangle $
<i>BetaDis</i>	$\langle \vdash Neg (Dis p q) \# z \rangle \mathbf{if} \langle \vdash Neg p \# z \rangle \mathbf{and} \langle \vdash Neg q \# z \rangle $
<i>GammaExi</i>	$\langle \vdash Exi p \# z \rangle \mathbf{if} \langle \vdash sub\ 0\ t\ p \# z \rangle $
<i>GammaUni</i>	$\langle \vdash Neg (Uni p) \# z \rangle \mathbf{if} \langle \vdash Neg (sub\ 0\ t\ p) \# z \rangle $
<i>DeltaUni</i>	$\langle \vdash Uni p \# z \rangle \mathbf{if} \langle \vdash sub\ 0\ (Fun\ i\ [])\ p \# z \rangle \mathbf{and} \langle news\ i\ (p \# z) \rangle $
<i>DeltaExi</i>	$\langle \vdash Neg (Exi p) \# z \rangle \mathbf{if} \langle \vdash Neg (sub\ 0\ (Fun\ i\ [])\ p) \# z \rangle \mathbf{and} \langle news\ i\ (p \# z) \rangle $
<i>NegNeg</i>	$\langle \vdash Neg (Neg p) \# z \rangle \mathbf{if} \langle \vdash p \# z \rangle $
<i>Ext</i>	$\langle \vdash y \rangle \mathbf{if} \langle \vdash z \rangle \mathbf{and} \langle ext\ y\ z \rangle$

Figure 7: SeCaV proof rules in Isabelle/HOL with associated names inserted manually to the left.

primrec sub-term and sub-list where

$\langle sub\ term\ v\ s\ (Var\ n) = (if\ n < v\ then\ Var\ n\ else\ if\ n = v\ then\ s\ else\ Var\ (n - 1)) \rangle |$
 $\langle sub\ term\ v\ s\ (Fun\ i\ l) = Fun\ i\ (sub\ list\ v\ s\ l) \rangle |$
 $\langle sub\ list\ v\ s\ [] = [] \rangle |$
 $\langle sub\ list\ v\ s\ (t \# l) = sub\ term\ v\ s\ t \# sub\ list\ v\ s\ l \rangle$

There are two cases for *sub-term*. At variables we leave smaller variables alone, substitute those matching the target index v , and decrement larger variables to account for the instantiated quantifier whose scope is now gone. At function symbols, we simply apply the substitution across the arguments.

Returning to *sub*, in the quantifier cases we increment v to account for the quantifier whose scope we are now under. For the same reason, we use the function *inc-term* to increment the variables in s .

It is defined mutually with *inc-list*:

primrec inc-term and inc-list where

$\langle inc\ term\ (Var\ n) = Var\ (n + 1) \rangle |$
 $\langle inc\ term\ (Fun\ i\ l) = Fun\ i\ (inc\ list\ l) \rangle |$
 $\langle inc\ list\ [] = [] \rangle |$
 $\langle inc\ list\ (t \# l) = inc\ term\ t \# inc\ list\ l \rangle$

With these at hand, we can now turn to the proof system itself.

4.3 Proof System

Our sequent calculus is a one-sided system like System G by Ben-Ari [1], which inspired it. A one-sided system has a couple of advantages. First, it can be explained and understood as simply meta-notation for a disjunction between formulas. Second, it can be formalized as a single list of formulas, in turn reducing the syntactic burden of writing down a sequent. Consider the SeCaV Unshortener syntax in e.g. Figure 4. Rule applications and sequents alternate throughout the derivation, with no need for a special symbol to distinguish a left- and right-hand side of the sequent as would be needed in a two-sided system [1, p. 69].

Figure 7 contains our proof rules. We use Smullyan's uniform notation [16] for the names, designating whether they branch (β) or not (α) and whether the quantifiers can be built from any term (γ) or only a fresh witness (δ). Notice that each rule is actually a schema: the symbols p and q etc. are not concrete formulas but metavariables that can be instantiated with any type-correct value.

4.3.1 Proof Rules

As mentioned our sequents are lists of formulas, which means that they are ordered. In Isabelle/HOL, the symbol # separates the head and tail of a list. All our rules except *Ext* use this notation to replace the head of the list.

Figure 7 begins with the only axiom, *Basic*, which states that a sequent with some formula p at the head and $Neg\ p$ somewhere in the tail can be derived. That is, we can derive the sequent $\vdash p \# z$, **if** we can demonstrate that $Neg\ p$ is a member of z , i.e. $member\ (Neg\ p)\ z$.

The function *member* is defined in SeCaV as a simple primitive recursive function on lists for users to inspect (or even run):

primrec member where

```
⟨ member p [] = False ⟩ |
⟨ member p (q # z) = (if p = q then True else member p z) ⟩
```

Since a sequent is understood as a disjunction, those of the *Basic* shape are clearly valid. To derive a sequent that contains both some p and a corresponding $Neg\ p$ but not necessarily in the order dictated by *Basic*, the final rule in Figure 7, *Ext*, can be used. As we have seen in Section 3, it allows one to rearrange the formulas of a sequent or to drop formulas. It should be read as follows: if we can derive a sequent z and the sequent y is an *extension* of z , then we are allowed to derive y itself. The function *ext* builds on *member* to check that the formulas in y constitute a superset of those in z :

primrec ext where

```
⟨ ext y [] = True ⟩ |
⟨ ext y (p # z) = (if member p y then ext y z else False) ⟩
```

Alpha Rules After *Basic* follow three α -rules that rely on just one sub-derivation. The *AlphaDis* rule moves the connective from the object language into the metalanguage, simply removing the connective and adding the two disjuncts to the sequent. To show that an implication $Imp\ p\ q$ holds, we can either falsify the antecedent, $Neg\ p$, or show the conclusion, q , so the rule *AlphaImp* replaces an implication with exactly those formulas. Finally, *AlphaCon* states that to show $Neg\ (Con\ p\ q)$ we can falsify either p or q . The pen-ultimate rule *NegNeg* also relies on just one sub-derivation, so we include it here. It introduces a double negation.

Beta Rules After the first α -rules follow three β -rules that make the derivation branch. A conjunction only holds if both conjuncts do, so the *BetaCon* rule adds each to separate sub-derivations. The *BetaImp* rule works on a negated implication, $Neg\ (Imp\ p\ q)$, and states that we must both prove p and falsify q . Finally, *BetaDis* replaces $Neg\ (Dis\ p\ q)$ with both $Neg\ p$ and $Neg\ q$ on separate branches, as both p and q must be falsified for their disjunction to be falsified.

Gamma Rules The γ -rules apply to formulas that are effectively existentially quantified. Such formulas can be built from any witnessing term. The next rule exemplifies this: *GammaExi* derives the sequent $Exi\ p\ \# z$ from $sub\ 0\ t\ p\ \# z$. In the sub-derivation, we have the formula p with its outermost variable instantiated with the term t using the *sub* function. This term, t , witnesses the existence, so in the conclusion we quantify over the variable, giving $Exi\ p$, instead of substituting it.

The *GammaUni* rule applies when the head of the sequent is $Neg\ (Uni\ p)$ for some formula p . In the sub-derivation, the head is replaced by $Neg\ (sub\ 0\ t\ p)$ since falsifying p instantiated with any term t is enough to falsify $Uni\ p$.

Delta Rules The two δ -rules apply to formulas that are effectively universally quantified. To prove a universal quantifier, we cannot abstract over just any term like with γ -rules. Instead, the term must be arbitrary, i.e. *new* to the sequent as formalized below, so that any other term could stand in its place. Sometimes this is called *fresh* rather than *new*.

The *DeltaUni* rule allows the derivation of $Uni\ p\ \#z$ if we can derive $sub\ 0\ (Fun\ i\ [])\ p\ \#z$ where the name i does not occur in either p or z , as checked by $news\ i\ (p\ \#z)$. The *DeltaExi* rule is similar but applies when the head of the sequent is $Neg\ (Exi\ p)$.

We define newness similarly to the other side conditions. The function *new* defines what it means for a function symbol c to be new to a formula:

primrec new where

```

⟨ 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 ⟩ |
⟨ new c (Neg p) = new c p ⟩

```

Only the predicate case is interesting; the rest simply consider sub-formulas. The function *new-list* checks whether c is new to a list of terms. It is defined mutually with *new-term*:

primrec new-term and new-list where

```

⟨ new-term c (Var n) = True ⟩ |
⟨ new-term c (Fun i l) = (if i = c then False else new-list c l) ⟩ |
⟨ new-list c [] = True ⟩ |
⟨ new-list c (t # l) = (if new-term c t then new-list c l else False) ⟩

```

If the term is a variable then the function symbol c is obviously new. Otherwise the term is a function application and we check whether the two function symbols coincide. If they do, c is not new, but even if they do not, c still has to be new to the arguments of the function, which we check with *new-list*.

The entry point to these functions is *news*, which checks whether the function symbol c is new to the given sequent:

primrec news where

```

⟨ news c [] = True ⟩ |
⟨ news c (p # z) = (if new c p then news c z else False) ⟩

```

4.3.2 Rule Design

After seeing how the proof rules work, we want to point out several choices in their design.

While sequents are sometimes unordered (cf. Ben-Ari [1], Nipkow and Michaelis [12]) ours do have an order. Where Ben-Ari underlines the formula in a sequent that the next rule applies to, our rules always work on the first one. This simplification has several benefits: (i) it makes the formalization simpler to state and the success of the verification easier to predict, (ii) it reduces the notational burden in the SeCaV Unshortener syntax and (iii) it provides a straight-forward recipe for new users to get started: “simply look at the first formula and see if any rules apply.” Of course, the recipe in (iii) may result in derivations that are longer than necessary and because of our γ -rules the recipe may even be insufficient for certain formulas, but such formulas are also out of reach if the user starts out overwhelmed and never gets going. The simplification forces us to include a structural rule like *Ext*. This is the price of separating concerns, but as we have seen, *Ext* can also, for instance, be used to drop formulas on branches that do not need them.

```

lemma ⟨ $\vdash$ 
  [
    GOAL
  ]
⟩
proof –
  from RULE1 have ?thesis if ⟨ $\vdash$ 
  [
    SUBGOAL1
    ⋮
    SUBGOALN
  ]
⟩
  using that by simp
    ⋮
  with Basic show ?thesis
  by simp
qed

```

Figure 8: SeCaV derivation template. Branches are added using the **and** keyword between sequents.

Another point is that our rules do not just add to the sequent, but always replace the head of it in the sub-derivation(s). Even the γ -rules do this, even though we may want to instantiate such formulas with several different terms (cf. Ben-Ari [1]). Again we separate concerns: to apply a γ -rule twice, first duplicate the formula with *Ext* and then apply the rule. This makes such duplication deliberate instead of an arbitrary feature of γ -rules that is sometimes useful and sometimes not.

Our last point relates to the benefit of specifying our system in Isabelle/HOL: every operation and side condition is explicitly spelled out and *computational*. It may seem obvious what membership in a sequent entails or what it means for a constant name to be *new*, but something like substitution is notoriously tricky, no matter the representation. In our system, these things are implemented by simple functional programs, accessible directly in the system. They are not opaque pieces of natural language or hidden away in an implementation, but can be inspected by the user and even run on simple examples.

It speaks to the complexity of substitution that only rules that involve this operation can be hard to verify: *GammaUni* and *GammaExi*. Otherwise, our definitions, like those of *member* and *ext*, play on the strengths of the Isabelle/HOL simplifier: they are simple functional programs that can be checked by rewriting. Similarly, by letting our rules work on the first formula in the sequent, it becomes a simple problem to unify it with the current goal, solving the meta-variables to check if they match the stated sub-derivation. These design choices make the system predictable to work with.

4.4 Writing Proofs

As alluded to in Section 3, derivations in SeCaV follow a common template, which we have sketched in Figure 8. Users of the system only need to worry about filling in the *GOAL* and a number of *RULE* applications with corresponding *SUBGOALS*. Those curious about Isabelle/HOL can investigate the meaning of the remaining keywords if they want to. Since derivations are entirely textual, it is easy to copy this template, or parts of it, from given examples or previous derivations.

```

Failed to finish proof:
goal (1 subgoal):
1 I. ( $\wedge p q z. \vdash \text{Neg } p \# q \# z \implies \vdash \text{Imp } p q \# z$ )  $\implies$ 
2    $\vdash [\text{Pre } 0 [\text{Fun } 0 [], \text{Fun } (\text{Suc } 0) []],$ 
3      $\text{Neg } (\text{Pre } 0 [\text{Fun } 0 [], \text{Fun } (\text{Suc } 0) []]) \implies$ 
4    $\vdash [\text{Dis } (\text{Pre } 0 [\text{Fun } 0 [], \text{Fun } (\text{Suc } 0) []])$ 
5      $(\text{Neg } (\text{Pre } 0 [\text{Fun } 0 [], \text{Fun } (\text{Suc } 0) []]))]$ 

```

Figure 9: Error message when *AlphaDis* is replaced by *AlphaImp* in Figure 2.

In Figure 9 we see the error message obtained when *AlphaDis* is replaced by *AlphaImp* in Figure 2. Isabelle/HOL will highlight the **by** keyword following the rule application, notifying the user that something is wrong. The error is then displayed in the output panel when placing the cursor over the highlight. Line 1 in Figure 9 contains the rule being applied, lines 2–3 the stated subgoal and lines 4–5 the goal itself. By inspection we see that since *Imp* and *Dis* do not match, the rule does not apply to produce the goal (the subgoal does not match either).

We obtain this error message completely for free by leveraging Isabelle/HOL as the platform for specifying our system.

We also inherit the interactive features of Isabelle/HOL. Users can click a name and be taken to its definition, e.g. that of *sub* if in doubt about substitution. Or when following the template, they can put their cursor on an applied rule and see its definition in the output panel as in Figure 6: “This derivation uses *AlphaImp*, how does that look again? Oh, right: $\vdash \text{Neg } ?p \# ?q \# ?z \implies \vdash \text{Imp } ?p ?q \# ?z$.”

4.5 Soundness and Completeness

Having specified SeCaV in a proof assistant enables us to give certain guarantees about not just our calculus but its implementation as well. The first and most obvious is soundness of the rules. If we can derive a sequent, then for any interpretation, some formula in the sequent is satisfied:

theorem *sound*: $\langle \vdash z \implies \exists p \in \text{set } z. \text{ semantics } e f g p \rangle$

See the formalization for the proof which works by induction over the rules and using a substitution lemma. We immediately obtain that if a derivable sequent contains just one formula then that formula must be valid:

corollary $\langle \vdash [p] \implies \text{ semantics } e f g p \rangle$

We build the completeness proof on existing work in the Archive of Formal Proofs, namely the entry “A Sequent Calculus for First-Order Logic” [9] (though this work also contains a soundness proof, we prefer to keep the soundness proof free of external dependencies since showing soundness is not a very difficult undertaking). In less than a hundred lines of Isabelle/HOL, we relate our syntax, semantics, side conditions and operations to an existing sequent calculus formalization and show that derivations in that one (\vdash) can be translated into ours (\vdash) (cf. the formalization):

lemma *sim*: $\langle (\vdash x) \implies (\vdash (\text{map to-fm } x)) \rangle$

From these components, completeness follows straightforwardly:

theorem *complete-sound*: $\langle \gg p \implies \vdash [p] \rangle \langle \vdash [q] \implies \text{ semantics } e f g q \rangle$


```

(* A shortened proof - with a mistake *)
Imp p p
AlphaImp
  Neg p
  p
Basic

(* Warning:
(Basic) Sequent not a tautology/Positive formula is not the first
*)
proposition <p → p> by metis

text <
  Predicate numbers
  0 = p
>

lemma <⊢
[
  Imp (Pre 0 []) (Pre 0 [])
]
>
proof -
  from AlphaImp have ?thesis if <⊢
  [
    Neg (Pre 0 []),
    Pre 0 []
  ]
  >
  using that by simp
  with Basic show ?thesis
  by simp
qed

```

Figure 10: The SeCaV Unshortener generating the example in Figure 6 — With a mistake.

The symbol \gg abbreviates validity in the universe of Herbrand terms. Validity in just this universe is enough to show the existence of a derivation (a slightly stronger completeness result than assuming validity in all universes). The soundness result, conversely, implies validity in any universe, as e, f and g can be picked at will.

These aspects can be ignored when working with the system, but used to concretize discussions of soundness and completeness in a course.

4.6 Isabelle/HOL Integration

Next, we want to reiterate a few consequences of building our system on top of Isabelle/HOL.

In terms of infrastructure we are relieved from implementing a lot of work ourselves. By giving two simple datatype declarations, in a syntax resembling BNF, we get to reuse Isabelle/HOL's parser when writing formulas in our object logic. The same reusability applies to the proof system, both in its declarative specification using the **inductive** command and when writing concrete derivations. Given the declaration in Figure 7, we inherit proof checking completely for free: Isabelle/HOL verifies the correctness of derivations for us.

The use of Isabelle/HOL also means that we can reuse its mature graphical editor Isabelle/jEdit. Besides regular editor features like undo, Isabelle/jEdit continually checks the correctness of what the user enters. As seen, it produces decent errors that are displayed in the same window as the derivation and where the offending rule application is highlighted directly in the derivation. Finally, all definitions used by the system can be inspected by using the editor to look them up within the same interface.

5 SeCaV Unshortener

While the embedding of SeCaV into Isabelle/HOL makes it possible for users to get quick feedback on their proofs and provides us with an editor for free, actually writing out the proofs in the Isabelle/HOL syntax can become tedious. To remedy this, we have introduced the SeCaV Unshortener, shown in Figure 1 and Figure 10. It allows proofs to be written in a much more compact syntax that resembles the

style one might use when writing pen-and-paper proofs. For instance, instead of $\text{Fun } 0 []$, we can simply write a and it will be “unshortened” for us.

The SeCaV Unshortener is a web application whose main page consists of two panes. The first pane is a text area in which the user can write proofs in the compact SeCaV Unshortener syntax. The second pane contains the result of “unshortening” the proofs written in the first pane into the full SeCaV syntax, ready to be copied into Isabelle/HOL for verification. For each proof, the SeCaV Unshortener also generates a representation of the statement in usual logical syntax and a mapping from predicate and function names to the natural numbers used in the full SeCaV syntax. The first of these is useful to detect misunderstandings in the statement to be proved, while the latter is needed to relate the actual proof to the representation in usual logical syntax, and may also be used to quickly detect typos in names. The second pane reacts to changes in the first pane in real time, and will contain an error message if a proof is written using wrong syntax or if proof rules are applied in a wrong manner.

Along the top of the main page is a link to a page containing extensive help and a number of examples, an indication of the currently selected line and column in the first pane (for use with error messages), and a button that copies the unshortened proof to the clipboard.

The SeCaV Unshortener provides a canonical formatting of proofs and a lighter, more readable syntax at the expense of introducing another step in the proof procedure. The SeCaV Unshortener does not verify the proofs entered into it to the same degree as Isabelle/HOL but it does add warnings when the proofs will most likely be rejected by Isabelle/HOL. Examples of these warnings include forgetting to actually change the sequent after a rule application, changing the sequent in a way that is inconsistent with the latest rule application, misapplying the *Basic* rule and forgetting the side conditions on δ -rules.

The SeCaV Unshortener is implemented in PureScript using the Concur web UI framework with a React backend. The application is compiled down to a few JavaScript, HTML, and CSS files, which can be hosted on any web server or downloaded for local use. The system basically consists of a parser for the Unshortener syntax and a generator for the full SeCaV syntax. After parsing the user input, the abstract syntax tree is checked for errors and warnings are added to the generated SeCaV syntax.

6 Conclusion

We have introduced SeCaV, a sequent calculus verifier built on top of Isabelle/HOL, and explained the syntax, semantics, and proof rules of the system. SeCaV is designed to be easy to learn and understand for students, and is therefore implemented as a number of simple functional programs utilizing the interactive Isabelle/jEdit editor to allow inspection of every part of the system. We have used Isabelle/HOL to prove soundness and completeness of the SeCaV calculus exactly as users work with it.

We have also introduced the SeCaV Unshortener, a web application that allows users of SeCaV to omit the boilerplate notation needed for the embedding in Isabelle/HOL. We are currently (fall 2021) using the SeCaV system in our BSc course “Logical Systems and Logic Programming” and many of the 77 students prefer the SeCaV Unshortener with the lighter, more readable syntax. We plan to release the teaching material as soon as possible.

Acknowledgements

We thank Agnes Moesgård Eschen, Alexander Birch Jensen, Simon Tobias Lund, Emmanuel André Ryom and Anders Schlichtkrull for comments on a draft. We also thank the anonymous reviewers,

whose comments and suggestions have materially improved the text and caused us to reassess several aspects of our system.

References

- [1] Mordechai Ben-Ari (2012): *Mathematical Logic for Computer Science*. Springer, doi:10.1007/978-1-4471-4129-7.
- [2] Joachim Breitner (2016): *Visual Theorem Proving with the Incredible Proof Machine*. In Jasmin Christian Blanchette & Stephan Merz, editors: *Interactive Theorem Proving - 7th International Conference, ITP 2016, Nancy, France, August 22-25, 2016, Proceedings, Lecture Notes in Computer Science 9807*, Springer, pp. 123–139, doi:10.1007/978-3-319-43144-4_8.
- [3] Joachim Breitner & Denis Lohner (2016): *The meta theory of the Incredible Proof Machine*. *Archive of Formal Proofs*. https://isa-afp.org/entries/Incredible_Proof_Machine.html, Formal proof development.
- [4] David M. Cerna, Rafael P. D. Kiesel & Alexandra Dzhiganskaya (2019): *A Mobile Application for Self-Guided Study of Formal Reasoning*. In Pedro Quaresma, Walther Neuper & João Marcos, editors: *Proceedings 8th International Workshop on Theorem Proving Components for Educational Software, ThEdu@CADE 2019, Natal, Brazil, 25th August 2019, EPTCS 313*, pp. 35–53, doi:10.4204/EPTCS.313.3.
- [5] David M. Cerna, Martina Seidl, Wolfgang Schreiner, Wolfgang Windsteiger & Armin Biere (2020): *Aiding an Introduction to Formal Reasoning Within a First-Year Logic Course for CS Majors Using a Mobile Self-Study App*. In Michail N. Giannakos, Guttorm Sindre, Andrew Luxton-Reilly & Monica Divitini, editors: *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education, ITiCSE 2020, Trondheim, Norway, June 15-19, 2020, ACM*, pp. 61–67, doi:10.1145/3341525.3387409.
- [6] Arno Ehle, Norbert Hundeshagen & Martin Lange (2017): *The Sequent Calculus Trainer with Automated Reasoning - Helping Students to Find Proofs*. In Pedro Quaresma & Walther Neuper, editors: *Proceedings 6th International Workshop on Theorem proving components for Educational software, ThEdu@CADE 2017, Gothenburg, Sweden, 6 August 2017, EPTCS 267*, pp. 19–37, doi:10.4204/EPTCS.267.2.
- [7] Asta Halkjær From, Alexander Birch Jensen, Anders Schlichtkrull & Jørgen Villadsen (2019): *Teaching a Formalized Logical Calculus*. In Pedro Quaresma, Walther Neuper & João Marcos, editors: *Proceedings 8th International Workshop on Theorem Proving Components for Educational Software, ThEdu@CADE 2019, Natal, Brazil, 25th August 2019, EPTCS 313*, pp. 73–92, doi:10.4204/EPTCS.313.5.
- [8] Asta Halkjær From, Jørgen Villadsen & Patrick Blackburn (2020): *Isabelle/HOL as a Meta-Language for Teaching Logic*. In Pedro Quaresma, Walther Neuper & João Marcos, editors: *Proceedings 9th International Workshop on Theorem Proving Components for Educational Software, ThEdu@IJCAR 2020, Paris, France, 29th June 2020, EPTCS 328*, pp. 18–34, doi:10.4204/EPTCS.328.2.
- [9] Asta Halkjær From (2019): *A Sequent Calculus for First-Order Logic*. *Archive of Formal Proofs*. https://isa-afp.org/entries/FOL_Seq_Calc1.html, Formal proof development.
- [10] Asta Halkjær From, Anders Schlichtkrull & Jørgen Villadsen (2021): *A Sequent Calculus for First-Order Logic Formalized in Isabelle/HOL*. In Stefania Monica & Federico Bergenti, editors: *Proceedings of the 36th Italian Conference on Computational Logic - CILC 2021, Parma, Italy, September 7-9, 2021, CEUR Workshop Proceedings 3002*, CEUR-WS.org, pp. 107–121. Available at <http://ceur-ws.org/Vol-3002/paper7.pdf>.
- [11] Graham Leach-Krouse (2017): *Carnap: An Open Framework for Formal Reasoning in the Browser*. In Pedro Quaresma & Walther Neuper, editors: *Proceedings 6th International Workshop on Theorem proving components for Educational software, ThEdu@CADE 2017, Gothenburg, Sweden, 6 August 2017, EPTCS 267*, pp. 70–88, doi:10.4204/EPTCS.267.5.

- [12] Julius Michaelis & Tobias Nipkow (2018): *Formalized Proof Systems for Propositional Logic*. In A. Abel, F. Nordvall Forsberg & A. Kaposi, editors: *23rd International Conference on Types for Proofs and Programs (TYPES 2017)*, LIPIcs 104, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, pp. 6:1–6:16, doi:10.4230/LIPIcs.TYPES.2017.5.
- [13] Tobias Nipkow (2012): *Teaching Semantics with a Proof Assistant: No More LSD Trip Proofs*. In Viktor Kunčak & Andrey Rybalchenko, editors: *Verification, Model Checking, and Abstract Interpretation*, Springer, pp. 24–38, doi:10.1007/978-3-642-27940-9_3.
- [14] Giselle Reis, Zan Naeem & Mohammed Hashim (2020): *Sequoia: A Playground for Logicians - (System Description)*. In Nicolas Peltier & Viorica Sofronie-Stokkermans, editors: *Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part II, Lecture Notes in Computer Science 12167*, Springer, pp. 480–488, doi:10.1007/978-3-030-51054-1_32.
- [15] Anders Schlichtkrull, Jørgen Villadsen & Andreas Halkjær From (2018): *Students' Proof Assistant (SPA)*. In Pedro Quaresma & Walther Neuper, editors: *Proceedings 7th International Workshop on Theorem proving components for Educational software, ThEdu@FLoC 2018, Oxford, United Kingdom, 18 July 2018, EPTCS 290*, pp. 1–13, doi:10.4204/EPTCS.290.1.
- [16] Raymond M. Smullyan (1995): *First-Order Logic*. Dover Publications.
- [17] Jørgen Villadsen (2020): *Tautology Checkers in Isabelle and Haskell*. In Francesco Calimeri, Simona Perri & Ester Zumpano, editors: *Proceedings of the 35th Italian Conference on Computational Logic - CILC 2020, Rende, Italy, October 13-15, 2020, CEUR Workshop Proceedings 2710*, CEUR-WS.org, pp. 327–341. Available at <http://ceur-ws.org/Vol-2710/paper21.pdf>.
- [18] Jørgen Villadsen, Andreas Halkjær From & Anders Schlichtkrull (2018): *Natural Deduction Assistant (NaDeA)*. In Pedro Quaresma & Walther Neuper, editors: *Proceedings 7th International Workshop on Theorem proving components for Educational software, ThEdu@FLoC 2018, Oxford, United Kingdom, 18 July 2018, EPTCS 290*, pp. 14–29, doi:10.4204/EPTCS.290.2.
- [19] Jørgen Villadsen & Frederik Krogsdal Jacobsen (2021): *Using Isabelle in Two Courses on Logic and Automated Reasoning*. In João F. Ferreira, Alexandra Mendes & Claudio Menghi, editors: *Formal Methods Teaching*, Springer International Publishing, Cham, pp. 117–132, doi:10.1007/978-3-030-91550-6_9.
- [20] Makarius Wenzel (2007): *Isabelle/Isar - a generic framework for human-readable proof documents*. *From Insight to Proof - Festschrift in Honour of Andrzej Trybulec, Studies in Logic, Grammar, and Rhetoric*. University of Białystok 10(23), pp. 277–298.



Using Isabelle in Two Courses on Logic and Automated Reasoning

Jørgen Villadsen^(✉)  and Frederik Krogsdal Jacobsen 

Technical University of Denmark, Kongens Lyngby, Denmark
jovi@dtu.dk

Abstract. We present our experiences teaching two courses on formal methods and detail the contents of the courses and their positioning in the curriculum. The first course is a bachelor course on logical systems and logic programming, with a focus on classical first-order logic and automatic theorem proving. The second course is a master course on automated reasoning, with a focus on classical higher-order logic and interactive theorem proving. The proof assistant Isabelle is used in both courses, using Isabelle/Pure as well as Isabelle/HOL. We describe our online tools to be used with Isabelle/HOL, in particular the Sequent Calculus Verifier (SeCaV) and the Natural Deduction Assistant (NaDeA). We also describe our innovative Students' Proof Assistant which is formally verified in Isabelle/HOL and integrated in Isabelle/jEdit using Isabelle/ML.

Keywords: Logic · Automated reasoning · Proof assistant Isabelle

1 Introduction

We present our experiences teaching two courses on formal methods at the Technical University of Denmark (DTU):

- BSc Course: DTU Course 02156 Logical Systems and Logic Programming
<https://kurser.dtu.dk/course/02156>
- MSc Course: DTU Course 02256 Automated Reasoning
<https://kurser.dtu.dk/course/02256>

Both courses are taught in English. Figure 1 shows the objectives and content of the two courses. The objectives need to be approved by the study board and are not expected to change much from year to year. The above links also include some official statistics like evaluations and grades but mostly in Danish. Both courses count for 5 ECTS points, which corresponds to approximately 2 h of lectures and 2 h of group exercise sessions per week, plus individual study and assignment work (expected around 9 h per week in total), for 13 weeks (summing up to 140 h with exam preparations).

(a) Objectives and content of the course Logical Systems and Logic Programming.

General course objectives

The aim of the course is to give the students an introduction to some of the basic declarative formalisms from formal computer science and logic that can be used for describing, analysing and constructing IT systems. It will cover theoretical insight as well as practical skills in relevant high-level programming languages.

Learning objectives

A student who has met the objectives of the course will be able to:

- relate different kinds of proof systems
- construct formal proofs in elementary logics
- exploit selected classical and non-classical logics
- use the backtracking algorithm for simple problem solving
- analyze the effect of a declarative program
- establish a functional design for a given problem, so that the main concepts of the problem are directly traceable in the design
- master logical approaches to programming in terms of defining recursive predicates
- communicate solutions to problems in a clear and precise manner

Content

The course covers logic programming (in particular Prolog as a rapid prototyping tool), elementary logics (including propositional and first-order logic), proof systems (deductive systems and/or refutation systems), and problem solving techniques (for instance the backtracking algorithm).

(b) Objectives and content of the course Automated Reasoning.

General course objectives

Reasoning is the ability to make logical inferences. The aim of the course is to give the students an introduction to automatic and interactive computer systems for reasoning about mathematical theorems as well as properties of IT systems. It will cover theoretical insight as well as practical skills in relevant proof assistants.

Learning objectives

A student who has met the objectives of the course will be able to:

- explain the basic concepts introduced in the course
- express mathematical theorems and properties of IT systems formally
- master the natural deduction proof system
- relate first-order logic, higher-order logic and type theory
- construct formal proofs in the procedural style and in the declarative style
- use automatic and interactive computer systems for automated reasoning
- evaluate the trustworthiness of proof assistants and related tools
- communicate solutions to problems in a clear and precise manner

Content

The natural deduction proof system, first-order logic, higher-order logic and type theory. Formal proofs in the procedural style and in the declarative style using automatic and interactive provers. The Isabelle proof assistant in artificial intelligence and computer science.

Fig. 1. Objectives and course content of the two courses.

The first course is on logical systems and logical programming, and is intended for final-year BSc students (over the years interested students have successfully taken it already at the start of the second year of their bachelor). The course has been given more or less in the same format since 2006 with an increasing number of students, currently around 80 students per year.

The second course is on automated reasoning, and is intended for MSc students (interested students have successfully taken it already during the final year of their bachelor). The course was given for the first time in 2020 and has around 40 students per year.

The main changes due to COVID-19 were online lessons using Zoom and online home exams instead of physical at DTU. We did not make any other changes to the courses and we will not elaborate on the COVID-19 situation in the present paper.

Both of the courses use the proof assistant Isabelle [30] to showcase verified proof systems and provers, which we have implemented in Isabelle. This allows us to discuss common proof methods for e.g. soundness and completeness and allows students to experiment with larger proofs without losing track of what is going on. We also use Isabelle for assignments and exam questions concerning these proof systems. This allows students to get immediate feedback from the proof assistant, and allows us to easily check if the submitted proofs are correct. Recent research indicates that quick formative evaluation has a large impact on learning when teaching introductory computer science [16]. We use Isabelle since it is the proof assistant that we know best and because Isabelle is a generic proof assistant which allows us to use both Isabelle/HOL and Isabelle/Pure as detailed in later sections.

The BSc course additionally revolves heavily around the Prolog programming language, on which we spend around half of the time. Students thus learn to couple logical programming with logic, and we showcase many interesting programs related to the rest of the course content. The MSc course focuses more on functional programming within the Isabelle proof assistant, and how this can be coupled to formal methods and proofs about programs.

To enable this use of Isabelle and Prolog, we need students to hit the ground running so they can use the implementations of the logical concepts from the beginning. We recall the following quote from Donald Knuth:

When certain concepts of $T_{E}X$ are introduced informally, general rules will be stated; afterwards you will find that the rules aren't strictly true. In general, the later chapters contain more reliable information than the earlier ones do. The author feels that this technique of deliberate lying will actually make it easier for you to learn the ideas. Once you understand a simple but false rule, it will not be hard to supplement that rule with its exceptions.

For Isabelle and Prolog, we throw the students into the deep end and return later to explain how everything actually fits together. Unification, for example, is treated informally until late in the course where students have the logical

background to understand how it works and need the details of it in order to master the resolution calculus for first-order logic [3].

On the other hand, we never cut corners about logic itself. With the proof assistant Isabelle/HOL we can create canonical reference documents for logics and their metatheory. The formal language of Isabelle/HOL, namely higher-order logic, is precise and unambiguous. This means every proof can be mechanically checked, and that it is impossible to cheat and omit any details.

We summarize our main points:

1. We use Isabelle in both courses, including the editor Isabelle/jEdit and the Isabelle/ML facilities.
2. By exploring formally verified proof systems and provers, we use formal methods on the field of formal methods itself.
3. In the advanced course we in addition use Isabelle/Pure, showing the generic Isabelle logical framework and forcing students to manage without the automation of Isabelle/HOL.
4. We rely on group exercise sessions with competent teaching assistants and peer assistance in combination with the Isabelle proof assistant and our own tools.
5. We have individual assignments, as often and as early as possible, with a quick feedback loop from the teaching assistants.

In the next section, we discuss related work. In Sect. 3 we detail the position of our courses within the context of the rest of our computer science and software engineering program. Next we describe the BSc course in Sect. 4, followed by a description of the MSc course in Sect. 5. Finally we describe our overall experiences and ideas for future work in Sect. 6 and conclude in Sect. 7.

2 Related Work

Our two courses are based on a number of tools for teaching logic developed in recent years [10–15, 21, 36–40]. In the present paper we elaborate, for the first time, on the courses and detail our experiences.

We are not aware of any textbooks for teaching logic using the Isabelle proof assistant, but textbooks on formalizing a number of other computer science topics exist, like the book on programming language semantics [24, 29] or functional algorithms [26–28]. These books show that the proof assistant Isabelle/HOL can be used for teaching semantics, algorithms and data structures. There are also impressive books for the proof assistant Coq [33] and the proof assistant Lean [1] but we are not aware of approaches to teaching logic and automated reasoning where the proof systems and provers are formalized in a proof assistant. We envision a textbook around our tools, but are currently relying on a number of unpublished smaller notes and tutorials to teach students how to use them.

Bella [2] presents a teaching methodology for the so-called Inductive Method to verified security protocols and notes the following step:

But the first and foremost step is to convince the learners that they already somewhat used formal methods, although for other applications, for example in the domains of Physics and Mathematics. The argument will convey as few technicalities as possible, in an attempt to promote the general message that formal methods are not extraterrestrial even for students who are not theorists.

We attempt to promote a similar message to the students following our courses.

Harrison [19], Blanchette [5] and Reis [34] discuss aspects of formalizing the metatheory of proof systems and provers. In contrast to our work they do not consider the use of such formalizations as central components and tools in logic and automated reasoning courses.

3 Curricular Overview

The first of our courses is the BSc course meant for final-year students, while our second course is the MSc course. We would like to briefly explain the positioning of our courses within the overall computer science and engineering curriculum at the Technical University of Denmark (DTU). The curriculum at DTU is organized in a half-year semester structure, but after the first year students are free to organize their own study plan and have many electives which can be used for any course offered at the institution.

While the MSc course is intended to be followed after our BSc course, our students have very varied backgrounds because many MSc students have BSc degrees from other institutions. The backgrounds of the students following the BSc course are also varied because the course is followed by many exchange students, BEng students, and General Engineering students. Figure 2 shows the context of our courses in the overall computer science and software engineering program.

Course numbers and ECTS points are as follows for the BSc courses:

- 01017 Discrete Mathematics (5 ECTS)
- 02101 Introductory Programming (5 ECTS)
- 02105 Algorithms and Data Structures 1 (5 ECTS)
- 02110 Algorithms and Data Structures 2 (5 ECTS)
- 02141 Computer Science Modelling (10 ECTS)
- 02156 Logical Systems and Logic Programming (5 ECTS)
- 02157 Functional Programming (5 ECTS)
- 02180 Introduction to Artificial Intelligence (5 ECTS)
- 02450 Introduction to Machine Learning and Data Mining (5 ECTS)

We have here omitted the traditional BSc courses in Computer Engineering and Software Engineering as they play only a minor role in this context.

The MSc courses are organized in study lines which are optional to follow, but nevertheless guide the study planning for the students. Except for a single Innovation in Engineering course we do not have any mandatory MSc courses, though

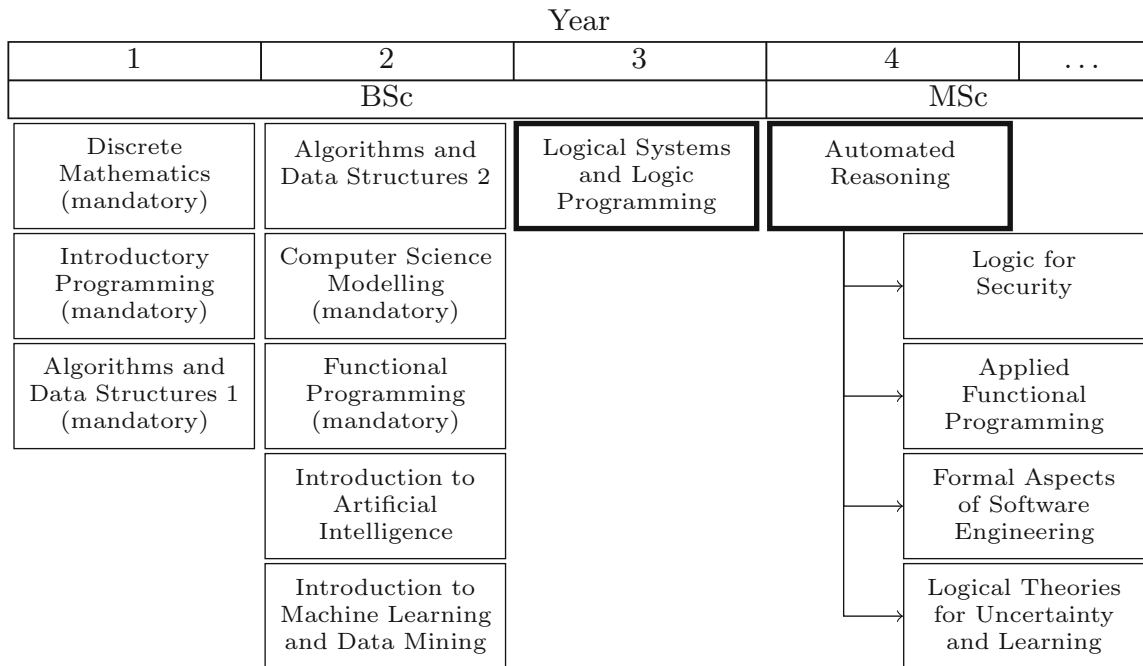


Fig. 2. Suggested course progression surrounding our courses.

students must of course primarily take courses related to computer science. The selected MSc courses to be taken after our courses on logic and automated reasoning are on the following study lines (we also have study lines in Computer Security and Digital Systems, but the former is more practically oriented compared to Safe and Secure by Design and the latter is more Electrical Engineering oriented):

- Study Line: Artificial Intelligence and Algorithms
02256 Automated Reasoning (5 ECTS)
02287 Logical Theories for Uncertainty and Learning (5 ECTS)
- Study Line: Embedded and Distributed Systems
02257 Applied Functional Programming (5 ECTS)
- Study Line: Safe and Secure by Design
02244 Logic for Security (7.5 ECTS)
- Study Line: Software Engineering
02263 Formal Aspects of Software Engineering (5 ECTS)

For the BSc course, we recommend that students have previous programming experience as well as knowledge of discrete mathematics and at least basic knowledge of algorithms and data structures. Functional programming is an advantage due to our use of systems implemented in Isabelle/HOL. These prerequisites are obtained in mandatory first and second year courses by most of the students following our BSc course.

However, a significant number of the students following our BSc course are either exchange students, come from the General Engineering program at DTU, or are BEng students. For exchange students, the structure of the curriculum of

their home institution may diverge from ours, which means that they sometimes have quite different backgrounds. Students from the General Engineering program have an interdisciplinary study plan, which means that they may not have all of the recommended prerequisites. Finally, BEng students have a curriculum which differs significantly from that of the BSc students, and are generally more focused on practical applications.

For the MSc course, we recommend that students have followed our BSc course and have experience with functional programming and basic algorithms in artificial intelligence. Students coming from the BSc program at DTU will mostly have these prerequisites, but a large amount of students on our MSc programmes come from other institutions. This means that we generally need to assume that students will not have all of the recommended prerequisites, and especially that they have not followed our BSc course.

Our courses provide skills that are useful in a number of MSc courses at DTU. A firm grasp of logic is of course useful for courses such as Logic for Security and Logical Theories for Uncertainty and Learning. Familiarity with formal methods and logic is useful for a course on Formal Aspects of Software Engineering. Several topics covered in our courses can provide interesting project ideas to implement for a course on Applied Functional Programming or for a BSc or MSc thesis. At DTU, it is also quite common to organize special elective courses based on student interest in a specific topic, and we have done so based on advanced topics related to our courses several times.

Both of our courses consist of a mix of lectures, live demonstrations of programs and proofs in Isabelle, and exercise sessions. During exercise sessions, students are free to discuss the problems within groups, and teaching assistants are available to provide help and formative evaluation during the sessions. Since many exercise sessions concern systems implemented in Isabelle, students can get immediate feedback on their proofs, and may ask teaching assistants for more detailed feedback and help if this is not sufficient. To aid student independence, we have for some of our systems developed tools which can provide more detailed formative evaluation of student work than Isabelle. Solutions are provided after all exercise sessions so students can compare their own proofs with ours. This is in contrast to the assignments where only feedback is provided.

Both courses additionally have several individual assignments, which we grade and provide feedback on quickly. These assignments count for part of the overall grade of the courses, with the rest of the grade coming from the exam.

4 BSc Course: Logical Systems and Logic Programming

The first of our courses is the BSc course on Logical Systems and Logic Programming. The course is essentially split in two concurrently running parts. One part of the course covers logic programming in Prolog, while the other part concerns formal logic. The course is based primarily on the textbook *Mathematical Logic for Computer Science* by Ben-Ari [3], and we cover most of the book in the course.

The course learning objectives can be seen in Fig. 1a and the week-by-week plan of the course can be seen in Table 1.

Table 1. Course plan for the course on Logical Systems and Logic Programming.

Week	Main topics	Assignment
1	Tutorial on Logic Programming	
2	Introduction (Prolog Note)	
3	Propositional Logic: Formulas, Models, Tableaux	
4	Propositional Logic: Deductive Systems	X
5	Propositional Logic: Sequent Calculus Verifier—Isabelle	
6	Propositional Logic: Resolution	X
7	First-Order Logic: Formulas, Models, Tableaux	
8	First-Order Logic: Deductive Systems	X
9	First-Order Logic: Sequent Calculus Verifier—Isabelle	
10	First-Order Logic: Terms and Normal Forms	X
11	First-Order Logic: Resolution	
12	First-Order Logic: Logic Programming	
13	First-Order Logic: Undecidability and Model Theory	X

We start by introducing the basic features of Prolog through a number of examples and exercises. We continue to introduce more Prolog features throughout the course, and use Prolog to show how to implement many of the concepts in logical systems.

After the short introduction to Prolog, we begin covering propositional logic. Following Ben-Ari’s book, we cover formulas, semantics, models, and semantic tableaux. This also allows us to discuss the issues of soundness and completeness. Next, we cover deductive systems in the styles of Hilbert and Gentzen, and show how to prove completeness by relating systems to existing systems that are known to be complete.

Having done this, we take an excursion into formal methods by introducing the Isabelle proof assistant. We use our Sequent Calculus Verifier (SeCaV) [11, 12, 15], which is implemented in Isabelle/HOL, to teach students how to write and formally verify proofs. This allows students to experiment with their proofs while getting immediate feedback on their correctness. For this first introduction, we use a version of SeCaV which is restricted to propositional logic. Since SeCaV is implemented within Isabelle/HOL, this also exposes students to the basics of proofs in the Isar proof language of Isabelle.

To conclude the sessions on propositional logic we introduce resolution, including Prolog programs that implement each step of a proof by resolution. This allows students to experiment with resolution proofs while also exposing them to non-trivial Prolog programs.

Next, we go through essentially the same topics as before, but now for first-order logic. We again use Prolog programs to explain concepts such as Skolemization and include a Prolog program for resolution in first-order logic.

At this point, we again digress to explore the full version of our Sequent Calculus Verifier, which is a deductive system for first-order logic. The system allows us to explain concepts such as de Bruijn indices and substitution of bound variables with simple implementations. Additionally, we showcase the formal proofs of soundness and completeness for the system. This allows us to explain these proofs in much detail while exposing students to more advanced usage of Isabelle. The implementation of SeCaV in Isabelle/HOL is also a good example for students, since it includes fully elaborated and concrete implementations of e.g., syntax, semantics, and proof rules.

Having done this, we include a number of exercises on implementing logical concepts in Prolog, including the implementation of a SAT solver. We briefly introduce concepts such as higher-order programming and constraint programming in Prolog. We also “close the loop” by finally explaining the relation between logic programming in Prolog and first-order logic. At this point the students have been sufficiently exposed to both to understand this quite quickly.

The final lecture is spent discussing some simple results in model theory and the concept of undecidability.

Throughout the course, students must hand in assignments concerning the various topics of the course. The first assignment is a mix of pen-and-paper formal proofs and Prolog programming exercises, while later assignments also include formal proof exercises in the Sequent Calculus Verifier. These assignments contribute to the final grade of the course. The rest of the grade is determined by a written final exam, which also includes a mix of pen-and-paper formal proofs and Prolog programming exercises.

5 MSc Course: Automated Reasoning

The second of our courses is the MSc course on Automated Reasoning. The course is essentially split in two concurrently running parts. One part of the course covers proving and programming in Isabelle [22, 25], while the other part concerns formal logic [10–15, 21, 36–40]. The course learning objectives can be seen in Fig. 1b and the week-by-week plan of the course can be seen in Table 2.

We start by exploring our formally verified micro provers for propositional logic [37, 38], which allow us to explain how provers can be implemented in e.g., Haskell, Isabelle/ML and Standard ML and how to prove correctness in Isabelle.

The Natural Deduction Assistant (NaDeA) [39] is a browser application for classical first-order logic with constants and functions. The syntax, the semantics and the inductive definition of the natural deduction proof system along with the soundness and completeness proofs are verified in Isabelle/HOL. Finished NaDeA proofs are automatically translated into the corresponding Isabelle-embedded proofs.

We have developed teaching materials about Isabelle/Pure [41], showing the generic Isabelle logical framework in order to ensure that students understand

Table 2. Course plan for the course on Automated Reasoning.

Week	Main topics	Assignment
1–2	Prerequisites, micro provers, getting started with Isabelle	X
3–4	Natural Deduction Assistant (NaDeA)	X
5–6	Isabelle/Pure for Intuitionistic and Classical First-Order Logic	X
7–8	Isabelle/Pure for Intuitionistic and Classical Higher-Order Logic	X
9–10	Axiomatic Propositional, First-Order and Higher-Order Logic	X
11–12	Students' Proof Assistant (SPA)	
13	Reserve/buffer lecture	X

what is going on at a lower level when they use the automation of Isabelle/HOL, and the learning outcome is tested in assignments using Isabelle/Pure.

We briefly describe our route from axiomatic propositional logic [7] to first-order logic with equality in our Students' Proof Assistant (SPA) [36] running inside Isabelle/HOL with a formally verified LCF-style prover kernel [31] and declarative proofs [41, 42].

The students can experiment in Isabelle/HOL with our formalized soundness and completeness theorems for several axiomatic systems [10], including the following well-known axioms in addition to the rule modus ponens:

Wajsberg 1937	$p \Rightarrow (q \Rightarrow p)$ $(p \Rightarrow q) \Rightarrow ((q \Rightarrow r) \Rightarrow (p \Rightarrow r))$ $((p \Rightarrow q) \Rightarrow p) \Rightarrow p$ $\perp \Rightarrow p$
Wajsberg 1939	$p \Rightarrow (q \Rightarrow p)$ $(p \Rightarrow (q \Rightarrow r)) \Rightarrow ((p \Rightarrow q) \Rightarrow (p \Rightarrow r))$ $((p \Rightarrow \perp) \Rightarrow \perp) \Rightarrow p$
Łukasiewicz 1948	$((p \Rightarrow q) \Rightarrow r) \Rightarrow ((r \Rightarrow p) \Rightarrow (s \Rightarrow p))$ $\perp \Rightarrow p$

We extend the Wajsberg 1939 axiomatic system for propositional logic to first-order logic with equality [20]:

$\vdash q$ if $\vdash p \Rightarrow q$ and $\vdash p$	(modus ponens rule)
$\vdash \forall x.p$ if $\vdash p$	(generalization rule)
$\vdash p \Rightarrow (q \Rightarrow p)$	
$\vdash (p \Rightarrow (q \Rightarrow r)) \Rightarrow ((p \Rightarrow q) \Rightarrow (p \Rightarrow r))$	
$\vdash ((p \Rightarrow \perp) \Rightarrow \perp) \Rightarrow p$	
$\vdash (\forall x.p \Rightarrow q) \Rightarrow (\forall x.p) \Rightarrow (\forall x.q)$	
$\vdash p \Rightarrow (\forall x.p)$ provided $x \notin \text{FV}(p)$	

$$\begin{aligned}
&\vdash (\exists x.x = t) \text{ provided } x \notin \text{FVT}(t) \\
&\vdash t = t \\
&\vdash s_1 = t_1 \Rightarrow \cdots \Rightarrow (s_n = t_n \Rightarrow f(s_1, \dots, s_n) = f(t_1, \dots, t_n)) \\
&\vdash s_1 = t_1 \Rightarrow \cdots \Rightarrow (s_n = t_n \Rightarrow P(s_1, \dots, s_n) = P(t_1, \dots, t_n)) \\
&\vdash (p \Leftrightarrow q) \Rightarrow (p \Rightarrow q) \\
&\vdash (p \Leftrightarrow q) \Rightarrow (q \Rightarrow p) \\
&\vdash (p \Rightarrow q) \Rightarrow ((q \Rightarrow p) \Rightarrow (p \Leftrightarrow q)) \\
&\vdash \top \Leftrightarrow (\perp \Rightarrow \perp) \\
&\vdash \neg p \Leftrightarrow (p \Rightarrow \perp) \\
&\vdash (p \wedge q) \Leftrightarrow ((p \Rightarrow (q \Rightarrow \perp)) \Rightarrow \perp) \\
&\vdash (p \vee q) \Leftrightarrow \neg(\neg p \wedge \neg q) \\
&\vdash (\exists x.p) \Leftrightarrow \neg(\forall x.\neg p)
\end{aligned}$$

Here FV is the set of free variables in a formula and FVT is the set of free variables in a term. Note that the axiomatic system is substitutionless as it uses equality in a clever way to avoid the complications of substitution [20, 36].

Amongst Pelletier's problems [32] for automated reasoning is problem 34, which is also known as Andrews's Challenge. The proof is not obvious at first glance since it relies on the fact that bi-implication is both commutative and associative [36]:

$$\begin{aligned}
&((\exists x.\forall y.P(x) \Leftrightarrow P(y)) \Leftrightarrow ((\exists x.Q(x) \Leftrightarrow (\forall y.Q(y)))) \Leftrightarrow \\
&((\exists x.\forall y.Q(x) \Leftrightarrow Q(y)) \Leftrightarrow ((\exists x.P(x) \Leftrightarrow (\forall y.P(y))))))
\end{aligned}$$

Comparing the declarative proofs in Isabelle/HOL and SPA is a good exercise for the students.

In addition to our tools for teaching logic we cover the following online papers:

1. M. Ben-Ari (2020): A Short Introduction to Set Theory [4]
2. W. M. Farmer (2008): The Seven Virtues of Simple Type Theory [6]
3. T. C. Hales (2008): Formal Proof [17]
4. T. Nipkow (2021): Programming and Proving in Isabelle/HOL [25]
5. L. C. Paulson (2018): Computational Logic: Its Origins and Applications[31]

The paper by Farmer provides a concise definition of higher-order logic and the tutorial by Nipkow provides a substantial set of exercises which the students must solve.

6 Discussion and Future Work

As mentioned, our BSc course uses our Sequent Calculus Verifier (SeCaV), which is embedded in Isabelle/HOL, for several exercise sessions and assignments. While the system is designed to be quite simple to use and understand, we have experienced that some students have a hard time writing proofs in the system. Additionally, the embedding in Isabelle/HOL is not able to give very helpful

error messages if a proof is wrong. To alleviate these issues, we have recently developed an online tool called the SeCaV Unshortener [11], which allows users to write proofs in a simpler syntax, which is then automatically translated into the embedding in Isabelle. Additionally, the tool is able to warn users about mistakes in their proofs by explicitly telling users why e.g. a proof rule cannot be applied. Recent research indicates that this kind of feedback impacts learning in computer science significantly, and is sufficient to allow students to move forward in most cases [18].

We also use SeCaV in our MSc course but only as self-study concerning the course prerequisites and selected parts of the papers [11, 12, 15] in the first weeks of the course.

We would like to integrate even more algorithms and proofs into Isabelle. Work is currently ongoing on an Isabelle implementation and proof of correctness of a tool for converting formulas to conjunctive normal form.

Michaelis and Nipkow [23] formalized a number of proof systems for propositional logic in Isabelle/HOL: resolution, natural deduction, sequent calculus and an axiomatic system. We would like to extend this line of work to first-order logic and higher-order logic.

We find that one of the main issues in both our 2020 and 2021 course on automated reasoning and formally verified functional programming is the course prerequisites. Functional programming is a prerequisite but we do not require a specific language and it is not possible to exclude any students. This is a real problem and in general we need to use the first part of the course to teach some of the prerequisites. Another prerequisite is mathematical logic—syntax, semantics and proof systems—and we use the micro provers to teach logic, functional programming and the basics of a proof assistant, in particular Isabelle, in a way that is challenging to almost all students. It is not for beginners and some students will most likely quit the course in the first month. In 2021, after the first month, 37 students were active and almost everyone submitted the first assignment. We have no clear solution to the issues concerning the course prerequisites but for 2022 we plan to offer a series of online sessions for self-study in mathematical logic and functional programming.

7 Conclusion

We have presented our detailed experiences teaching two courses on formal methods. The first course is the bachelor course on logical systems and logic programming, which has a focus on classical first-order logic and automatic theorem proving. We have additionally described how we use Prolog and Isabelle to introduce students to logic and formal methods.

The second course is the master course on automated reasoning, which has a focus on classical higher-order logic and interactive theorem proving. The proof assistant Isabelle is used more heavily in this course, and we use Isabelle/Pure as well as Isabelle/HOL. We have also described our online tools to be used with Isabelle/HOL, in particular the Sequent Calculus Verifier (SeCaV) and

the Natural Deduction Assistant (NaDeA). In addition, we have described our innovative Students' Proof Assistant which is formally verified in Isabelle/HOL and integrated in Isabelle/jEdit using Isabelle/ML.

We have described how our courses fit into the overall computer science and engineering curriculum, and what issues and challenges we experience that students often face. We have suggested some future work on the courses by which we hope to improve student learning outcomes.

Our teaching philosophy is related to the IsaFoL (Isabelle Formalization of Logic) project [5] which aims at developing formalizations in Isabelle/HOL of logics, proof systems, and automatic/interactive provers. Notable work in the same line includes the soundness and completeness of epistemic [8] and hybrid [9] logic and an ordered resolution prover for first-order logic [35]. These formalizations can serve as starting point for a student project to formalize the soundness and completeness of various other proof systems and provers.

We would like to formalize even more topics within basic logic such that students can explore concrete and executable definitions of various topics such as Skolemization while also seeing formal proofs of their correctness. Our overall conclusion is that using formal methods, in particular the proof assistant Isabelle, as a central tool for teaching logic and formal methods is possible as we have demonstrated since our first use of the Natural Deduction Assistant (NaDeA) and the Sequent Calculus Verifier (SeCaV) in 2014 and 2019, respectively.

Acknowledgements. We thank Asta Halkjær From for comments on drafts. We thank the three anonymous reviewers whose comments and suggestions helped improve the paper.

References

1. Baanen, A., Bentkamp, A., Blanchette, J., Limperg, J., Hölzl, J.: The Hitchhiker's Guide to Logical Verification (2020). https://github.com/blanchette/logical_verification_2020
2. Bella, G.: You already used formal methods but did not know it. In: Dongol, B., Petre, L., Smith, G. (eds.) FMTea 2019. LNCS, vol. 11758, pp. 228–243. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-32441-4_15
3. Ben-Ari, M.: Mathematical Logic for Computer Science. Springer, London (2012)
4. Ben-Ari, M.: A Short Introduction to Set Theory (2020). <https://www.weizmann.ac.il/sci-tea/benari/sites/sci-tea.benari/files/uploads/books/set.pdf>
5. Blanchette, J.C.: Formalizing the metatheory of logical calculi and automatic provers in Isabelle/HOL (invited talk). In: Mahboubi, A., Myreen, M.O. (eds.) Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, 14–15 January 2019, pp. 1–13. ACM (2019)
6. Farmer, W.M.: The seven virtues of simple type theory. *J. Appl. Log.* **6**(3), 267–286 (2008). <https://doi.org/10.1016/j.jal.2007.11.001>
7. From, A.H.: Formalizing Henkin-style completeness of an axiomatic system for propositional logic. In: Proceedings of the Web Summer School in Logic, Language and Information (WeSSLI) and the European Summer School in Logic, Language

- and Information (ESSLLI) Virtual Student Session, pp. 1–12 (2020). Preliminary paper, accepted for Springer post-proceedings
8. From, A.H.: Epistemic logic: completeness of modal logics. Archive of Formal Proofs, October 2018. https://devel.isa-afp.org/entries/Epistemic_Logic.html, Formal proof development
 9. From, A.H.: Formalizing a Seligman-style tableau system for hybrid logic. Archive of Formal Proofs, December 2019. https://isa-afp.org/entries/Hybrid_Logic.html, Formal proof development
 10. From, A.H., Eschen, A.M., Villadsen, J.: Formalizing axiomatic systems for propositional logic in Isabelle/HOL. In: Kamareddine, F., Sacerdoti Coen, C. (eds.) CICM 2021. LNCS (LNAI), vol. 12833, pp. 32–46. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-81097-9_3
 11. From, A.H., Jacobsen, F.K., Villadsen, J.: SeCaV: a sequent calculus verifier in Isabelle/HOL. In: 16th International Workshop on Logical and Semantic Frameworks with Applications (LSFA 2021) – Presentation Only/Online Papers, pp. 1–16 (2021). https://mat.unb.br/lsfa2021/pages/lsfa2021_proceedings/LSFA_2021_paper_5.pdf
 12. From, A.H., Jensen, A.B., Schlichtkrull, A., Villadsen, J.: Teaching a formalized logical calculus. Electron. Proc. Theor. Comput. Sci. **313**, 73–92 (2020). <https://doi.org/10.4204/EPTCS.313.5>
 13. From, A.H., Lund, S.T., Villadsen, J.: A case study in computer-assisted meta-reasoning. In: González, S.R., Machado, J.M., González-Briones, A., Wikarek, J., Loukanova, R., Katranas, G., Casado-Vara, R. (eds.) DCAI 2021. LNNS, vol. 332, pp. 53–63. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-86887-1_5
 14. From, A.H., Villadsen, J.: Teaching automated reasoning and formally verified functional programming in Agda and Isabelle/HOL. In: 10th International Workshop on Trends in Functional Programming in Education (TFPIE 2021) – Presentation Only/Online Papers, pp. 1–20 (2021). <https://wiki.tfpie.science.ru.nl/TFPIE2021>
 15. From, A.H., Villadsen, J., Blackburn, P.: Isabelle/HOL as a meta-language for teaching logic. Electron. Proc. Theor. Comput. Sci. **328**, 18–34 (2020). <https://doi.org/10.4204/eptcs.328.2>
 16. Grover, S.: Toward a framework for formative assessment of conceptual learning in K-12 computer science classrooms. In: Proceedings of the 52nd ACM Technical Symposium on Computer Science Education, SIGCSE 2021, pp. 31–37 (2021). <https://doi.org/10.1145/3408877.3432460>
 17. Hales, T.C.: Formal proof. Not. Am. Math. Soc. **55**, 1370–1380 (2008)
 18. Hao, Q., et al.: Towards understanding the effective design of automated formative feedback for programming assignments. Comput. Sci. Educ. 1–23 (2021). <https://doi.org/10.1080/08993408.2020.1860408>
 19. Harrison, J.: Formalizing basic first order model theory. In: Grundy, J., Newey, M. (eds.) TPHOLs 1998. LNCS, vol. 1479, pp. 153–170. Springer, Heidelberg (1998). <https://doi.org/10.1007/BFb0055135>
 20. Harrison, J.: Handbook of Practical Logic and Automated Reasoning. Cambridge University Press, Cambridge (2009)
 21. Jensen, A.B., Larsen, J.B., Schlichtkrull, A., Villadsen, J.: Programming and verifying a declarative first-order prover in Isabelle/HOL. AI Commun. **31**(3), 281–299 (2018)
 22. Krauss, A.: Defining Recursive Functions in Isabelle/HOL (2021). <https://isabelle.in.tum.de/doc/functions.pdf>

23. Michaelis, J., Nipkow, T.: Formalized proof systems for propositional logic. In: Abel, A., Forsberg, F.N., Kaposi, A. (eds.) 23rd International Conference on Types for Proofs and Programs, TYPES 2017, Budapest, Hungary, 29 May–1 June 2017. LIPIcs, vol. 104, pp. 5:1–5:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2017)
24. Nipkow, T.: Teaching semantics with a proof assistant: no more LSD trip proofs. In: Kuncak, V., Rybalchenko, A. (eds.) VMCAI 2012. LNCS, vol. 7148, pp. 24–38. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-27940-9_3
25. Nipkow, T.: Programming and Proving in Isabelle/HOL (Tutorial) (2021). <https://isabelle.in.tum.de/doc/prog-prove.pdf>
26. Nipkow, T.: Teaching algorithms and data structures with a proof assistant (invited talk). In: Hritcu, C., Popescu, A. (eds.) 10th ACM SIGPLAN International Conference on Certified Programs and Proofs, Virtual Event, CPP 2021, Denmark, 17–19 January 2021, pp. 1–3. ACM (2021). <https://doi.org/10.1145/3437992.3439910>
27. Nipkow, T., et al.: Functional Algorithms, Verified! (2021). <https://functional-algorithms-verified.org/>
28. Nipkow, T., Eberl, M., Haslbeck, M.P.L.: Verified textbook algorithms. In: Hung, D.V., Sokolsky, O. (eds.) ATVA 2020. LNCS, vol. 12302, pp. 25–53. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-59152-6_2
29. Nipkow, T., Klein, G.: Concrete Semantics - With Isabelle/HOL. Springer, Heidelberg (2014)
30. Nipkow, T., Wenzel, M., Paulson, L.C. (eds.): Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002). <https://doi.org/10.1007/3-540-45949-9>
31. Paulson, L.C.: Computational logic: its origins and applications. Proc. R. Soc. A. **474**(2210), 20170872 (2018). <https://doi.org/10.1098/rspa.2017.0872>
32. Peltier, N.: A variant of the superposition calculus. Archive of Formal Proofs, September 2016. <http://isa-afp.org/entries/SuperCalc.shtml>, Formal proof development
33. Pierce, B.C., et al.: Software Foundations – 6 Online Textbooks (2021). <https://softwarefoundations.cis.upenn.edu/>
34. Reis, G.: Facilitating meta-theory reasoning (invited paper). In: Pimentel, E., Tassi, E. (eds.) Proceedings Sixteenth Workshop on Logical Frameworks and Meta-Languages: Theory and Practice, Pittsburgh, USA, 16th July 2021. Electronic Proceedings in Theoretical Computer Science, vol. 337, pp. 1–12. Open Publishing Association (2021). <https://doi.org/10.4204/EPTCS.337.1>
35. Schlichtkrull, A., Blanchette, J., Traytel, D., Waldmann, U.: Formalizing Bachmair and Ganzinger’s ordered resolution prover. J. Autom. Reason. **64**(7), 1169–1195 (2020)
36. Schlichtkrull, A., Villadsen, J., From, A.H.: Students’ Proof Assistant (SPA). In: Quaresma, P., Neuper, W. (eds.) Proceedings 7th International Workshop on Theorem Proving Components for Educational Software, ThEdu@FLoC 2018, Oxford, United Kingdom, 18 July 2018. Electronic Proceedings in Theoretical Computer Science, vol. 290, pp. 1–13. Open Publishing Association (2018). <https://doi.org/10.4204/EPTCS.290.1>
37. Villadsen, J.: A micro prover for teaching automated reasoning. In: Seventh Workshop on Practical Aspects of Automated Reasoning (PAAR 2020) – Presentation Only/Online Papers, pp. 1–12 (2020). <https://www.eprover.org/EVENTS/PAAR-2020.html>

38. Villadsen, J.: Tautology checkers in Isabelle and Haskell. In: Calimeri, F., Perri, S., Zumpano, E. (eds.) Proceedings of the 35th Edition of the Italian Conference on Computational Logic (CILC 2020), Rende, Italy, 13–15 October 2020. CEUR Workshop Proceedings, vol. 2710, pp. 327–341. CEUR-WS.org (2020). <http://ceur-ws.org/Vol-2710/paper-21.pdf>
39. Villadsen, J., From, A.H., Schlichtkrull, A.: Natural Deduction Assistant (NaDeA). In: Quaresma, P., Neuper, W. (eds.) Proceedings 7th International Workshop on Theorem Proving Components for Educational Software, THedu@FLoC 2018, Oxford, United Kingdom, 18 July 2018. EPTCS, vol. 290, pp. 14–29 (2018). <https://doi.org/10.4204/EPTCS.290.2>
40. Villadsen, J., Schlichtkrull, A., From, A.H.: A verified simple prover for first-order logic. In: Konev, B., Urban, J., Rümmer, P. (eds.) Proceedings of the 6th Workshop on Practical Aspects of Automated Reasoning (PAAR 2018) co-located with Federated Logic Conference 2018 (FLoC 2018), Oxford, UK, 19 July 2018. CEUR Workshop Proceedings, vol. 2162, pp. 88–104. CEUR-WS.org (2018). <http://ceur-ws.org/Vol-2162/paper-08.pdf>
41. Wenzel, M.: The Isabelle/Isar Reference Manual (2021). <https://isabelle.in.tum.de/doc/isar-ref.pdf>
42. Wenzel, M.: Isar—a generic interpretative approach to readable formal proof documents. In: Bertot, Y., Dowek, G., Théry, L., Hirschowitz, A., Paulin, C. (eds.) TPHOLs 1999. LNCS, vol. 1690, pp. 167–183. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48256-3_12

Teaching Functional Programmers Logic and Metatheory

Frederik Krogsdal Jacobsen Jørgen Villadsen

DTU Compute - Department of Applied Mathematics and Computer Science - Technical University of Denmark

We present a novel approach for teaching logic and the metatheory of logic to students who have some experience with functional programming. We define concepts in logic as a series of functional programs in the language of the proof assistant Isabelle/HOL. This allows us to make notions which are often unclear in textbooks precise, to experiment with definitions by executing them, and to prove metatheoretical theorems in full detail. We have surveyed student perceptions of our teaching approach to determine its usefulness and found that students felt that our formalizations helped them understand concepts in logic, and that they experimented with them as a learning tool. However, the approach was not enough to make students feel confident in their abilities to design and implement their own formal systems. Further studies are needed to confirm and generalize the results of our survey, but our initial results seem promising.

1 Introduction

Logic is the foundation on which all of mathematics and computer science rests, and many undergraduate computer science programs therefore include an introductory course on logic. The logical systems introduced in such a course can be applied to databases, domain-specific languages, artificial intelligence, computer security, formal verification, and many other topics. At the Technical University of Denmark (DTU) we teach logic alongside logic programming in Prolog in a late-stage undergraduate course in addition to discrete mathematics taught in the first semester of the study program. Undergraduate courses like ours give students a basic understanding of logic and just enough knowledge to start applying basic logical systems in their work. But for students who really need to work with logic and design their own systems, this is not enough: they also need a good understanding of the metatheory of logic, i.e. why the systems work and how to prove that they do. Proofs about logical systems are fraught with possibilities for subtle mistakes of understanding, and it is our experience that many students never really “get” how the logical systems, and the proofs about them, work. Additionally, many textbooks define logical systems in terms of informal set theory and omit parts of their proofs (sometimes sweeping major complexities such as binders and substitution under the rug), which leads to difficulties in actually applying the theory when implementing real systems. For many students, implementing logical systems in their projects (and making sure that they are correct) can thus seem like an insurmountable challenge.

We have recently begun giving a graduate course (course number 02256) on automated reasoning with course material implemented in the proof assistant Isabelle/HOL. This year the course started with 80 students. The official course listing is available at <https://kurser.dtu.dk/course/02256>.

Isabelle/HOL is the higher-order logic version of the generic proof assistant Isabelle. Briefly, we can consider higher-order logic as the sum of functional programming and logic. Isabelle/HOL allows us to write definitions as functional programs and to formally prove properties of these programs. The proof assistant continuously checks that our proofs are correct, thus making it impossible to neglect any complexities. Additionally, Isabelle/HOL allows us to automatically export appropriate definitions into “real” functional languages such as Haskell, OCaml, Scala and Standard ML, whereby they can be integrated into larger systems.

By defining the logical systems we teach within Isabelle/HOL, we are thus able to give precise and executable definitions of every aspect of the systems, and our proofs of metatheoretical properties such as soundness and completeness of systems relate directly to these precise definitions and are verified by the proof assistant. Our experience is that students appreciate this: if they are in doubt about what something means, they can look up a precise definition and even execute it on examples of their own choosing to gain further understanding, and this should also make it clear how to implement the concepts in practice. To determine whether students actually find our approach useful, we survey student perceptions of the components of our course and of their own abilities and behaviour. Our hypotheses are that:

1. Concrete implementations in a programming language aid understanding of concepts in logic.
2. Students experiment with definitions to gain understanding.
3. Our formalizations make it clear to students how to implement the concepts in practice.
4. Our course makes students able to design and implement their own logical systems.
5. Prior experience with functional programming is useful for our course.
6. Our course helps students gain proficiency in functional programming.

In summary, we contribute:

- functional implementations of several logical systems which can be used to teach topics such as sequent calculus, natural deduction, de Bruijn indices, and algorithms for automatic theorem proving.
- formally verified proofs of common properties such as soundness and completeness for the systems mentioned above.
- an evaluation of the usefulness of our approach based on surveying student perceptions.

In the next section, we survey related work. In Section 3 we explain our teaching approach using a small verified automated theorem prover as an example. In Section 4 we survey student perceptions of our teaching approach to evaluate its usefulness, before concluding in Section 5.

2 Related work

There are of course numerous textbooks on logic, and several which include implementations of many of their definitions. Examples include books by Harrison [12], Ben-Ari [3], and Doets and van Eijck [6], which all contain implementations in various programming languages. Unfortunately, these implementations are not connected directly with the definitions and proofs in the books, and it is thus sometimes unclear how the programs really relate to the definitions in the text. Additionally, the proofs in these books are not verified by a proof assistant, and are about the informal definitions, not the programs themselves. Language, Logic and Proof [2] is a textbook and accompanying software package containing a number of small proof assistants designed to aid in teaching logic. While this means that students can rest assured that their exercise proofs are formally verified, the proofs in the book itself are not, and it is again sometimes unclear exactly how the software packages relate to the notions introduced in the text. In our approach, every logical system and notion is formally defined, and every theorem is proven in Isabelle/HOL, which allows students to see exactly how to implement concepts and inspect every detail of the proofs.

Some textbooks on logic do have formally verified proofs of their theorems, but these are typically not introductions to logic, but instead introductions to proof assistants that just happen to introduce some

Weeks	Topics
1 – 2	Basic set theory, propositional logic, sequent calculus, automatic theorem proving
3 – 4	Syntax and semantics of first-order logic, natural deduction, the LCF approach
5 – 6	Isar, intuitionistic logic, foundational systems
7 – 8	Proof by contradiction, classical logic, higher-order logic, type theory
9 – 10	Proofs in sequent calculus
11 – 13	Metatheory, prover algorithms, program verification

Table 1: Week plan and topics of our course.

particular logic they are working in. One example is Coq’Art [4], which is an introduction to the Coq proof assistant and contains many formally verified proofs about logic and other topics. Unfortunately, such books, being written for a much more advanced audience, are not by themselves good introductions to logic, since they presuppose knowledge beyond the usual computer science program. In our approach, on the other hand, we teach an introduction to logic without presupposing much beyond basic knowledge of functional programming.

The approach of formalizing definitions as functional programs has successfully been used for a number of other topics in computer science. The Software Foundations series [22] covers some basic logic, programming languages, formal verification, functional algorithms and separation logic with definitions and proofs in the Coq proof assistant. Concrete Semantics [20] is a textbook on programming language semantics with definitions and proofs in Isabelle/HOL. Functional Algorithms, Verified! [19] is an introduction to functional data structures and algorithms with definitions and proofs in Isabelle/HOL. Verified Functional Programming in Agda [25] is an introduction to functional programming itself, as well as functional algorithms, using the Agda programming language, which supports proofs about the programs written in it.

We have previously written about the contents of our courses [29], about teaching with Isabelle/Pure and Isabelle/HOL [8,9,28], and about individual formalizations [10,24,26,27], but not about the specifics and benefits of teaching logic and metatheory using functional programming to aid understanding, and we have not previously surveyed student perceptions about our approach in a rigorous manner.

3 A glimpse of the teaching approach

As mentioned, our approach is to define the logical systems which are our objects of study as functional programs in Isabelle/HOL. Having done this, we can then define e.g. automatic theorem provers and other derived programs. Isabelle/HOL then allows us to prove results about the implementations directly, e.g. that our automatic theorem provers are sound and complete.

3.1 The structure of the course

Table 1 contains a general overview of the topics we went through in our course in spring 2022. Note that functional programming is integrated in more or less all topics. For a more detailed explanation of the contents of the 2020 and 2021 versions of our course and its place in the overall curriculum, see [29]. For a discussion of our approach to teaching formal methods with Isabelle/HOL, see [14].

The course consists of lectures, exercise sessions, and assignments. This year, the lectures were in-person, but some parts were recorded for later use by the students. The exercise sessions were supervised by teaching assistants who were only available in-person. The lectures primarily covered theoretical topics, but also included tutorials on Isabelle/HOL and related software. The exercise sessions consisted primarily of programming and proving exercises in Isabelle/HOL, with occasional use of other software. There were 6 assignments during the course, and students worked on them individually. The assignments consisted of larger exercises as well as exercises similar to those in the exam.

This year, our graduate course on automated reasoning started with 80 students, but only 43 were left at the time of the survey. This is not unexpected, since our university allows students to deregister for courses within the first month with no consequences. This means that many students spend the first few weeks auditing many courses, and then choose which courses to stay registered to. Almost all of the students registered at the time of the survey stayed on for the rest of the course and registered for the exam. If a student registers for the exam then the course becomes binding for the student and the student must pass the course in order to graduate. Three exam attempts are allowed.

The exam consisted of five problems each comprising two questions, and the students were given two hours to complete it. One of the problems consisted of writing simple functions and proving properties about them in Isabelle/HOL. The two questions were as follows:

- Using only the constructors `0` and `Suc` and no arithmetical operators, define a recursive function `triple :: nat ⇒ nat` and prove `triple n = 3 * n`.
- Using the usual operators `+` and `-`, define two recursive functions `add42 :: int list ⇒ int list` and `sub42 :: int list ⇒ int list` that adds 42 to each integer and subtracts 42 from each integer, respectively, and prove `sub42 (add42 xs) = xs ∧ add42 (sub42 xs) = xs`.

Both questions can be solved by defining the functions and instructing Isabelle to perform induction. While these questions are not very difficult to solve, being able to prove properties of basic functional programs provides the basis for formalizing concepts in logic. From these simple beginnings, we can build an understanding of more complicated functions and proofs such as the example we introduce in the next section. Along the way we of course need to introduce both the logical concepts and the language and proof methods of the Isabelle proof assistant itself.

For a concrete example of the types of exam problems used, in particular on logic and the use of Isabelle, see the extended abstract [15].

The course evaluations and the grade history can be found on the Information tab on the official course listing available at <https://kurser.dtu.dk/course/02256>.

3.2 Logical systems and provers as functional programs

As a simple example, Figure 1 contains a definition of classical propositional logic. We define a datatype of formulas (this is the so-called deep embedding approach), then introduce semantics as a function from truth value assignments (interpretations) and formulas to truth values (we also define a semantics function for sequents, *sc*). We can then define an automatic theorem prover (function *mp* for micro prover) for the system which works by breaking formulas up using a system similar to a sequent calculus. All of these definitions are simple functional programs, and students should thus be able to understand them quite quickly when the ideas behind them are explained.

The programs can be executed directly inside of Isabelle/HOL, but can also be exported to standard functional programming languages, which allows us to use the concepts in practice. The Haskell program in Figure 2 has been automatically generated by Isabelle/HOL from the definitions in Figure 1.

datatype *'a form*

= *Pro* *'a* (\cdot) | *Falsity* (\perp) | *Imp* *'a form* *'a form* (**infixr** \rightarrow) 0)

primrec *semantics* **where**

$\langle \text{semantics } i \ (\cdot n) = i \ n \rangle$ |
 $\langle \text{semantics } - \ \perp = \text{False} \rangle$ |
 $\langle \text{semantics } i \ (p \rightarrow q) = (\text{semantics } i \ p \longrightarrow \text{semantics } i \ q) \rangle$

abbreviation $\langle \text{sc } X \ Y \ i \equiv (\forall p \in \text{set } X. \text{semantics } i \ p) \longrightarrow (\exists q \in \text{set } Y. \text{semantics } i \ q) \rangle$

primrec *member* **where**

$\langle \text{member } - \ \square = \text{False} \rangle$ |
 $\langle \text{member } m \ (n \# A) = (m = n \vee \text{member } m \ A) \rangle$

lemma *member-iff* [*iff*]: $\langle \text{member } m \ A \longleftrightarrow m \in \text{set } A \rangle$

by (*induct* *A*) *simp-all*

primrec *common* **where**

$\langle \text{common } - \ \square = \text{False} \rangle$ |
 $\langle \text{common } A \ (m \# B) = (\text{member } m \ A \vee \text{common } A \ B) \rangle$

lemma *common-iff* [*iff*]: $\langle \text{common } A \ B \longleftrightarrow \text{set } A \cap \text{set } B \neq \{\} \rangle$

by (*induct* *B*) *simp-all*

function *mp* **where**

$\langle \text{mp } A \ B \ (\cdot n \# C) \ \square = \text{mp } (n \# A) \ B \ C \ \square \rangle$ |
 $\langle \text{mp } A \ B \ C \ (\cdot n \# D) = \text{mp } A \ (n \# B) \ C \ D \rangle$ |
 $\langle \text{mp } - \ (\perp \# -) \ \square = \text{True} \rangle$ |
 $\langle \text{mp } A \ B \ C \ (\perp \# D) = \text{mp } A \ B \ C \ D \rangle$ |
 $\langle \text{mp } A \ B \ ((p \rightarrow q) \# C) \ \square = (\text{mp } A \ B \ C \ [p] \wedge \text{mp } A \ B \ (q \# C) \ \square) \rangle$ |
 $\langle \text{mp } A \ B \ C \ ((p \rightarrow q) \# D) = \text{mp } A \ B \ (p \# C) \ (q \# D) \rangle$ |
 $\langle \text{mp } A \ B \ \square \ \square = \text{common } A \ B \rangle$

by *pat-completeness simp-all*

termination

by (*relation* $\langle \text{measure } (\lambda(-, -, C, D). \sum p \leftarrow C \ @ \ D. \text{size } p) \rangle$) *simp-all*

theorem *main*: $\langle (\forall i. \text{sc } (\text{map} \cdot A \ @ \ C) \ (\text{map} \cdot B \ @ \ D) \ i) \longleftrightarrow \text{mp } A \ B \ C \ D \rangle$

by (*induct rule*: *mp.induct*) (*simp-all*, *blast*, *meson*, *fast*)

definition $\langle \text{prover } p \equiv \text{mp } \square \ \square \ [p] \rangle$

corollary $\langle \text{prover } p \longleftrightarrow (\forall i. \text{semantics } i \ p) \rangle$

unfolding *prover-def* **by** (*simp flip*: *main*)

Figure 1: An example Isabelle/HOL development defining a propositional logic and an automatic theorem prover for it. We begin by defining formulas with propositions, falsity, and implication as a datatype, then define semantics of formulas as a function. We then define an automatic theorem prover (function *mp*) for the system and prove that it terminates and is sound and complete. Note how every definition is a simple functional program and that Isabelle/HOL allows us to prove properties of these programs directly. Our students study this example very early on in our graduate course.

```

{-# LANGUAGE EmptyDataDecls, RankNTypes, ScopedTypeVariables #-}

module Scratch(Form, prover) where {

import Prelude ((==), (/=), (<), (<=), (>=), (>), (+), (-), (*), (/), (**),
  (>>=), (>>), (<<=), (&&), (||), (^), (^ ^), (.), ($), ($!), (++), (!!), Eq,
  error, id, return, not, fst, snd, map, filter, concat, concatMap, reverse,
  zip, null, takeWhile, dropWhile, all, any, Integer, negate, abs, divMod,
  String, Bool(True, False), Maybe(Nothing, Just));
import qualified Prelude;

data Form a = Pro a | Falsity | Imp (Form a) (Form a);

member :: forall a. (Eq a) => a -> [a] -> Bool;
member uu [] = False;
member m (n : a) = m == n || member m a;

common :: forall a. (Eq a) => [a] -> [a] -> Bool;
common uu [] = False;
common a (m : b) = member m a || common a b;

mp :: forall a. (Eq a) => [a] -> [a] -> [Form a] -> [Form a] -> Bool;
mp a b (Pro n : c) [] = mp (n : a) b c [];
mp a b c (Pro n : d) = mp a (n : b) c d;
mp uu uv (Falsity : uw) [] = True;
mp a b c (Falsity : d) = mp a b c d;
mp a b (Imp p q : c) [] = mp a b c [p] && mp a b (q : c) [];
mp a b c (Imp p q : d) = mp a b (p : c) (q : d);
mp a b [] [] = common a b;

prover :: forall a. (Eq a) => Form a -> Bool;
prover p = mp [] [] [] [p];

}

```

Figure 2: Haskell code generated by Isabelle/HOL from the example in Figure 1. Note that it is essentially a direct translation of the relevant Isabelle/HOL definitions, and that the proofs are not included. While the code is not pretty, a module such as this one can easily be integrated with other (handwritten) code to create a full application. This provides an example of how to implement and use the logical systems we cover in our course in practice.

Note that only the functions themselves, and not the proofs about them, are present in the Haskell code. The idea behind this approach is that the difficult parts of a program can be formally verified using Isabelle/HOL, then exported to a “normal” programming language as modules ready for integration into the overall program. This is similar to the LCF approach [13, 21] to theorem proving and the “hexagonal architecture” pattern common in object-oriented and functional programming [5], both of which advocate for programs consisting of a core application which interacts with users and other programs through an outer layer. In this case, the core can be implemented and verified in Isabelle/HOL, then exported, while the communication layer can be implemented in the target programming language.

3.3 Metatheory as properties of programs

Once we have defined a logical system in Isabelle/HOL, we can begin proving results about it. In Figure 1, we first prove that the functions *member* and *common* are correct by simple structural induction. Note that the proofs very much resemble the classic “The proof is by induction” often found in textbooks, but that the proof has been verified by Isabelle/HOL. This allows the reader to rest easy knowing that there is no hidden complexity in the proof.

We then prove termination of our prover by noting that the sum of the sizes of the two last arguments decrease. Isabelle/HOL requires that all functions are total, and in this case the proof is complicated enough that we have to prove it manually, but simple enough that we can do so easily. Next, we prove that our automatic theorem prover returns true if and only if the provided sequent is valid (theorem *main*). This proof is by induction, and the automation in Isabelle/HOL is enough to handle all cases. Finally, we conclude that our prover returns true for a formula exactly when it is valid (true in all interpretations), which means that the prover is sound and complete.

In this fashion, we can prove metatheoretical results about several logical systems used in our course, and thus showcase the proof techniques needed. We are also able to explore and prove equivalences between different logical systems by proving that the programs implementing them return the same results.

4 Student perceptions of the teaching approach

To attempt to shed some light on whether our approach is actually useful, we have conducted a survey study to analyze student perceptions about our course. The survey consisted of a single self-administered questionnaire, which all students following the course were invited to fill in during the tenth week of the course.

4.1 Methods

The study was designed to address the following six hypotheses:

1. Concrete implementations in a programming language aid understanding of concepts in logic.
2. Students experiment with definitions to gain understanding.
3. Our formalizations make it clear to students how to implement the concepts in practice.
4. Our course makes students able to design and implement their own logical systems.
5. Prior experience with functional programming is useful for our course.
6. Our course helps students gain proficiency in functional programming.

To address these hypotheses, we developed 12 questions to measure student perception of the course and their own behaviour and abilities. The questions can be seen on the left in Figure 3 or in the appendix. The questions were developed following evidence-based best practices for questionnaire item question design as described in [16], but the consistency and reproducibility of answers to the questionnaire items have not been tested.

Questions were close-ended and answers were given in terms of perception on one of three Likert-type [17] scales (multiple answers not possible) based on established best practice response options [16]: **An importance scale** consisting of the levels: "Not important", "Slightly important", "Moderately important", "Quite important", and "Essential".

A frequency scale consisting of the levels: "Almost never", "Once in a while", "Sometimes", "Often", and "Almost always".

A confidence scale consisting of the levels: "Not at all confident", "Slightly confident", "Moderately confident", "Quite confident", and "Completely confident".

Since the questionnaire consisted almost exclusively of attitudinal questions, we did not include a "Don't know" option for any question since this incentivizes partial responses [11]. "Don't know" responses were thus not possible, and students were required to fill in the entire questionnaire to submit it, which means that partial responses were not possible.

Since the survey consisted of a single questionnaire which was administered only once, we have only one sample (but note that question 10 asks about previous perception, which we analyze as a separate sample). The survey population was all 43 students following the course during course week 10. To preserve student anonymity, the sampling frame does not include auxiliary information such as gender, age, grades, etc. Specifically, this means that we cannot connect student answers to the questionnaire with measures of learning outcomes. We expect that there is some self-selection bias, since some students officially follow the course but do not actually show up to class, and are thus unlikely to answer the questionnaire. Additionally, we expect that students who either really like or really dislike the course will be more likely to answer the questionnaire, which could also result in some self-selection bias.

The questionnaire was administered using an online surveying solution provided by our university. This let us ensure that each student was only able to fill in the questionnaire once, and that participating students were anonymous. Participation in the survey was optional and not compensated financially or otherwise. The students were asked to participate in several lectures and through email.

4.2 Data analysis

The questionnaire has been designed to answer the hypotheses listed above. In this section, we will describe how we intend to confirm or reject each of the hypotheses based on the answers to the questions. Since we did not allow partial responses, we do not need to handle missing data for individual questions.

All of our data analysis was performed using the R environment for statistical computing [23]. The data from the questionnaire was pre-processed using the `readr` package [30]. The analysis script is available at <https://github.com/fkj/tfpie-2022-statistics>.

4.2.1 Concrete implementations in a programming language aid understanding of concepts in logic

Questions 1–3 ask about the perceived importance of the three elements of the course. We would like to compare the relative importance of the three elements to determine whether the implementations play an important part in gaining understanding.

If we had a measure of student learning outcomes, it would be interesting to perform a relative importance analysis on the three factors, but due to the anonymity of the questionnaire, we are not able to couple it to e.g. grades. Instead, we simply qualitatively compare the results of the questionnaire.

4.2.2 Students experiment with definitions to gain understanding

Question 4 asks whether students think it is important to experiment with their own examples when learning about a new concept. Question 5 asks whether students actually evaluate concrete examples to gain understanding when they are using Isabelle. We can look at the frequencies and median answers to gain an understanding of student behaviour.

4.2.3 Our formalizations make it clear to students how to implement the concepts in practice

Question 6 asks directly about how the students perceive their abilities to do this. We can look at the frequencies and median answer to gain an understanding of student perceptions.

4.2.4 Our course makes students able to design and implement their own logical systems

The meaning of this hypothesis is a bit vague, so we divide it into multiple concrete questions. Question 7 asks about perception of ability to design a formal system to solve a practical problem. Question 8 asks about perception of ability to implement a formal system in any programming language (without proofs). Question 9 asks about perception of ability to prove a formal system correct. We can look at the frequencies and median answers to gain an understanding of student perceptions.

4.2.5 Prior experience with functional programming is useful for our course

Question 10 asks about perceived ability with functional programming before starting the course and we will measure the association with perceived ability during the exercises and assignments in the course, which we ask about in question 11. Since both categories are ranked from low to high, there is no possibility of multiple choices, and we are interested in question 10 as a predictor for question 11, we will use Somers' Delta statistic [1] to measure the association [18]. We can also look at the frequencies and median answers to gain an understanding of student perceptions.

4.2.6 Our course helps students gain proficiency in functional programming

Question 10 asks about perceived ability before starting the course, while question 12 asks about perceived ability towards the end of the course. We can test whether the probability of high perceived ability has increased during the course by comparing the two answers for each student. Since we have two groups with paired observations (i.e. the same students twice), we can perform a two-sample paired signed-rank test [18] (i.e. a paired Wilcoxon signed-rank test) to determine whether there is a significant difference in perceived functional programming ability before and after the course [18]. The effect size of the paired Wilcoxon signed-rank test can be measured by the matched-pairs rank biserial correlation coefficient [18].

4.3 Results

21 students completed the entire questionnaire (48.84% response rate). The margin of error is thus below 15.48% (95% CI, p set to 0.5 to obtain worst-case margin). A summary of the answers can be seen in Figure 3, and the full data set is available in the appendix. Based on these answers, we will present the result of the analyses mentioned for each hypothesis above.

4.3.1 Concrete implementations in a programming language aid understanding of concepts in logic

Comparing the answers to question 1–3, we see that the median student thinks that:

1. the implementations in Isabelle are “Quite important” for their understanding of the course topics
2. the reading material is “Moderately important” for their understanding of the course topics
3. the lectures are “Slightly important” for their understanding of the course topics

This indicates that students believe that the implementations in Isabelle are the most important component of the course when it comes to their understanding of the course topics. While we do not have access to an association between these beliefs and actual learning outcomes, this data suggests that the hypothesis is plausible.

4.3.2 Students experiment with definitions to gain understanding

Looking at the answers to question 4, we see that all students find experiments with their own examples at least slightly important, while most students find them at least quite important. The median student finds experiments with their own examples “Essential” when learning about a new concept.

Looking at the answers to question 5, we see that there seems to be two distinct groups: one group which “Almost never” evaluates concrete examples in Isabelle, and a group which does so quite often. The median student evaluates concrete examples in Isabelle “Quite often”, which confirms our hypothesis.

4.3.3 Our formalizations make it clear to students how to implement the concepts in practice

Looking at the answers to question 6, we see that most students are not at all, or only slightly confident, but that there are also a number of students who are quite, or completely confident. The median student, however, is only “Slightly confident” that they could implement the concepts in practice. This suggests to reject the hypothesis.

4.3.4 Our course makes students able to design and implement their own logical systems

Looking at the answers to question 7, we see that the general trend is that student do not feel that they have the ability to design formal systems to prove practical problems. The median student is only “Slightly confident” in their ability to do this. This suggests to reject the hypothesis.

Looking at the answers to question 8, we see that the general trend is that students do not feel that they have the ability to implement their own logical systems (in any programming language). The median student is only “Slightly confident” in their ability to do this. This suggests to reject the hypothesis.

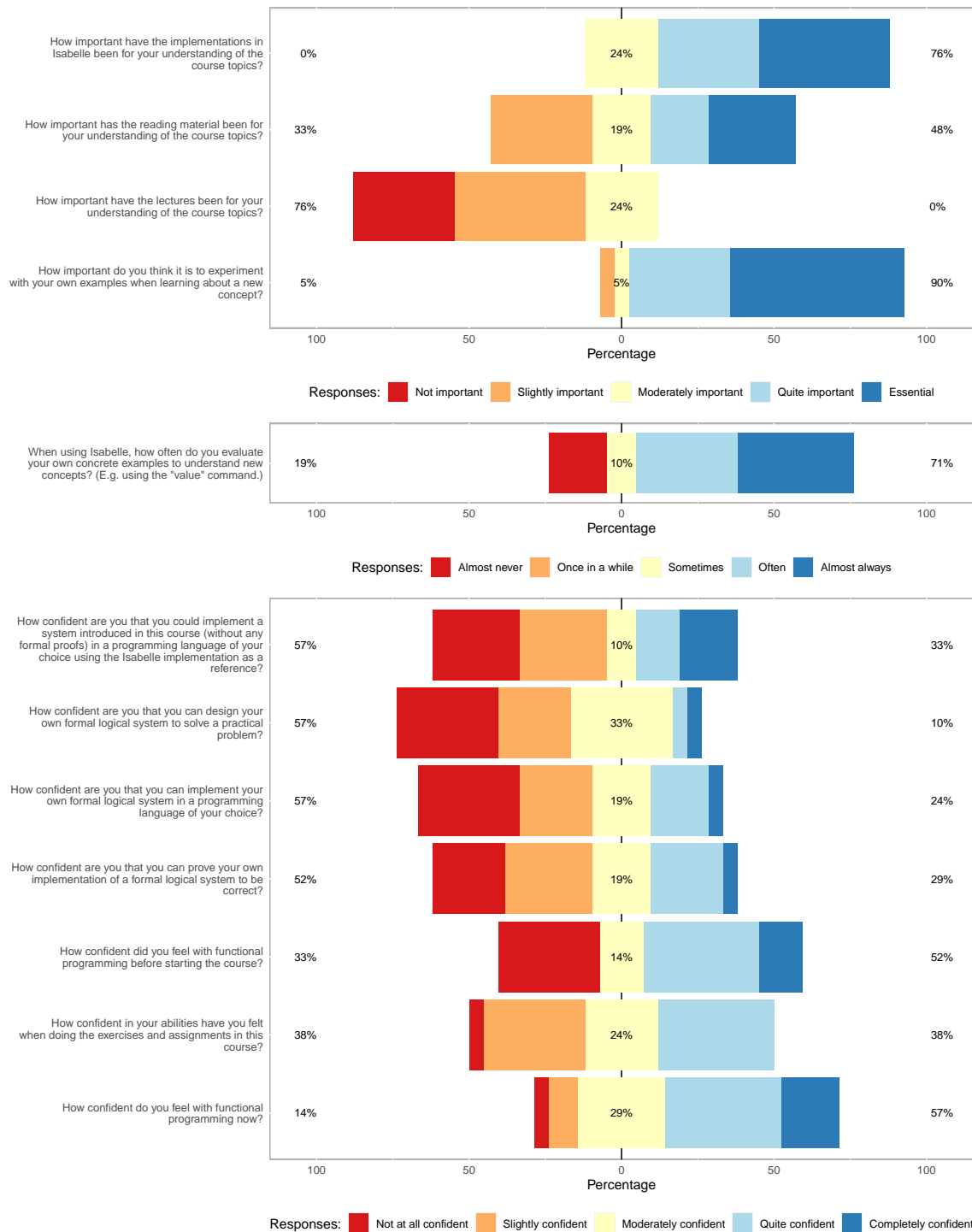


Figure 3: Summary of answers to the student self-evaluation questionnaire (n = 21). See the appendix for the full data set.

Looking at the answers to question 9, we see that the general trend is that students are only slightly to moderately confident that they have the ability to prove a formal system correct. The median student is only “Slightly confident” in their ability to do this. This suggests to reject the hypothesis.

Overall, the data suggests to reject the hypothesis.

4.3.5 Prior experience with functional programming is useful for our course

Looking at the answers to question 10, we see that there is an approximately normal distribution of students who felt moderately to completely confident as well as a second mode of students who were not at all confident with functional programming before starting the course. We know that this second mode consists of students who did not have any experience with functional programming prior to the course (although it was an explicit prerequisite). The median student felt “Quite confident” with functional programming before starting the course.

Looking at the answers to question 11, we see that nobody felt completely confident, while most students felt slightly to quite confident when doing the exercises and assignments in the course. The median student felt “Moderately confident” when doing the exercises and assignments.

Somers’ Delta statistic for the association of question 10 as a predictor for question 11 is 0.3642 (95% CI [0.0038, 0.7247]), which suggests a small to moderate association between students who felt confident with functional programming before starting the course and students who felt confident during the exercises and assignments [7]. This suggests to confirm the hypothesis.

4.3.6 Our course helps students gain proficiency in functional programming

Looking at the answers to question 10, we see that a number of students had no confidence with functional programming before starting the course (presumably due to them missing the functional programming prerequisite). Looking at the answers to question 12, we see that after the course, we obtain what looks more like a normal distribution centered around the “Quite confident” answer (although it is quite skewed).

The two-sample paired signed-rank test gives $V = 2$ (95% CI $[-3.5, 0.00]$), with $p = 0.0498$. Since $p \leq 0.05$, we conclude that there is a statistically significant difference in perceived functional programming ability before and after the course [18]. The effect size as measured by the matched-pairs rank biserial correlation is -0.857 (95% CI $[-1.00, -0.333]$), which suggests that the course has a large positive effect on perceived functional programming ability [18]. This suggests to confirm the hypothesis.

4.4 Discussion

We begin with a summary of our results:

Plausible: Concrete implementations in a programming language aid understanding of concepts in logic.

Confirmed: Students experiment with definitions to gain understanding.

Rejected: Our formalizations make it clear to students how to implement the concepts in practice.

Rejected: Our course makes students able to design and implement their own logical systems.

Confirmed: Prior experience with functional programming is useful for our course.

Confirmed: Our course helps students gain proficiency in functional programming.

Our first result suggests that our students feel that our teaching approach does help them understand concepts in logic. Unfortunately, our study does not have the data to confirm that they actually do, since we do not have access to assessments of learning outcomes linked to student responses. We also confirm that students do in fact use the formal definitions as learning tools by experimenting with examples and definitions within Isabelle.

On the other hand, students did not feel that our formalizations were enough to make it clear how to design and implement their own logical systems, or the concepts in the course in general. While we did not expect that students would feel very confident doing this (since formal verification is generally very difficult), we had still hoped for more confident answers. This suggests that we should attempt to find ways to improve our course in this respect. We also note that students do not seem to think that our lectures are very useful, which suggests that we should consider whether they are necessary or if even more time should be spent on exercises. By reducing the time spent on lectures, it may also be possible to obtain time for some project work, which may help students become more confident in designing and implementing their own logical systems.

We also found that students with more functional programming experience before starting the course felt more confident during the course, which is as expected. Our data also suggests that our course has a large positive effect on self-perceived functional programming confidence (though this may be a result of some students having no prior functional programming experience, see Section 4.4.1).

Our study is quite limited in scope, and the confidence intervals on our results are thus not very narrow. A larger study is needed to obtain precise knowledge of effect sizes. Since students participated in the survey of their own volition and with no compensation, we expect that our results also suffer from some self-selection bias. Further studies are needed to determine whether our results generalize.

4.4.1 Post hoc exploratory analyses

Having seen the answers to our survey, we notice some phenomena that could be interesting to explore further. Note that the analyses in this section are post hoc, i.e. they have been formulated after seeing the data, and that further studies are thus necessary to confirm any effects described in this section.

In Section 4.3.2, we noted that the answers to question 5 seem to contain two distinct groups. We can think of two possible reasons that a group of students almost never experiment with concrete examples in Isabelle: either there is a group of students who simply had not understood that this is possible, or some students do not find experimentation valuable enough to actually do it outside of the abstract. To investigate this further, we can look at the distribution of answers to whether students think that experimentation is important for students who answered that they almost never evaluate concrete examples in Isabelle and students who answered otherwise. We can split the data set in "Almost never" and "Other" and look at the distribution of how important students think experimentation is for each category. By doing this, we see that the shapes of the two distributions do not seem to differ significantly.

We can also measure the association between students who think that experimentation is important and students who often evaluate concrete examples in Isabelle. Since both categories are ranked from low to high, there is no possibility of multiple choices, and we are interested in question 4 as a predictor for question 5, we will use Somers' Delta statistic [1] to measure the association [18]. The statistic is -0.1039 (95% CI $[-0.4777, 0.2699]$), which shows a small negative association between the variables, i.e. that students who find experimentation more important evaluate concrete examples slightly less often than those who find it less important. This is surprising. If we assume that the answers of "almost never" are due to students not knowing about the feature, and thus remove them from the data set as

special cases, the association becomes -0.0814 (95% CI $[-0.5046, 0.3418]$), which again shows a small negative association. While there is not enough data to give good confidence intervals, it seems like there is no statistically significant association between the perceived importance of experimentation and the frequency of evaluating concrete examples in Isabelle. Note, however, that students still generally tend to believe that experimentation is important, and that they generally tend to evaluate their own concrete examples in Isabelle. One possible explanation is that some students who believe that experiments are essential are used to doing experiments with pen and paper, and thus do this instead of using Isabelle to evaluate examples, but we do not have any data to investigate this hypothesis in this study.

In Section 4.3.3, we noted that the answers to question 6 seem to contain two distinct groups. The bimodality is interesting, since we know that many students did not have experience with functional programming prior to our course (even though it was an explicitly stated prerequisite). We can test whether there is an association between perceived functional programming ability and perceived ability to implement the concepts in practice. We have data for perceived programming ability both before and after the course, so we can analyze both. Since all categories are ranked from low to high, there is no possibility of multiple choices, and we are interested in questions 10 and 12 as predictors for question 6, we will use Somers' Delta statistic [1] to measure the associations [18]. For previous functional programming ability, the statistic is 0.194 (95% CI $[-0.139, 0.527]$). This suggests that there is an association, but that it is barely significant [7]. For current functional programming ability, the statistic is 0.394 (95% CI $[0.0807, 0.707]$). This suggests a small to moderate, but statistically significant association [7].

We conclude that it seems that our formalizations may help students understand how to implement the concepts in practice, but only significantly so if they were confident functional programmers after following the course. Further studies are needed to confirm this effect, since the analysis above is post hoc.

In Section 4.3.4, we concluded that students do not generally feel confident in their abilities to implement their own logical systems. It could be interesting to measure the association between perceived functional programming ability and perceived ability to implement logical systems. We have data for perceived programming ability both before and after the course, so we can analyze both. Since both categories are ranked from low to high, there is no possibility of multiple choices, and we are interested in questions 10 and 12 as predictors for question 8, we will use Somers' Delta statistic [1] to measure the associations [18]. For previous functional programming ability, the statistic is 0.108 (95% CI $[-0.259, 0.474]$). This suggests that there is no statistically significant association [7]. For current functional programming ability, the statistic is 0.419 (95% CI $[0.0445, 0.794]$). This suggests a small to moderate association [7].

We conclude that it seems that our formalizations may help students understand how to implement logical systems, but only if they are confident functional programmers after the course, and only slightly so.

In Section 4.3.5, we noted that the answers to question 10 seem to have two modes. If we look at the two modes independently, the students with the functional programming prerequisite tend to feel moderately to quite confident while most students without the functional programming prerequisite felt only slightly confident. This also supports the original hypothesis.

We can try to remove the students with no functional programming experience and measure the association between perceived ability with functional programming before starting the course and perceived ability during the exercises and assignments in the course again to see whether the course requires advanced functional programming skills. We will again use Somers' Delta statistic to measure association [18]. The value is 0.0000 (95% CI $[-0.5019, 0.5019]$), which we interpret as meaning that only basic knowledge of functional programming seems to be needed in our course. This tracks well with our design choices in our systems, where we deliberately avoid "advanced" concepts such as folds in our implementations and exercises in an effort to obtain this result.

In Section 4.3.6, we found that the course has a significant positive effect on perceived functional programming ability. We can try to remove the students with no functional programming experience to see whether there is still a significant difference for those students who already have functional programming experience. We again perform a two-sample paired signed-rank test (i.e. a paired Wilcoxon signed-rank test) to determine whether there is a significant difference in perceived functional programming ability before and after the course [18]. The test shows $V = 1$, with $p = 1$. This means there is no statistically significant difference in perceived functional programming ability for the group of students who already have functional programming experience. In fact, by inspection of the data set we see that every student has answered exactly the same for both questions, except one student who moved from feeling "Quite confident" to feeling only "Moderately confident" after having taken the course. The lack of effect on perceived functional programming ability for students who already have some experience is to be expected since our course does not utilize any advanced concepts in functional programming.

4.5 Future work

Our results suggest that our approach seems to aid student understanding of logic, but that students do not feel that our formalizations are enough to make it clear how to implement the concepts in practice, and that students do not feel capable of designing and implementing their own logical systems after the course. This suggests that we should attempt to find ways to make it more clear how to do this in future iterations of the course. One possible way to do this would be to introduce a project about implementing some system to solve a practical problem, but this would almost certainly require us to extend the course load. It may also be possible to replace some assignments and exercises during the course with more practically oriented ones. Additionally, we may be able to reduce the time spent on lectures, since students do not seem to find them very important for their learning outcomes.

Our survey consisted almost entirely of attitudinal questions, which means that we are forced to believe student self-perceptions to some degree. We could improve this aspect by assessing e.g. understanding of logic and experience with functional programming by learning outcome measures such as previous grades or tests during the survey. Adding tests of aptitude to the survey would make completing it a much larger endeavour, however, which might decrease the number of students willing to participate and introduce additional self-selection bias. Including measures such as previous grades would require us to handle non-anonymous data, which makes the survey a much larger undertaking, since we would then need to collect informed consent letters, obtain institutional review board approval, etc. The same is true for demographic data such as gender, age, etc.

The generalizability of our survey may be doubted, since it may have issues of self-selection bias, and since our sample size was not large enough to obtain a narrow margin of error. Self-selection bias may be decreased by screening participants to ensure that our sample is representative of the population, but this will also require collection of demographic data and possibly previous grades, and will most likely

reduce the sample size. Another way to reduce self-selection bias is to collect data on every student, but this has obvious ethical issues. In the future, our survey could be repeated on other cohorts to increase the generalizability of our results.

Since our course is new, we have no data from previous pen-and-paper based courses at our university to compare our approach to. Performing such a comparison in a rigorous way would also be difficult, since it is not obvious how to choose a test of student learning outcomes that is comparable for both pen-and-paper approaches and our approach.

Finally, it may be interesting to conduct further studies to determine whether the effects discussed in Section 4.4.1 actually exist. We have noted the following interesting questions which arise from phenomena in our sample, but which, being the result of post hoc analyses, cannot be confirmed from the answers to the present survey:

- Why do students who think experimentation is important do it less? Do they do it on paper instead?
- Does functional programming experience play a significant role in understanding of how to implement concepts in practice?
- Does functional programming experience play a significant role in understanding of how to design and implement one's own logical systems?
- Does our course have a positive effect on functional programming skill for students who are already confident functional programmers?

5 Conclusion

We have presented our approach for teaching logic and metatheory using functional programming and demonstrated how logical concepts can be defined as simple functional programs about which we can prove metatheoretical results in a natural way. We have surveyed our students to determine their perceptions of our teaching approach. Our survey shows that students feel that our formalizations do aid them in understanding concepts in logic. We also confirm that students do in fact use the formal definitions as learning tools by experimenting with examples and definitions within Isabelle. On the other hand, students did not feel that our formalizations were enough to make it clear how to design and implement their own logical systems, or the concepts in the course in general. This suggests that we should attempt to find ways to make this clearer in future iterations of the course. Our survey also revealed a number of interesting phenomena which may be interesting to pursue further in future studies. Future studies are also needed to improve the generalizability of our results. We believe that our study design can serve as a useful methodology for such further studies.

For now, our course material consists of a series of lecture notes and corresponding formalizations and student exercise templates in Isabelle/HOL, and we combine this with excerpts from textbooks and tutorials. It would be great to have a comprehensive and coherent textbook written with this style of teaching in mind, but our course material has not yet stabilized enough for us to consider writing one. We would like to make clear that a book containing nothing but programs is not what we advocate; instead, we would like a textbook containing explanations based on, and corresponding to, the precise definitions in the functional programs, such that readers can always clarify any doubts about the “intuitive” explanations given in the text by referring to the programs implementing the definitions. Similarly, we do not believe that textbooks should contain meticulous proofs going into every detail of every case, but that such proofs should be available to the reader if they are not convinced by the abbreviated arguments given in the text.

Acknowledgements

We thank Agnes Moesgård Eschen, Asta Halkjær From, Alceste Scalas, Michael Reichhardt Hansen and Simon Tobias Lund for comments on drafts. We also thank the anonymous reviewers for their useful suggestions.

References

- [1] Signorell Andri et mult. al. (2021): *DescTools: Tools for Descriptive Statistics*. Available at <https://cran.r-project.org/package=DescTools>. R package version 0.99.44.
- [2] David Barker-Plummer, Jon Barwise & John Etchemendy (2011): *Language, Proof and Logic*, second edition. Center for the Study of Language and Information.
- [3] Mordechai Ben-Ari (2012): *Mathematical Logic for Computer Science*. Springer, doi:10.1007/978-1-4471-4129-7.
- [4] Yves Bertot & Pierre Castéran (2004): *Interactive Theorem Proving and Program Development*. Springer, Berlin, Heidelberg, doi:10.1007/978-3-662-07964-5.
- [5] Alistair Cockburn (2005): *Hexagonal architecture*. Available at <https://alistair.cockburn.us/hexagonal-architecture/>. Blog article.
- [6] Kees Doets & Jan van Eick (2012): *The Haskell Road to Logic, Maths and Programming*, second edition. College Publications.
- [7] Christopher J. Ferguson (2009): *An Effect Size Primer: A Guide for Clinicians and Researchers*. *Professional Psychology: Research and Practice* 40, pp. 532–538, doi:10.1037/a0015808.
- [8] Asta Halkjær From, Alexander Birch Jensen, Anders Schlichtkrull & Jørgen Villadsen (2019): *Teaching a Formalized Logical Calculus*. In Pedro Quaresma, Walther Neuper & João Marcos, editors: *Proceedings 8th International Workshop on Theorem Proving Components for Educational Software, ThEdu@CADE 2019, Natal, Brazil, 25th August 2019, EPTCS* 313, pp. 73–92, doi:10.4204/EPTCS.313.5.
- [9] Asta Halkjær From, Jørgen Villadsen & Patrick Blackburn (2020): *Isabelle/HOL as a Meta-Language for Teaching Logic*. In Pedro Quaresma, Walther Neuper & João Marcos, editors: *Proceedings 9th International Workshop on Theorem Proving Components for Educational Software, ThEdu@IJCAR 2020, Paris, France, 29th June 2020, EPTCS* 328, pp. 18–34, doi:10.4204/EPTCS.328.2.
- [10] Asta Halkjær From, Anders Schlichtkrull & Jørgen Villadsen (2021): *A Sequent Calculus for First-Order Logic Formalized in Isabelle/HOL*. In Stefania Monica & Federico Bergenti, editors: *Proceedings of the 36th Italian Conference on Computational Logic - CILC 2021, Parma, Italy, September 7-9, 2021, CEUR Workshop Proceedings* 3002, CEUR-WS.org, pp. 107–121. Available at <http://ceur-ws.org/Vol-3002/paper7.pdf>.
- [11] Jason Harlacher (2016): *An educator's guide to questionnaire development (REL 2016-108)*. Technical Report, Washington, DC: U.S. Department of Education, Institute of Education Sciences, National Center for Education Evaluation and Regional Assistance, Regional Educational Laboratory Central. Available at <https://ies.ed.gov/ncee/edlabs/regions/central/resources/pemtoolkit/pdf/module-6/CE5.3.2-An-Educators-Guide-to-Questionnaire-Development.pdf>.
- [12] John Harrison (2009): *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, doi:10.1017/CBO9780511576430.
- [13] John Harrison, Josef Urban & Freek Wiedijk (2014): *History of Interactive Theorem Proving*. In Jörg H. Siekmann, editor: *Computational Logic, Handbook of the History of Logic* 9, North-Holland, pp. 135–214, doi:10.1016/B978-0-444-51624-4.50004-6. Available at <https://www.sciencedirect.com/science/article/pii/B9780444516244500046>.

- [14] Frederik Krogsdal Jacobsen & Jørgen Villadsen (2022): *Lessons of Teaching Formal Methods with Isabelle*. Isabelle Workshop. Available at https://files.sketis.net/Isabelle_Workshop_2022/Isabelle_2022_paper_9.pdf.
- [15] Frederik Krogsdal Jacobsen & Jørgen Villadsen (2022): *On Exams with the Isabelle Proof Assistant*. 11th International Workshop on Theorem proving components for Educational software. Available at <https://www.uc.pt/en/congressos/thedu/ThEdu22>. Extended Abstract.
- [16] Anthony R. Artino Jr., Jeffrey S. La Rochelle, Kent J. Dezee & Hunter Gehlbach (2014): *Developing questionnaires for educational research: AMEE Guide No. 87*. *Medical Teacher* 36(6), pp. 463–474, doi:10.3109/0142159X.2014.889814.
- [17] Rensis Likert (1932): *A Technique for the Measurement of Attitudes*. *Archives of Psychology* 140, pp. 1–55.
- [18] Salvatore S. Mangiafico (2016): *Summary and Analysis of Extension Program Evaluation in R*, 1.19.10 edition. Rutgers Cooperative Extension. Available at <https://rcompanion.org/handbook>.
- [19] Tobias Nipkow, Jasmin Blanchette, Manuel Eberl, Alejandro Gómez-Londoño, Peter Lammich, Christian Sternagel, Simon Wimmer & Bohua Zhan (2021): *Functional Algorithms, Verified!* Available at <https://functional-algorithms-verified.org/>.
- [20] Tobias Nipkow & Gerwin Klein (2014): *Concrete Semantics*. Springer, Cham, doi:10.1007/978-3-319-10542-0.
- [21] Lawrence C. Paulson, Tobias Nipkow & Makarius Wenzel (2019): *From LCF to Isabelle/HOL*. *Formal Aspects of Computing* 31(6), pp. 675–698, doi:10.4204/EPTCS.118.4.
- [22] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hrițcu, Vilhelm Sjöberg & Brent Yorgey (2017): *Software Foundations*. Electronic textbook. <http://www.cis.upenn.edu/~bcpierce/sf>.
- [23] R Core Team (2022): *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. Available at <https://www.R-project.org/>.
- [24] Anders Schlichtkrull, Jørgen Villadsen & Andreas Halkjær From (2018): *Students' Proof Assistant (SPA)*. In Pedro Quaresma & Walther Neuper, editors: *Proceedings 7th International Workshop on Theorem proving components for Educational software, ThEdu@FLoC 2018, Oxford, United Kingdom, 18 July 2018, EPTCS 290*, pp. 1–13, doi:10.4204/EPTCS.290.1.
- [25] Aaron Stump (2016): *Verified Functional Programming in Agda*. Association for Computing Machinery and Morgan & Claypool, doi:10.1145/2841316.
- [26] Jørgen Villadsen (2020): *Tautology Checkers in Isabelle and Haskell*. In Francesco Calimeri, Simona Perri & Ester Zupanò, editors: *Proceedings of the 35th Italian Conference on Computational Logic - CILC 2020, Rende, Italy, October 13-15, 2020, CEUR Workshop Proceedings 2710*, CEUR-WS.org, pp. 327–341. Available at <http://ceur-ws.org/Vol-2710/paper21.pdf>.
- [27] Jørgen Villadsen, Andreas Halkjær From & Anders Schlichtkrull (2018): *Natural Deduction Assistant (NaDeA)*. In Pedro Quaresma & Walther Neuper, editors: *Proceedings 7th International Workshop on Theorem proving components for Educational software, ThEdu@FLoC 2018, Oxford, United Kingdom, 18 July 2018, EPTCS 290*, pp. 14–29, doi:10.4204/EPTCS.290.2.
- [28] Jørgen Villadsen, Asta Halkjær From & Patrick Blackburn (2022): *Teaching Intuitionistic and Classical Propositional Logic Using Isabelle*. In João Marcos, Walther Neuper & Pedro Quaresma, editors: *Proceedings 10th International Workshop on Theorem Proving Components for Educational Software, (Remote) Carnegie Mellon University, Pittsburgh, PA, United States, 11 July 2021, Electronic Proceedings in Theoretical Computer Science 354*, Open Publishing Association, pp. 71–85, doi:10.4204/EPTCS.354.6.
- [29] Jørgen Villadsen & Frederik Krogsdal Jacobsen (2021): *Using Isabelle in Two Courses on Logic and Automated Reasoning*. In João F. Ferreira, Alexandra Mendes & Claudio Menghi, editors: *Formal Methods Teaching*, Springer International Publishing, Cham, pp. 117–132, doi:10.1007/978-3-030-91550-6_9.
- [30] Hadley Wickham, Jim Hester & Jennifer Bryan (2022): *readr: Read Rectangular Text Data*. <https://readr.tidyverse.org>, <https://github.com/tidyverse/readr>.

Appendix: Survey data

To save space, the survey data has been compressed using numeral encodings. The questions are numerically encoded according to the following table, which also shows the Likert-type scale used for their response options (see subsection 4.1 for details).

#	Question	Scale
1	How important have the implementations in Isabelle been for your understanding of the course topics?	Importance
2	How important has the reading material been for your understanding of the course topics?	Importance
3	How important have the lectures been for your understanding of the course topics?	Importance
4	How important do you think it is to experiment with your own examples when learning about a new concept?	Importance
5	When using Isabelle, how often do you evaluate your own concrete examples to understand new concepts? (E.g. using the "value" command.)	Frequency
6	How confident are you that you could implement a system introduced in this course (without any formal proofs) in a programming language of your choice using the Isabelle implementation as a reference?	Confidence
7	How confident are you that you can design your own formal logical system to solve a practical problem?	Confidence
8	How confident are you that you can implement your own formal logical system in a programming language of your choice?	Confidence
9	How confident are you that you can prove your own implementation of a formal logical system to be correct?	Confidence
10	How confident did you feel with functional programming before starting the course?	Confidence
11	How confident in your abilities have you felt when doing the exercises and assignments in this course?	Confidence
12	How confident do you feel with functional programming now?	Confidence

The responses to our questionnaire are numerically encoded in the following table according to the position on the Likert scale of the question, with 1 being the lowest possible level and 5 being the highest possible level.

Question #	Answers (each column contains the answers from a single student)																				
1	5	4	4	5	3	5	3	4	5	5	4	4	3	4	4	5	5	5	3	5	3
2	5	5	3	3	4	5	4	3	5	2	4	2	5	2	2	2	3	2	5	2	4
3	3	2	2	3	2	2	2	3	2	2	1	1	3	1	1	2	1	1	1	3	2
4	5	2	4	4	5	5	4	5	4	5	5	5	4	4	4	3	5	5	5	5	5
5	5	5	4	5	4	4	4	1	4	5	3	1	5	3	5	1	4	1	5	5	4
6	5	2	1	1	5	3	1	5	4	1	2	2	2	2	4	1	1	2	3	4	5
7	5	1	1	1	3	3	1	3	4	2	2	2	3	2	1	3	1	1	3	3	2
8	5	1	1	1	3	3	1	4	4	2	2	1	2	2	1	4	1	2	3	4	3
9	5	1	1	1	2	4	2	4	3	2	3	2	3	4	4	3	1	1	4	2	2
10	1	3	4	3	5	4	1	4	5	1	4	5	4	4	3	4	1	1	1	1	4
11	3	4	4	3	3	4	1	4	4	2	4	2	3	3	2	2	2	4	2	2	4
12	5	3	4	3	5	4	2	4	5	3	3	5	4	4	3	4	1	3	4	2	4

On Exams with the Isabelle Proof Assistant

Frederik Krogsdal Jacobsen Jørgen Villadsen

Technical University of Denmark, Kongens Lyngby, Denmark

fkjac@dtu.dk

jovi@dtu.dk

We present an approach for testing student learning outcomes in a course on automated reasoning using the Isabelle proof assistant. The approach allows us to test both general understanding of formal proofs in various logical proof systems and understanding of proofs in the higher-order logic of Isabelle/HOL in particular. The use of Isabelle enables almost automatic grading of large parts of the exam. We explain our approach through a number of example problems, and explain why we believe that each of the kinds of problems we have selected are adequate measures of our intended learning outcomes. Finally, we discuss our experiences using the approach for the exam of a course on automated reasoning and suggest potential future work.

1 Introduction

At the Technical University of Denmark, we currently teach a MSc level course on automated reasoning using the Isabelle proof assistant [11] as our main tool. The course is a 5 ECTS optional course and the homepage is here:

<https://courses.compute.dtu.dk/02256/>

The course is an introduction to automatic and interactive theorem proving, and Isabelle is used to formalize almost all of the concepts we introduce during the course. We have developed a number of external tools to allow us to teach basic proofs in natural deduction and sequent calculus while slowly progressing towards showing students the full power of Isabelle. The learning objectives of the course are as follows:

1. explain the basic concepts introduced in the course
2. express mathematical theorems and properties of IT systems formally
3. master the natural deduction proof system
4. relate first-order logic, higher-order logic and type theory
5. construct formal proofs in the procedural style and in the declarative style
6. use automatic and interactive computer systems for automated reasoning
7. evaluate the trustworthiness of proof assistants and related tools
8. communicate solutions to problems in a clear and precise manner

We expect students to already know some logic and to be relatively proficient in functional programming before starting the course. Additionally, we expect students to have some basic knowledge of artificial intelligence algorithms for deduction. Our undergraduate program in computer science and engineering, which many of our students have completed, contains several courses that introduce students to these topics. Many of our MSc students are however from other universities or from other

undergraduate programs at our university, and may thus only be familiar with some of the topics we consider prerequisites. At our institution, prerequisites are not mandatory, but only recommended (partly to avoid a large administrative burden of attempting to determine equivalences between our undergraduate courses and courses at other universities). For this reason, we cannot expect students to be familiar with all prerequisites, and we thus endeavour to explain the prerequisites throughout the course.

During the course we need to test student learning outcomes. We ask students to hand in six assignments during the course, but we also have a two hour written exam at the end of the course. In the present paper, we will explain how we have designed this exam to make use of the automation afforded by Isabelle while testing understanding of both logic in the general sense and the specifics of proving theorems in Isabelle. We find that this setup works well for our course, which had 41 registered students at the time of the exam.

The exam is a two hour written exam with all aids allowed (including internet access, though students are of course not allowed to ask others for help during the exam). The exam sheet consists of an Isabelle file to be filled in as well as a “library” file containing the definitions needed for the proof systems used in the exam. The exam consists of five problems (weighted equally), each of which contains two questions. The problems in the exam can be solved in any order. The five problems concern the following topics:

1. Isabelle proofs without automation
2. Verification of functional programs in Isabelle/HOL
3. Natural deduction proofs
4. Sequent calculus proofs
5. General proofs in Isabelle/HOL with Isar

In the present paper, we will give examples of the kinds of questions we have designed for each topic, and explain why we believe these questions are adequate tests of learning outcomes for each topic. Each problem is weighted equally, but the two questions within each problem may have different weights. This allows us to create one easier and one harder question in some of the problems, which we find helps students not to “freeze” during the exam by giving them some early successes.

The examples and solutions given in this paper are drawn from a test exam which is also handed out to students so they can practice the format (with no impact on their grade) before the actual exam. At our institution, students are usually given full access to exam questions from previous years, often including solutions. Since we changed the format of the exam quite significantly this year, we introduced the test exam to give students a chance to ask questions about how to fill in the Isabelle file they were given, and to let students see the types of questions they would be asked in the actual exam. It should be noted that the exam problems mentioned in the present paper do not test all of the learning objectives mentioned above. The assignments handed in during the course test the remaining learning objectives and are also considered when giving the final grade.

One of the main benefits of using Isabelle for our exam is of course that much of the grading is almost automatic. Since Isabelle can check whether the proofs handed in by the students are correct, all that is needed to grade the exam are a few manual checks for style and to make sure that the students have not changed any definitions such that they are proving something different than what they were asked to. We have not experienced students maliciously changing definitions to make proofs easier, but some students misunderstand the assignment and write faulty proofs, then change definitions (in an obvious, non-obfuscated way) to make the proofs go through. For larger courses it may be desirable or necessary to completely automate the grading, but doing so puts much stricter constraints on the kinds of questions we can ask, and we have thus chosen not to do this. For instance, it becomes impossible to give partial

points to students who have the right proof idea, but hand in files with minor syntax errors (e.g. forgetting to end their otherwise correct proofs with a *qed*), which happens quite often. Some of the challenges in using the fully automatic approach have been described by Pierce [12]. For smaller courses, we believe that an oral exam or an exam based on a project may be both easier to set up and more beneficial, but at the scale of our course we simply do not have the time for these.

We also teach a BSc level course on logical systems and logic programming, which provides the prerequisite logic for many of our students. In that course the students are briefly introduced to Isabelle, but only in a much more limited setting, and they do not use Isabelle during the final exam [17].

In the next section, we will compare our approach to a selection of related work. In the sections after that, we will discuss and showcase some of the questions and solutions and explain which learning outcomes they test. We will then describe our experiences using the exam questions in practice and discuss possible future work before concluding.

2 Related work

The proof assistant Isabelle/HOL has previously been used to teach programming language semantics and their applications at MSc level [8]. This course specifically avoids focusing on logic itself, and instead uses both logic and Isabelle as tools to understand semantics. In contrast, the aim of our course is exactly to teach logic, and thus also parts of the inner workings of Isabelle. Additionally, our course does not focus very much on applications. On the other hand, the courses are similar in that we also do not focus on teaching the proof assistant for its own sake, but rather as a tool to understand and use logic. Importantly, the semantics course does not use Isabelle at all for the exam.

More recently, Isabelle/HOL has also been used to teach algorithms and data structures [10]. This course also does not focus on logic itself, but uses logic and Isabelle as tools to understand algorithms and data structures. Some introduction to Isabelle is however necessary, and this takes up the first third of the course. This course also uses Isabelle for the exam, but does not focus on logic.

The proof assistant Coq has been used for a broad series of textbooks on basic logic, programming languages, functional algorithms and separation logic [13]. These books also focus quite a bit on learning the Coq proof assistant, the user interface of which is further removed from pen-and-paper proofs than the interface of Isabelle. The books feature many exercises which are all to be completed in Coq, as well as scripts that can automatically grade most of the exercises. The exercises have been specifically designed such that they can be graded automatically. We believe that this is more easily done for proofs about the topics of programming languages and algorithms than for proofs about logic, since proofs about logic are, in our experience, more difficult to split into independent lemmas and require more auxiliary concepts. The books only cover basic logic, and mostly to demonstrate features of Coq. The exams in the courses they have been used in do not, as far as we know, use Coq, though homework exercises in Coq have been used [12].

The proof assistant Lean has been used to teach programming language semantics and the basic foundations of mathematics [2]. The course also focuses quite a bit on learning the Lean proof assistant itself. The exam of this course is on paper, but students are given problems using the Lean syntax and are also expected to write several of their answers using Lean syntax. The exam includes questions about the types of certain Lean terms, questions asking the students to write definitions in Lean, and questions asking students to prove certain properties of definitions in Lean. The proofs do not have to be actual Lean proofs, but do need to be very formal.

Lean has also been used to teach mathematics in general [3, 4]. A longer review of the use of proof

```

section <Problem 2 - 20%>
subsection <Question 1>
text <
Using only the constructors 0 and Suc and no arithmetical operators, define a recursive function
triple :: <nat => nat> and prove <triple n = 3 * n>.
>

fun triple :: <nat => nat> where
  <triple 0 = 0> |
  <triple (Suc n) = Suc (Suc (Suc (triple n)))>

lemma <triple n = 3 * n>
  by (induct n) simp_all

```

Figure 1: Problem 2 — The given text and the solution below.

assistants for general mathematics is available in [1].

The proof assistant Agda has been used to teach logic and functional programming in a course at our institution, but only for a single example, and not for the exam [7].

3 Isabelle proofs without automation

The first problem of the exam is designed to test whether the student understands the proof system of higher-order logic, which is the underpinning of Isabelle/HOL. We do this by asking students to prove the validity of simple logical formulas such as $p \leftrightarrow \neg\neg p$ and

$$(\exists x.\forall y.r(x,y)) \rightarrow (\forall y.\exists x.r(x,y))$$

The students must formally prove the validity of the formulas within Isabelle, and this problem thus also tests whether students master the natural deduction proof system used for Isabelle proofs and whether students actually know how to use Isabelle.

To avoid students simply using Isabelle’s automated tools such as Sledgehammer to prove the formulas, we provide a re-implementation of higher-order logic without any automation within Isabelle/Pure, and require students to use the proof rules from this system directly. We have recently described this re-implementation for intuitionistic and classical propositional logic [15]. For the course we have added the usual quantifiers to obtain first-order logic and finally we re-implement higher-order logic [16]. With this system, Isabelle can still often suggest an appropriate proof rule to use if the student first writes their intended subgoal, but only for single steps at a time, and not in all cases.

4 Verification of functional programs in Isabelle/HOL

The second problem of the exam is mainly designed to test whether students can program and prove in Isabelle/HOL, as well as whether the students can express properties of IT systems formally. We do this by asking students to implement very simple programs and prove simple properties of them. These problems are similar to some of the simpler exercises in the “Programming and Proving in Isabelle/HOL” tutorial [9]. An example question is given in fig. 1, which also contains a solution.

In designing questions such as these, it is quite difficult to obtain a reasonable amount of complexity in the programs and properties we ask students to write and prove. Even very simple properties can be very difficult to prove formally, and students have only a few minutes to solve the problem under the high pressure of the exam situation. We thus keep the programs and proofs very simple, aiming to design questions that are trivial for experienced Isabelle users, but which will be difficult to solve within a few minutes for someone who has never used Isabelle before. Note that the solution requires the use of induction, which means that the Sledgehammer tool can not be used to prove the property directly. The intention is that students who have actually followed the course and know how to use Isabelle should find the questions relatively easy, while any students who have not paid attention during the course should find the questions very difficult.

While the questions in the exam itself are quite easy, students are given a number of assignments with more complex questions during the course. Topics include programming with relations and sets, defining inductive types and recursive functions over them, and proving properties by structural induction, including rule induction over inductively defined relations.

5 Natural deduction and sequent calculus proofs

The next two problems of the exam involve proofs in a natural deduction system and a sequent calculus system, respectively. Both of these systems concern classical first-order logic with functions, but are simpler to use than Isabelle, which can be quite complicated. We have designed two external tools which allow students to write formal proofs in these proof systems, which are then automatically translated to Isabelle proofs. The natural deduction system, NaDeA, has a graphical front-end, while the sequent calculus system, SeCaV, has a textual front-end [5, 14]. Both of these front-ends are implemented as web applications, which students can access during the exam since they have access to the internet:

<https://nadea.compute.dtu.dk/>

<https://secav.compute.dtu.dk/>

The main idea behind these systems is to make it easier for students to understand which options they have when attempting to prove a formula valid. In NaDeA, the graphical interface means that students cannot input formulas with syntax mistakes. Additionally, NaDeA automatically filters the proof rules in the system such that only rules that can actually be applied in a given situation are shown in the graphical interface. Finally, the system handles the actual application of the proof rules automatically, including keeping track of branches in the proof tree.

The SeCaV tool is closer to apply-style proofs in Isabelle, except that the proof system is sequent calculus instead of natural deduction. The tool takes input as text, which means that students can make syntactical mistakes when inputting formulas and proof rules just as in Isabelle. The other main difference to NaDeA is that students must manually determine what applying a proof rule actually does and write out the result of applying the rule themselves. Since the proof system is much simpler than the proof system of Isabelle/HOL, however, the SeCaV system is able to give quite detailed warnings and error messages when students make mistakes.

We have described the pedagogical and logical design considerations of these systems in more detail elsewhere [6]. A key point is that we have a formalization of the syntax, semantics and proof systems for both NaDeA and SeCaV. We also have formal soundness and completeness theorems for the proof systems. The formalizations are part of the teaching materials during the course. In the exam, we ask students to prove both propositional and first-order formulas, and involve classical formulas such that

students will need to understand how to reason in classical logic. These problems thus test whether the student masters natural deduction, whether the student understands when classical proof rules are needed and how to use them, and whether the student can construct formal proofs using interactive computer systems. Both NaDeA and SeCaV run in the browser of each student independently, and any number of students can thus use the tools at the same time without performance issues.

6 General proofs in Isabelle/HOL with Isar

The final problem of the exam is designed to test student understanding of structured proofs in the Isar language of Isabelle/HOL [18], i.e. whether the student can construct formal proofs in the declarative style. The questions in this problem ask students to finish a proof of a more complicated property, e.g. the one seen in fig. 2a. The given question in fig. 2a contains a lemma and a comment with a “proof” of the lemma which has several mistakes. The student must then fix these mistakes and thus finish the proof. The idea behind this kind of question is to avoid the issue of devising properties of appropriate complexity also mentioned in section 4. By giving students a partially completed proof and asking them to fix the mistakes in it, we can ask students to prove properties that they would otherwise not be able to prove within the short time frame of the exam situation. This allows us to test student understanding of non-trivial proofs without succumbing to merely testing that students can use the automation of Isabelle/HOL.

We will showcase the concept by walking through a solution of a question from the test exam. In fig. 2b, we have uncommented the partially completed proof, which induces the Isabelle/jEdit development environment to highlight the errors in the proof. Notice the red underlining of several keywords and the brown backgrounds of some variables in the proof. More detailed information about the errors is also available either by moving the cursor over the underlined keywords or in a separate “Output” pane in Isabelle/jEdit. It should be noted that while Isabelle can highlight the errors and often provide some information about the cause, Isabelle is not able to say anything about how to fix the error. Starting from the top, the first error we notice is that the line `show ⟨p x⟩` contains the variable x , which has not been defined. Indeed, the given proof introduces a local constant a on the line immediately before this, so the proof can be fixed by changing every highlighted x into an a , as done in fig. 2c. Next, we look at the subproof attempting to prove the goal we just fixed. In fig. 2c, we notice that the keywords `show` and `assume` have been switched, so that the mistaken proof proceeds by first attempting to show the assumption, then assuming the conclusion at the end. This is fixed by simply exchanging the keywords, which has been done in fig. 2d. Next, the formula $\neg (\exists x. \neg p x)$, which is used to show \perp in the subproof, is not an assumption that is available at this point in the proof. This error requires some more knowledge of Isabelle/HOL than the previous ones to fix, since the error is not actually on this line, but instead in the very beginning of the proof, where a wrong assumption is made. The proof starts with the proof rule for classical contradiction, but the first assumption assumes the original formula, not its negation. By fixing this, as is done in fig. 3a, the appropriate formula becomes available as an assumption later in the proof. The final mistake in the proof is also related to the proof by contradiction, since the final line of the proof attempts to show \top while the correct way to end a proof by contradiction is to show \perp . Fixing this we obtain fig. 3b, which Isabelle verifies as being a correct proof.

While this lemma and its proof are quite simple, the time constraints of the exam situation means that students will need to identify and fix the mistakes very quickly. During the course, students are asked to prove more complicated lemmas from scratch during exercises and as assigned problems that are considered when giving the final grade. For these lemmas, students have much more time, and access to teaching assistants who can help steer students onto the right track. In the exam situation, we would like

section <Problem 5 - 20%>

subsection <Question 1>

text <

Replace \<proof> with the "proof ... qed" lines in the following comment and correct the errors such that the structured proof is a proper proof in Isabelle/HOL (do not alter the lemma text).

>

Lemma Foobar:

```

assumes < $\neg (\forall x. p\ x)$ >
shows < $\exists x. \neg p\ x$ >
\<proof>

(*
proof (rule ccontr)
  assume < $\exists x. \neg p\ x$ >
  have < $\forall x. p\ x$ >
  proof
    fix a
    show <p a>
    proof (rule ccontr)
      show < $\neg p\ a$ >
      then have < $\exists x. \neg p\ x$ > ..
      with < $\neg (\exists x. \neg p\ x)$ > assume  $\perp$  ..
    qed
  qed
  with < $\neg (\forall x. p\ x)$ > show  $\top$  ..
qed

```

(a) Problem 5

Lemma Foobar:

```

assumes < $\neg (\forall x. p\ x)$ >
shows < $\exists x. \neg p\ x$ >
proof (rule ccontr)
  assume < $\exists x. \neg p\ x$ >
  have < $\forall x. p\ x$ >
  proof
    fix a
    show <p a>
    proof (rule ccontr)
      show < $\neg p\ a$ >
      then have < $\exists x. \neg p\ x$ > ..
      with < $\neg (\exists x. \neg p\ x)$ > assume  $\perp$  ..
    qed
  qed
  with < $\neg (\forall x. p\ x)$ > show  $\top$  ..
qed

```

(b) Problem 5 — Step 1

Lemma Foobar:

```

assumes < $\neg (\forall x. p\ x)$ >
shows < $\exists x. \neg p\ x$ >
proof (rule ccontr)
  assume < $\exists x. \neg p\ x$ >
  have < $\forall x. p\ x$ >
  proof
    fix a
    show <p a>
    proof (rule ccontr)
      show < $\neg p\ a$ >
      then have < $\exists x. \neg p\ x$ > ..
      with < $\neg (\exists x. \neg p\ x)$ > assume  $\perp$  ..
    qed
  qed
  with < $\neg (\forall x. p\ x)$ > show  $\top$  ..
qed

```

(c) Problem 5 — Step 2

Lemma Foobar:

```

assumes < $\neg (\forall x. p\ x)$ >
shows < $\exists x. \neg p\ x$ >
proof (rule ccontr)
  assume < $\exists x. \neg p\ x$ >
  have < $\forall x. p\ x$ >
  proof
    fix a
    show <p a>
    proof (rule ccontr)
      assume < $\neg p\ a$ >
      then have < $\exists x. \neg p\ x$ > ..
      with < $\neg (\exists x. \neg p\ x)$ > show  $\perp$  ..
    qed
  qed
  with < $\neg (\forall x. p\ x)$ > show  $\top$  ..
qed

```

(d) Problem 5 — Step 3

Figure 2: Problem 5 — The given text and steps 1–3.

```

Lemma Foobar:
  assumes  $\langle \neg (\forall x. p\ x) \rangle$ 
  shows  $\langle \exists x. \neg p\ x \rangle$ 
proof (rule ccontr)
  assume  $\langle \neg (\exists x. \neg p\ x) \rangle$ 
  have  $\langle \forall x. p\ x \rangle$ 
  proof
    fix a
    show  $\langle p\ a \rangle$ 
    proof (rule ccontr)
      assume  $\langle \neg p\ a \rangle$ 
      then have  $\langle \exists x. \neg p\ x \rangle$  ..
      with  $\langle \neg (\exists x. \neg p\ x) \rangle$  show  $\perp$  ..
    qed
  qed
with  $\langle \neg (\forall x. p\ x) \rangle$  show  $\top$  ..
qed

```

(a) Problem 5 — Step 4

```

Lemma Foobar:
  assumes  $\langle \neg (\forall x. p\ x) \rangle$ 
  shows  $\langle \exists x. \neg p\ x \rangle$ 
proof (rule ccontr)
  assume  $\langle \neg (\exists x. \neg p\ x) \rangle$ 
  have  $\langle \forall x. p\ x \rangle$ 
  proof
    fix a
    show  $\langle p\ a \rangle$ 
    proof (rule ccontr)
      assume  $\langle \neg p\ a \rangle$ 
      then have  $\langle \exists x. \neg p\ x \rangle$  ..
      with  $\langle \neg (\exists x. \neg p\ x) \rangle$  show  $\perp$  ..
    qed
  qed
with  $\langle \neg (\forall x. p\ x) \rangle$  show  $\perp$  ..
qed

```

(b) Problem 5 — Step 5 / Solution

Figure 3: Problem 5 — Steps 4–5 / Solution.

to prevent the situation where solving the problem requires “getting the right idea”, since students will not have very much time to experiment with different approaches. We find that providing the students with an outline in the form of a partial proof helps prevent this issue. It would of course also be an option to give the students more time for the exam, but this would increase the workload for designing and grading the exam questions significantly, and make it harder to find space and time slots for the exam during the busy exam period.

7 Experiences in practice

We have used the exam problems in our course during the most recent exam in May. Of the 41 students registered for the exam, 36 passed the course. The remaining five students did not show up for the exam, and two additional students deregistered for the exam before the deadline.

The grades for the course were given based on both the exam and the assignments handed in during the course. The assignments were graded throughout the course and the students were given feedback after handing in each assignment. The final grade was given as an evaluation of the whole based on the assignments and the exam. There was a good correlation between the results in the assignments and the results in the exam, and no student grade moved more than a single step when comparing the overall grade to the preliminary grade suggested by their performance in the assignments.

The grade average for the course was 9.9 out of 12 (in the Danish national grading system, in which 12 is equivalent to the ECTS grade A and 10 is equivalent to the ECTS grade B). While this may seem high, it should be noted that this average only includes the students who actually showed up for the exam, and that it is common for Danish students to skip an exam if they do not feel prepared to take it, since they may then take a new exam later without “risking” passing with a low final grade due to lack of preparation for the exam. Skipping an exam in this manner does not impact the grades of a student as long as the student passes the exam within 3 attempts, but students who pass the exam are not allowed

to take a new exam. If we were instead to count students who did not show up for the exam with the lowest possible grade, the average would instead be 7.8 (slightly above ECTS grade C), which is in line with the intended average of the Danish scale. Another possible reason for the relatively high grade average is of course that our course is an advanced, non-mandatory course, which means that students self-select towards those who are motivated and interested in the topic of the course. At our institution, it is possible for students to audit courses for the first several weeks before committing to taking the course, and indeed a bit more than one third of the 69 initial students had dropped the course by the deadline for doing so. We presume that many of these students would have performed less well in the exam than those who stayed.

Adjusting the grades based on relative performance after seeing the exam results to obtain a specific distribution of grades (so-called “curving”) is illegal in Denmark, and is thus not an option. Instead, instructors are expected to adjust the difficulty of courses over several semesters to obtain the intended distribution of grades. We will consider making the course slightly more difficult next time it runs to move towards a lower average grade.

Students did not have any issues understanding how to use Isabelle or how to hand in their solutions during the exam. The submitted solutions also generally indicate that students understood what they were supposed to do to solve every problem, although they were of course not always able to provide a correct solution. Many students handed in Isabelle files with syntax errors, but these were generally quite easy to fix, and consisted primarily of abandoned proof attempts not being marked as such.

We generally found the exam submissions quite easy to grade. Correct solutions to the questions could in most cases be reviewed almost automatically by simply noticing that the student followed a reasonable approach and that Isabelle found no mistakes in their proof. Partial solutions were harder to grade, since it is of course difficult to estimate how close the student was to actually finishing the proof. This difficulty is no larger than it would have been for a pen and paper proof, however, and again Isabelle was of assistance by highlighting mistakes and missing parts in the proofs.

Since students had access to all aids (including the internet) during the exam, we should mention the possibility of cheating. The exam was supervised by several proctors to detect cheating, and mobile phones were not allowed in the exam hall. If students had cheated by communicating with each other, we would expect to see identical solutions, but we did not notice any irregularities in this regard. It might be possible to mitigate cheating by giving students slightly different problems, but we believe it would be hard to ensure that such problems would be of the same difficulty. We have developed versions of our auxiliary tools which can be downloaded and run locally (offline) in a browser, and in future iterations of the exams we consider not allowing students access to the internet.

In the rest of this section, we will describe the student evaluation of the exam, then discuss in more detail the performance and issues students displayed while attempting to solve each of the types of problems in our exam.

7.1 Student evaluation

All students at our institution are asked to anonymously evaluate exams they participate in. The evaluation form for the exam for the course was available to 43 students (including the two students who deregistered for the exam before the deadline). The evaluation was filled in by 17 students. Of the students who evaluated the course, 15 had not yet received their grade at the time of evaluation, while one student had. A single student filled in the evaluation by indicating that they had not been following the course.

The first question in the evaluation asked whether the exam corresponded to the teaching activities in

the course with regards to form, content, and level of complexity. 7 student answered that they completely agreed, 5 that they agreed, 3 that they felt neutral, and 1 that they disagreed. The average student thus agreed that the exam corresponded to the rest of the activities in the course. Several students wrote additional comments concerning their confusion with regards to proofs requiring classical proof rules. One student asked for more problems of “medium difficulty”.

The next question in the evaluation asked whether the examination form and content corresponded to the learning objectives of the course. 6 students answered that they completely agreed, 5 that they agreed, 4 that they felt neutral, and 1 that they disagreed. The average student thus agreed that the exam corresponded to the learning objectives of the course.

The final section of the evaluation was for further comments and suggestions. Several students wrote that they found that the exam matched what they expected, and that the test exam was helpful in this regard. Some students suggested to remove the exam and instead move to a project-based evaluation of their learning. One student had issues with figuring out how to correctly abort proof attempts to move on to the next question without getting errors in Isabelle. We will return to these suggestions in the section on potential future work.

7.2 Isabelle proofs without automation

The first problem of the exam seemed to be one of the hardest. The problem was, as mentioned, split in two questions, and one of them was significantly harder than the other. The easier question was intended to give students an “early win” before getting into the rest of the problems.

While most students completed the easier question, very few students completed both questions. About half of the students had a reasonable attempt at a proof for the harder question, but were missing significant parts. Several students did not hand in anything at all for the harder question.

This was essentially as expected, since this type of problem requires creativity and “getting the right idea” to solve. The harder question required the use of classical proof rules, which are in our experience generally confusing to students.

7.3 Verification of functional programs in Isabelle/HOL

The second problem of the exam was solved completely by most of the students. Several students only solved one of the questions fully, and only a few students solved none of the two questions. This may indicate that we were a bit too cautious with the difficulty of the questions in this problem. Our experience from similar problems given in the assignments throughout the course is that it is difficult to predict whether students will find program verification problems hard or easy. We thus decided to err on the side of making the problem too easy.

7.4 Natural deduction proofs

The third problem of the exam was solved completely by around half of the students. This problem was also split in an easier and a harder question, and all of the students solved the easier question. Several students had a reasonable attempt at a proof for the harder question, but were still missing significant parts.

The situation here is similar to the one in the first problem, and again the harder question required the use of classical proof rules. This again suggests that students find classical proof rules harder to use.

7.5 Sequent calculus proofs

The fourth problem of the exam was solved completely by all of the students. This may be slightly surprising, since one of the questions in this problem also required the use of classical proof rules. In the sequent calculus system of SeCaV, however, the students do not have to explicitly choose where in the proof to apply a classical proof rule, in contrast to the natural deduction systems of NaDeA and Isabelle/HOL. As such, proofs involving classical proof rules are not necessarily harder than proofs which do not when working in SeCaV. For this reason, we might consider increasing the difficulty of this type of problem in future exams.

7.6 General proofs in Isabelle/HOL with Isar

The fifth and final problem of the exam was solved completely by around half of the students. Most students solved one of the questions completely, and one of the questions partially. Only a few students did not solve any of the questions, and almost all of these students handed in partial solutions.

One of the questions was harder than the other, so this was as expected. This problem was also the one where partial solutions were easiest to obtain, since the questions asked students to fix a number of more or less independent errors.

8 Future work

There are many potential avenues for further work. As discussed above, we are continuously adjusting the difficulty of our exams based on our experiences each year. This necessarily also includes creating new problems for each exam. This presents an issue, since finding problems of the right difficulty is, in our experience, not very easy. An interesting potential line of research would accordingly be to develop guidelines and evaluation criteria for creating problems. We expect that this will not be very easy, since different students may find different problems difficult, and since it is generally difficult to predict how difficult a verification problem is. Rigorous studies of larger student populations and problem classes would be necessary to design good guidelines. One possibility would be tracking the behaviour of students while solving problems of certain types to determine the issues they run into, then designing criteria to take the most common errors and misunderstandings into account when estimating the difficulty of a problem. We expect that new tools would need to be developed to accommodate this kind of behavioural study.

A related potential line of research is the creation of more auxiliary tools to control the potential difficulty of problems. As detailed above, we have already created several tools which interface with Isabelle, but limit the options the students have, making it easier to estimate the difficulty of the problems we create. Our method of asking students to finish existing partial proofs is another way to limit the ways in which students can misunderstand the problem. We expect that there are many other ways which may already be in use at various institutions, and we encourage others to share their ideas and experiences. Additionally, we expect that experiences from other fields with similar problems, e.g. geometry or graph theory, could also be valuable, and conversely, that our experiences could be used to develop auxiliary tools for use in these fields. It seems that students have a remarkably hard time solving problems which require the use of formal classical reasoning, so constraining the difficulty of these is particularly interesting.

A completely different option is to dispense with a written exam altogether. We see two main options for evaluating the learning outcomes without a written exam: an oral exam based on assignments handed

in during the course, or a larger project with a report to be handed in at the end of the course. An oral exam works well for relatively small courses, but becomes infeasible as the course grows. At the scale of our course, we do not believe that an oral exam could be implemented without requiring excessive amounts of time for examinations. On the other hand, we do believe that a larger project with the objective of proving a number of theorems could be implemented without requiring much more time for grading than a written exam. Such a project would however need to be structured in a way that allows students to progress in reasonable steps without overwhelming them, and we expect that designing the project assignment would be a large amount of work. It might also be even harder to estimate the difficulty of such a project in comparison to exam problems.

Another line of potential future work concerns the development of problems that can be graded fully or nearly fully automatically. As detailed above, Isabelle already makes grading quite easy with our existing types of exam problems. For very large scale courses, however, fully automated grading may be necessary. While we could of course require that the files students hand in are free of syntax errors (such that they can be processed by the batch checking tools of Isabelle), we believe that such a restriction may be too severe. Our experience is that many students have a difficult time distinguishing between errors and correctly aborted proof attempts, and thus hand in files containing plenty of errors. One possibility might be to develop a tool for fixing Isabelle theories with errors, either automatically or by suggesting fixes for any errors found in the file. We do not expect that a practically usable version of such a tool would be very easy to develop.

In any case, if we were certain that all student submissions could be processed by the Isabelle batch checking tools, fully automated grading would still require careful design of the problems to ensure that they contain sensible milestones that can be graded individually. Inspiration for this might be taken from the automated grading scripts for the Software Foundations books, which contain exercises in the Coq proof assistant [12, 13]. We expect, however, that designing questions with reasonable milestones for our problems concerning natural deduction proofs and sequent calculus proofs would require a rework of our tools to support stopping partway through a proof. We currently also deduct a few points for very bad style, in the sense that proofs which are significantly longer or much more complicated than they need to be do not give full marks. We expect that automatically performing such a judgment would be difficult, but it might be possible to develop a tool which fully automatically grades reasonable proofs and can mark “problematic” submissions for manual review.

9 Conclusion

When conducting a course on automated reasoning using Isabelle, we need to test learning outcomes, and have chosen to do so in an exam setting. We have described our approach for doing this, and explained how Isabelle can help us quickly grade exam submissions. We have also explained the various kinds of questions in our exam, and how their design tests various aspects of student understanding of logic in general, and of formal proofs in Isabelle/HOL in particular, through a number of examples.

Students generally seem to understand how to solve the problems and are positive about the exam format. The main challenge in designing exams using this approach is to balance the difficulty appropriately, avoiding trivial problems while still allowing all students to make at least some progress in each problem. We have explained how simple programming problems can be combined with problems about completing larger proofs to test student understanding at several levels. Other courses may have different design constraints, especially those with many students, where fully automated grading may be desirable or necessary.

We believe that our experiences and the techniques described in this paper may also be useful for exams in other fields with similar constraints and a desire or need to fully or partially automate grading. Such fields might include geometry, graph theory, and any other field in which reasoning plays a significant role and where students may easily misunderstand questions or proofs. We expect that bespoke tools similar to NaDeA and SeCaV would be necessary to make formal reasoning in these fields accessible to students without previous experience with proof assistants.

Acknowledgements

We would like to thank Asta Halkjær From, Deniz Sarikaya, Frederik Lyhne Andersen, Simon Tobias Lund, and the anonymous reviewers for their helpful comments on drafts.

References

- [1] Jeremy Avigad & John Harrison (2014): *Formally Verified Mathematics*. *Communications of the ACM* 57(4), p. 66–75, doi:10.1145/2591012.
- [2] Jasmin Christian Blanchette (2021): *Logical Verification*. <https://lean-forward.github.io/logical-verification/2021/>. Course description.
- [3] Kevin Buzzard & Mohammad Pedramfar (2021): *The Natural Number Game, version 1.3.3*. https://www.ma.imperial.ac.uk/~buzzard/xena/natural_number_game/. Educational game.
- [4] Edmund M. Clarke & Jeremy Avigad (2015): *Interactive Theorem Proving*. <http://leanprover.github.io/cmu-15815-s15/>. Course description.
- [5] Asta Halkjær From, Frederik Krogsdal Jacobsen & Jørgen Villadsen (2022): *SeCaV: A Sequent Calculus Verifier in Isabelle/HOL*. In Mauricio Ayala-Rincon & Eduardo Bonelli, editors: *Proceedings 16th Logical and Semantic Frameworks with Applications*, Buenos Aires, Argentina (Online), 23rd - 24th July, 2021, *Electronic Proceedings in Theoretical Computer Science* 357, Open Publishing Association, pp. 38–55, doi:10.4204/EPTCS.357.4.
- [6] Asta Halkjær From, Jørgen Villadsen & Patrick Blackburn (2020): *Isabelle/HOL as a Meta-Language for Teaching Logic*. In Pedro Quaresma, Walther Neuper & João Marcos, editors: *Proceedings 9th International Workshop on Theorem Proving Components for Educational Software, ThEdu@IJCAR 2020, Paris, France, 29th June 2020*, *Electronic Proceedings in Theoretical Computer Science* 328, Open Publishing Association, pp. 18–34, doi:10.4204/EPTCS.328.2.
- [7] Asta Halkjær From & Jørgen Villadsen (2021): *Teaching Automated Reasoning and Formally Verified Functional Programming in Agda and Isabelle/HOL*. In: *10th International Workshop on Trends in Functional Programming in Education (TFPIE 2021) — Presentation Only / Online Papers*, pp. 1–20. Available at <https://wiki.tfpie.science.ru.nl/TFPIE2021>.
- [8] Tobias Nipkow (2012): *Teaching Semantics with a Proof Assistant: No More LSD Trip Proofs*. In Viktor Kuncak & Andrey Rybalchenko, editors: *Verification, Model Checking, and Abstract Interpretation - 13th International Conference, VMCAI 2012, Philadelphia, PA, USA, January 22-24, 2012*. *Proceedings, Lecture Notes in Computer Science* 7148, Springer, pp. 24–38, doi:10.1007/978-3-642-27940-9_3.
- [9] Tobias Nipkow (2021): *Programming and Proving in Isabelle/HOL (Tutorial)*. <https://isabelle.in.tum.de/doc/prog-prove.pdf>.
- [10] Tobias Nipkow (2021): *Teaching algorithms and data structures with a proof assistant (invited talk)*. In Catalin Hritcu & Andrei Popescu, editors: *CPP '21: 10th ACM SIGPLAN International Conference on Certified Programs and Proofs, Virtual Event, Denmark, January 17-19, 2021*, ACM, pp. 1–3, doi:10.1145/3437992.3439910.

- [11] Tobias Nipkow, Lawrence C. Paulson & Markus Wenzel (2002): *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. *Lecture Notes in Computer Science* 2283, Springer, doi:10.1007/3-540-45949-9.
- [12] Benjamin C. Pierce (2009): *Lambda, the Ultimate TA: Using a Proof Assistant to Teach Programming Language Foundations*. In: *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, ICFP '09, Association for Computing Machinery, New York, NY, USA, p. 121–122, doi:10.1145/1596550.1596552.
- [13] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hrițcu, Vilhelm Sjöberg & Brent Yorgey (2022): *Logical Foundations*. *Software Foundations* 1, Electronic textbook. Version 6.2, <http://softwarefoundations.cis.upenn.edu>.
- [14] Jørgen Villadsen, Andreas Halkjær From & Anders Schlichtkrull (2018): *Natural Deduction Assistant (NaDeA)*. In Pedro Quaresma & Walther Neuper, editors: *Proceedings 7th International Workshop on Theorem proving components for Educational software, THedu@FLoC 2018, Oxford, United Kingdom, 18 July 2018*, *Electronic Proceedings in Theoretical Computer Science* 290, Open Publishing Association, pp. 14–29, doi:10.4204/EPTCS.290.2.
- [15] Jørgen Villadsen, Asta Halkjær From & Patrick Blackburn (2022): *Teaching Intuitionistic and Classical Propositional Logic Using Isabelle*. In João Marcos, Walther Neuper & Pedro Quaresma, editors: *Proceedings 10th International Workshop on Theorem Proving Components for Educational Software*, (Remote) Carnegie Mellon University, Pittsburgh, PA, United States, 11 July 2021, *Electronic Proceedings in Theoretical Computer Science* 354, Open Publishing Association, pp. 71–85, doi:10.4204/EPTCS.354.6.
- [16] Jørgen Villadsen (2022): *A Formulation of Classical Higher-Order Logic in Isabelle/Pure*. *Journal of Logic and Artificial Intelligence (LAI)*. Accepted.
- [17] Jørgen Villadsen & Frederik Krogsdal Jacobsen (2021): *Using Isabelle in Two Courses on Logic and Automated Reasoning*. In João F. Ferreira, Alexandra Mendes & Claudio Menghi, editors: *Formal Methods Teaching*, Springer International Publishing, Cham, pp. 117–132, doi:10.1007/978-3-030-91550-6_9.
- [18] Makarius Wenzel (2021): *The Isabelle/Isar Reference Manual*. <https://isabelle.in.tum.de/doc/isar-ref.pdf>.

PROOFBUDDY: A Proof Assistant for Learning and Monitoring

Nadine Karsten
Technische Universität Berlin
Berlin, Germany
n.karsten@tu-berlin.de

Frederik Krogsdal Jacobsen
Technical University of Denmark
Kongens Lyngby, Denmark
fkjac@dtu.dk

Kim Jana Eiken
Technische Universität Berlin
Berlin, Germany
k.eiken@campus.tu-berlin.de

Uwe Nestmann
Technische Universität Berlin
Berlin, Germany
uwe.nestmann@tu-berlin.de

Jørgen Villadsen
Technical University of Denmark
Kongens Lyngby, Denmark
jovi@dtu.dk

Proof competence, i.e. the ability to write and check (mathematical) proofs, is an important skill in Computer Science, but for many students it represents a difficult challenge. The main issues are the correct use of formal language and the ascertainment of whether proofs, especially the students' own, are complete and correct. Many authors have suggested using proof assistants to assist in teaching proof competence, but the efficacy of the approach is unclear. To improve the state of affairs, we introduce PROOFBUDDY: a web-based tool using the Isabelle proof assistant which enables researchers to conduct studies of the efficacy of approaches to using proof assistants in education by collecting fine-grained data about the way students interact with proof assistants. We have performed a preliminary usability study of PROOFBUDDY at the Technical University of Denmark.

1 Introduction

The curriculum of a Bachelor study programme in Computer Science typically includes programming skills, technical understanding about computers, theoretical knowledge and math. Especially the latter two cause difficulties for students. Theoretical courses cover formal languages, automata, logic, complexity and computability, which all have in common that they build on mathematical structures and proofs about them. Hence Computer Science students have to write and verify proofs in several theoretical courses. This requires *proof competence*, which includes the following four specific competencies [7]: (1) *professional* competence describes the contextual knowledge about the proposition that has to be proved; (2) *representation* competence is the ability to write down a proof in sufficiently formal language; (3) *communication* competence is then understood as arguing about the procedure and solution of a proof; (4) finally, *methodological* competence summarizes three aspects [16]: proof scheme, proof structure and chain of conclusions.

All of these competencies are taught as part of introductory courses in Theoretical Computer Science, but proof competence with the above-mentioned facets is usually not an explicit part of the curriculum. As a result, students work through proofs that the teacher presents during lectures and try to replicate [33]. Most students fail this way. In [14], Frede and Knobelsdorf analyzed the homework and the written exams of an introductory course at Technische Universität Berlin (TUB). The result was that students make the most mistakes in exercises with proofs, regardless of the final grade. The main challenges in writing proofs are the usage of formal language while writing proofs and the ascertainment of whether a proof is complete and correct [19–21]. When students try to write proofs in introductory courses in Theoretical Computer Science, the failure rate is therefore high.

As we will see in the next section, proof assistants have often been suggested as a tool to improve proof competence. A proof assistant can be thought of as a functional programming language combined with a language for reasoning about programs. This allows users of proof assistants to write proofs in a formal, structured way, and get instant feedback on the correctness of their proofs from the proof checker. Proponents of using proof assistants in education claim many alleged benefits of the approach, but unfortunately, these claims are rarely supported by more than anecdotal evidence or surveys asking students to report their attitudes about the efficacy of the approach. Conversely, there are reasonable doubts about the efficacy of the approach: it is not clear that proof competences gained using proof assistants transfer to pen-and-paper proofs, proof assistants may enable a certain flavor of “brute-force proving” and introducing students to a proof assistant may use precious teaching time that could be better spent elsewhere.

Clearly, authors should do something to test these claims, but as we will see in the next section this is rarely done, at least partially due to a lack of tools that would enable the studies needed to test them. Inspired by similar approaches in studying the efficacy of didactic techniques when teaching functional programming and mathematics, we introduce PROOFBUDDY: an instrumented version of the Isabelle proof assistant [27, 28]. The idea of PROOFBUDDY is to collect fine-grained data about the interactions of students with the proof assistant and to use this data to conduct studies of the efficacy of various didactic approaches. Additionally, PROOFBUDDY has optional features to attempt to counteract some of the common doubts about the use of proof assistants in education. We have evaluated the usability of PROOFBUDDY and the data collection capabilities based on a set of research questions that we imagine future didactic studies could want answers to and found that the type of data collected by the tool is sufficient to answer these types of questions.

In summary, our contributions are:

- An overview of the current issues in studying the efficacy of using proof assistants in education.
- PROOFBUDDY: a version of the Isabelle proof assistant with instrumentation for collecting data about user interaction with the proof assistant.
- A preliminary evaluation of the usability of PROOFBUDDY from the perspective of students using the tool.
- A preliminary evaluation of the adequacy of the interaction data collected by PROOFBUDDY for conducting didactic studies.

In the next section, we will discuss existing uses of proof assistants in education, including the claimed benefits and drawbacks of using proof assistants in education, as well as some of the tools from the domain of functional programming that have inspired the design of PROOFBUDDY. We will outline potential approaches to mitigate the drawbacks and investigate the claimed benefits of using proof assistants in education in Section 3. Next, we describe the features (inspired by these potential approaches) and implementation of PROOFBUDDY in Section 4. In Section 5 we evaluate PROOFBUDDY from two perspectives: the adequacy of the collected data for conducting didactic studies and the usability from the perspective of students. Finally we outline future work in Section 6 before concluding in Section 7.

2 Related Work

This section covers work related to PROOFBUDDY, by which we mean not only descriptions of superficially similar tools, but also the articles and studies that have motivated the development of PROOFBUDDY, and the tools from similar fields that have inspired its design. We will cover a selection of reports on

using proof assistants in education, then summarize the claimed benefits and observed drawbacks of doing so. Finally, we will note some approaches from similar fields which have inspired the design of PROOFBUDDY.

2.1 Using Proof Assistants in Education

The idea to use proof assistants in education is not new, but the scientific results of these efforts have mostly been experience reports, and not rigorous evaluations of the efficacy of the approach. In this paper we will give only a short overview of approaches. Several courses target students in higher semesters and assume proof competence and functional programming skills [26, 36]. Knobelsdorf et al. [21] report on the use of the proof assistant Coq [3] to enhance proof competence, with the following result: while students were able to prove theorems with Coq after the course, their ability to write pen-and-paper proofs was not significantly better than before the course. Böhne and Kreitz [5] therefore attempt to improve pen-and-paper proof competences by requiring students to add comments in the formal Coq proof; nevertheless the improvement in writing pen-and-paper proofs was only minimal.

Some have also tried to integrate a proof assistant in first-year courses. Avigad [2] used the proof assistant Lean [25] in a logic course in 2015 to combine mathematical language and natural language in proofs. Avigad describes only anecdotal evidence and concludes that it seems necessary to develop quantitative methods and tools for data collection before carrying out a comparative study of the efficacy of the approach. Thoma and Iannone [39] describe a study where students could choose to use Lean in addition to the usual material in a first-year university course. The small number of students who used Lean in voluntary workshops used more mathematical language during an interview and structured their proofs better. Because Lean was not mandatory it is possible that only the students who were already doing well in the course chose to also use the proof assistant. Lean was also used for the Natural Number Game [9], which is a web application designed to teach formal proofs in a proof assistant using basic theorems in arithmetic and logic. Kreitz, Knobelsdorf and Böhne [4] consider a first-semester course where Coq is used in the lectures and the exercises to support the handling of formalisms, but this course was not realized. The web tool Waterproof by Wemmenhoven et al. [43] allows users to write Coq proofs with natural language. The authors developed a library to teach students in Analysis I where using the proof assistant was voluntary. The students who used the tool had better grades at the end of the course, but again this may simply be a result of self-selection bias. At TUB, we integrated Isabelle in a second semester course in 2021. We used Isabelle to introduce propositional and first-order logic step by step, but using Isabelle for the exercises and homework was optional. The students had fun in proving but it was time-consuming for them to step into Isabelle.

Lurch was a “proof-checking word processor” in which students could write natural language texts and mark certain parts of their input as having mathematical content, which the tool would then check [10, 11]. The tool focused on good user experience for beginners, and the expressivity of the formalism is unclear. The desktop version of the tool is no longer available, and the development of a new web-based version seems to have ceased in 2018 [12]. The web tool Carnap [22] offers a wide range of exercises in mathematics and logic. In 2017 there were several courses where different approaches with different students were tested. The author anecdotally found that Carnap offered exercises appropriate for different levels of competence. The AProS project is an effort to develop a number of courses and tools to teach logic and proofs [34]. These courses are computer-taught, not only computer-assisted, and include a number of interactive environments which are essentially proof assistants. Burke provides anecdotal evidence for the efficacy of the AProS project [8]. Clide [23, 30] is a web application based on Isabelle. Clide was implemented already in 2013 and consists of an editor integrated within a web interface, and an

Isabelle backend for checking proofs. Clide was not focused on learning, however, but focused on allowing collaborative writing and editing of proofs. The Clide application seems to no longer be available.

Many purpose-built proof assistants designed for learning specific topics have been developed, including Jape [37, 38], ProofWeb [17], SPA [31], SeCaV [15] and NaDeA [40–42].

The second and fifth author have previously conducted a study of student experiences using Isabelle, but our methods suffered from a lack of data on actual student behavior and interactions when using the proof assistant [18]. Knobelsdorf et al. limited themselves to manually observing student behavior and categorizing the types of questions asked by students [21]. Mariotti has conducted a number of studies on learning to prove with a dynamic geometry environment, but similarly describes only manual observations and interviews with students [24].

2.2 Claimed Benefits and Drawbacks of Using Proof Assistants in Education

As described in the previous sections, there have been many attempts to use proof assistants in education. In this section, we will summarize the claimed benefits and drawbacks of this approach in the literature.

We start with the claimed benefits of using proof assistants in education, of which there are many:

- Proof assistants are useful for teaching mathematics [2, 4, 10, 11, 17]
- Proof assistants are useful for teaching functional programming [4, 5, 17, 18, 21, 26]
- Proof assistants are useful for teaching logic [2, 4, 5, 8, 15, 17, 18, 21, 26, 34, 36, 41]
- Proof assistants are useful for teaching abstract thinking [4, 11, 34, 36]
- Proof assistants make the rules of formal reasoning clear [2, 4, 11]
- Proof assistants help students learn how to structure proofs [4, 5, 8, 10, 11, 17, 19, 21, 31]
- Students are helped by instant feedback on their proofs [2, 4, 8, 10–12, 15, 21, 26]
- Proof competence gained using a proof assistant transfers to pen-and-paper [2, 12, 26, 34]
- Proof assistants help students fix their proof errors as early as possible [4, 11]
- Proof assistants make it easier for students to get started on a proof [4, 15]
- Students consult formal definitions to gain understanding [18, 42]
- Students experiment with formal definitions to gain understanding [18, 26]
- Proof assistants help students experiment with proof ideas [4]
- Correcting assignments checked by proof assistants is easier [8]

Some authors have however also observed various difficulties that students encounter when trying to use proof assistants, including:

- Difficulties learning proof assistant syntax [2, 12]
- Difficulties understanding proof assistant error messages [2]
- Difficulties transferring proof competencies from proof assistants to pen-and-paper [5, 21]
- Difficulties stating properties formally [8]
- Technical difficulties in installing and using a proof assistant [8, 17, 21]
- Difficulties understanding the very expressive language of a proof assistant [17, 21]

- Difficulties remembering formal proof rules [21]

Some authors also note potential issues of using a proof assistant in education from the course instructor's point of view, including:

- The overhead of introducing a proof assistant to students [2, 26]
- The need to develop specialized proof scripts for each topic [4]
- The need to design exercises that cannot be solved by brute force [5, 11, 17, 21, 41]
- The worry that electronic exams using proof assistants may make it easier to cheat [17, 26]
- That proof assistants have automation that makes too many exercises trivial [17]

Finally, several authors note methodological issues in attempting to provide evidence for any of the above-mentioned claims:

- No suitable quantitative measures of proof competence exist [2]
- It is difficult to collect evidence about how students interact with proof assistants [5, 18]
- It is unclear how to compare the efficacy of approaches based on proof assistants with approaches based on pen-and-paper approaches in a fair way [18, 26]

2.3 Approaches From Similar Fields

Wrenn and Krishnamurthi have developed a tool to enable problem comprehension in functional programming by letting students test their understanding by writing property-based tests against a specification before writing code [46], and shown that students will use the tool voluntarily [47]. In studies using this tool, they found that students still had several types of misunderstandings that the tool or the lectures could have been improved to alleviate [45] and that course instructors had blind spots in predicting the misunderstandings students had [29]. Instrumenting the tool with data collection facilities allowed them to inspect student interactions with the tool in a fine-grained manner, and the tool thus helped instructors discover students' actual misunderstandings, which were different from what the instructors had imagined.

Aleven and Koedinger have designed a so-called Cognitive Tutor, which is a specially designed "proof assistant" instrumented with features for tracking student behavior and guiding students to explain their work [1]. They show that the use of this tool with the guidance features enabled improves student learning outcomes in geometry. The study was enabled by instrumenting the Cognitive Tutor with facilities for collecting fine-grained data about time spent and steps taken in the exercises completed by students using the tool.

3 Enabling Educational Research on Proof Assistants

As mentioned in the previous section, there are few rigorous studies about the influence of proof assistants on the learning of proof competences, but many claims about their efficacy. In this section we will discuss the importance of performing studies on the impact of proof assistants in education and the obstacles that we need to overcome to enable these studies. First, however, we discuss approaches to curtailing the observed difficulties students and instructors encounter when using proof assistants in education.

3.1 Making it Easier to Learn Proof Assistants

Proof assistants are generally not developed for students, but for expert users, and the learning curve is thus very steep. For beginning students, using any formal language can be a big challenge, and the rigorous languages of proof assistants can be even more difficult to break into. Instructors could potentially decrease the challenge by introducing the language one element at a time and developing tools that can guide the students for the specific learning goals of each activity. One could imagine this approach combined with adaptive learning tools to guide students at their own pace. A related issue is that proof assistants typically require users to remember not only the contents, but also the formal names of proof rules. Course developers could curtail this issue by providing an easy way to see the relevant proof rules in each learning activity without having to look them up in a larger list containing many irrelevant rules. Students can then engage with the content of the course instead of memorizing names.

Another issue is that students find error messages from proof assistants vague and confusing. This is typically a result of the language of the proof assistant being so expressive, and implementations being so general, that errors refer to concepts that students are not familiar with, e.g. type classes or functors. In many educational contexts, the full expressive power of the proof assistant is not necessary, and instructors could potentially flatten the learning curve by restricting the language to the fragment required by each activity. This would allow more specific error messages and perhaps even hints about why a specific application of a proof rule is wrong.

Some students have issues installing and using proof assistants from a purely technical perspective. Some of the issues could be reduced by developing web-based tools, which have the additional benefit of being easy to update such that students are always on the newest version of the tool in case any issues with the instrumentation are found during the course.

Finally, students have issues moving between proof assistants and traditional pen-and-paper proofs and statements. This occurs both when formalizing properties stated in natural language and when transferring the proof competences developed with a proof assistant to competences in writing pen-and-paper proofs. We are not convinced that it is possible to solve these issues by technical means, but approaches where the difference between proof assistant and pen-and-paper is gradually minimized seem promising.

3.2 Making it Easier to Use Proof Assistants in Education

Choosing to use a proof assistant in education unavoidably introduces the need to balance the time spent learning the actual course content and the time spent learning to use the proof assistant. The challenge is to design a course that uses the proof assistant as a tool instead of a course which teaches how to use proof assistants [26] (though a course which aims to do this with intention can of course also be valuable).

When designing exercises which are intended to be solved using proof assistants, there are several potential issues: exercises may be too easy to solve by brute force, automation in the proof assistant may make the exercises trivial and using electronic exercises for assignments or exams may make it easier for students to cheat. Restricting the language of the proof assistant (as described in the previous section) may also facilitate the design of exercises that are difficult to brute force and remove excessive automation.

The problem of designing exercises and supporting developments for each learning activity remains. We conjecture that this problem is mainly historical, i.e. caused by the fact that many courses have pen-and-paper exercises developed through several years. When designing new learning activities, we conjecture that using a proof assistant may actually expedite the development of good exercises since obscure or easily overlooked corner cases must be handled during the development and are thus not present to confuse students during the actual course.

3.3 Gathering Evidence

One of the main issues with implementing proof assistants in education is that there is little evidence of the efficacy of the approach in improving student proof competences. It is thus unclear whether spending time and resources on designing course material using proof assistants will actually result in students learning more. A compounding issue is that it is unclear how to measure proof competence across approaches based on proof assistants and approaches based on pen-and-paper in a quantitative way and without favoring one of the approaches. When considering studies, instructors must thus be careful to specify the learning objectives they are measuring precisely: is the intention of their course to improve proof competency *in general* or for instance specifically in pen-and-paper proofs?

The other major obstacle to conducting studies of proof assistants in education is that collecting objective, quantitative data about student interactions with proof assistants and learning outcomes is difficult. Most reports thus rely on surveys which ask students to self-report about their experiences or vague measures based on average grades in the course. One of the main objectives of PROOFBUDDY is to enable researchers to conduct studies of the efficacy of approaches to using proof assistants in education by collecting fine-grained data about the way students interact with proof assistants.

4 PROOFBUDDY

Inspired by the claimed benefits and drawbacks of using proof assistants for educational purposes described in the previous section, we have developed PROOFBUDDY: an instrumented version of the Isabelle proof assistant which is accessible through a web interface. The tool communicates with a full Isabelle proof assistant running on our server to check proofs and programs and additionally collects data about how users interact with the tool. It would also have been possible to instrument one of the usual Isabelle editors Isabelle/jEdit or Isabelle/VSCoDe. However, that would mean having to update the tool with every new Isabelle version, whereas communicating via the Isabelle server protocol allows us to develop against a more stable target. Besides the advantage that users do not have to install a special version of Isabelle, a web tool gives us the opportunity to add new features directly on the server without having users reinstall the tool. Through a management backend on the server it is possible to organize different courses, teacher and students and the collection of data (log-files and the submitted Isabelle theories). Furthermore a web interface yields more flexibility, and it is conceivable to add other languages like Coq or Lean while preserving a unified interface. Figure 1 shows a screenshot of PROOFBUDDY.

PROOFBUDDY is a web application and hence divided into two parts: the frontend, running in the browser, and the backend, running on the server. Figure 2 outlines the system architecture of PROOFBUDDY. The design decision to use Isabelle in the backend instead of the frontend came as a result of Isabelle’s development languages, ML and Scala, which are difficult to run in a browser. Furthermore it is easier to log the communication with the Isabelle server in the backend and save the theories which are sent to be verified. The frontend and backend of our application communicate via Socket.IO [35], allowing for instantaneous and bi-directional data transfer. Socket.IO is an event-driven library based on the WebSocket protocol [13]. In browsers where WebSockets are not supported, Socket.IO assures a stable connection via HTTP long-polling.

4.1 Frontend

The user interface of PROOFBUDDY mainly consists of three panes: an editor pane, an output pane and a PDF viewer. The *editor pane* is the main interactive component. An Isar proof of the “drinker’s principle”

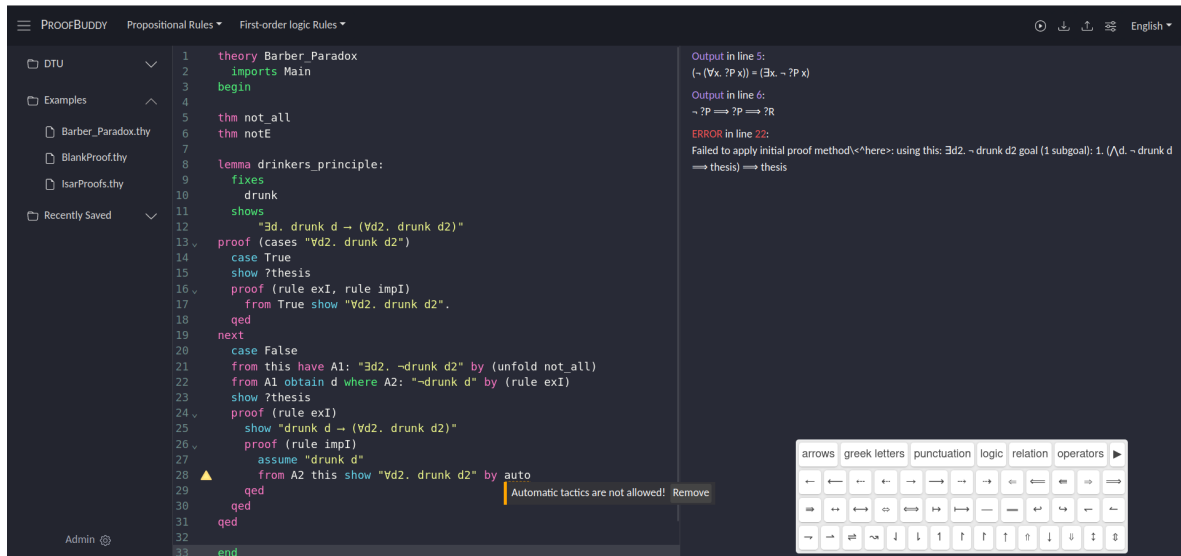


Figure 1: Screenshot of the interface of PROOFBUDDY with file browser on the left, editor in the middle and output messages on the right. Note also the linter popup (bottom center) and the symbol keyboard.

is shown in the *editor pane* in the middle of Figure 1.

PROOFBUDDY includes a parser for the language of Isabelle which enables syntax highlighting, autocompletion, code folding, bracket closing, search and replace functionality, etc. The editor of PROOFBUDDY thus functions like a typical integrated development environment (IDE). Since Isabelle is solely running in the backend, PROOFBUDDY does not have the ability of Isabelle/jEdit and Isabelle/VSCoDe to look up definitions and display type information by clicking on names of e.g. functions and theorems.

Instructors can restrict the language, and thus expressivity, of Isabelle on an activity-to-activity basis by adding linters, which disallow certain syntactic constructs. Linters display errors and warnings instantly in form of popups directly at the point where the failure occurs. We have implemented a linter based on regular expressions (as shown in Figure 1 and Figure 3) to warn about the usage of the automatic tactics `auto`, `simp`, `arith` and `blast`. At this point of development, the linters do not prevent users from asking Isabelle to check their work even if a linter detects the usage of prohibited syntax for the activity at hand. We could also restrict syntax by hiding definitions via a prelude, but this would not give users any specific information about their mistake when attempting to use e.g. a prohibited tactic, since the error from Isabelle would simply be about attempting to use an undefined tactic. Students can currently toggle the linter for automatic tactics through a switch on the user interface, but this feature can be removed to force students to only use automation in specific activities. It is thus possible to introduce features one at a time, ensuring that students can only use the features that they are supposed to learn in a given activity.

The *output pane*, located on the right-hand side, displays the feedback from Isabelle regarding the correctness of the current development. The closable *PDF viewer* allows the display of tutorials, lecture notes, exercise descriptions, etc. within the tool.

The collapsible *sidebar* contains a file browser and the profile of the current user. Users must log in to access PROOFBUDDY. User profiles are used to store progress and access previously created theories as well as to track interactions of individual users in the collected data. We did however implement a guest login to allow testing of PROOFBUDDY. The *toolbar* contains buttons to run the development (i.e.

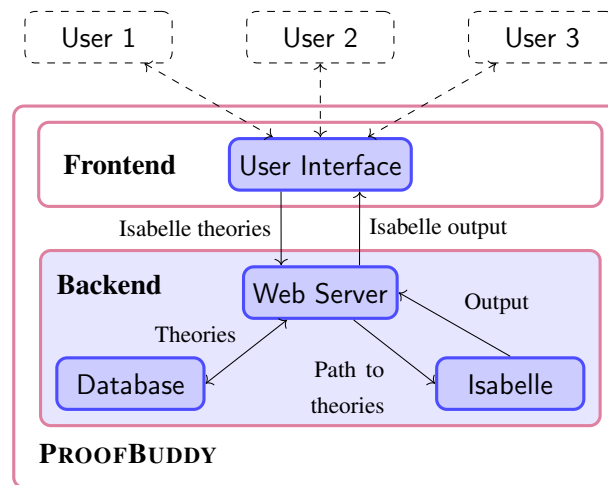


Figure 2: Architecture of PROOFBUDDY. Note that each user has their own instance of the frontend running in their browser, while there is only a single backend.

send the content of the editor to Isabelle for checking) and to download and upload Isabelle theories. Additionally, the toolbar contains dropdown menus which list the names and definitions of relevant proof rules on an activity-to-activity basis, as shown in Figure 3. Users can insert rule names at the current cursor position in the editor by clicking on a rule in the list.

Users can open the *symbol keyboard* by clicking the keyboard-button in the bottom right corner as seen in Figure 3. As shown in Figure 1, the keyboard offers an easy way to enter relevant mathematical and logical symbols, which instructors can configure on an activity-to-activity basis.

4.2 Backend

The backend of PROOFBUDDY is responsible for handling user authentication and data management, but its main functionality is to check the correctness of received developments using Isabelle.

The Isabelle proof assistant usually runs as two processes: a daemon, the Isabelle server, and an interactive application, the Isabelle client, through which theories can be sent to be checked by the server [44]. The Isabelle server listens on a TCP socket and allows for bi-directional communication with multiple clients following a protocol of structured messages. The PROOFBUDDY backend communicates with Isabelle via the Isabelle client by sending requests to check the developments it receives (checking is incremental as in the usual Isabelle development environments, so only theories with changes are checked). The messages consist of the session ID and a number of theories. The Isabelle server acknowledges the request and answers asynchronously whenever it finishes a task. The message of the Isabelle server includes all the usual error messages which are known from Isabelle/jEdit. Additionally, the Isabelle server periodically sends progress reports on ongoing tasks. The PROOFBUDDY backend starts the Isabelle client as a child process after ensuring that the Isabelle server is running. By writing to the input stream and reading the output stream of the Isabelle client, the backend is able to manage Isabelle sessions and check theories using Isabelle while simultaneously logging the interactions.

Isabelle sessions [44] consist of a collection of related Isabelle theories and build upon other Isabelle sessions. It is necessary to add Isabelle theories to an Isabelle session in order to check them using the

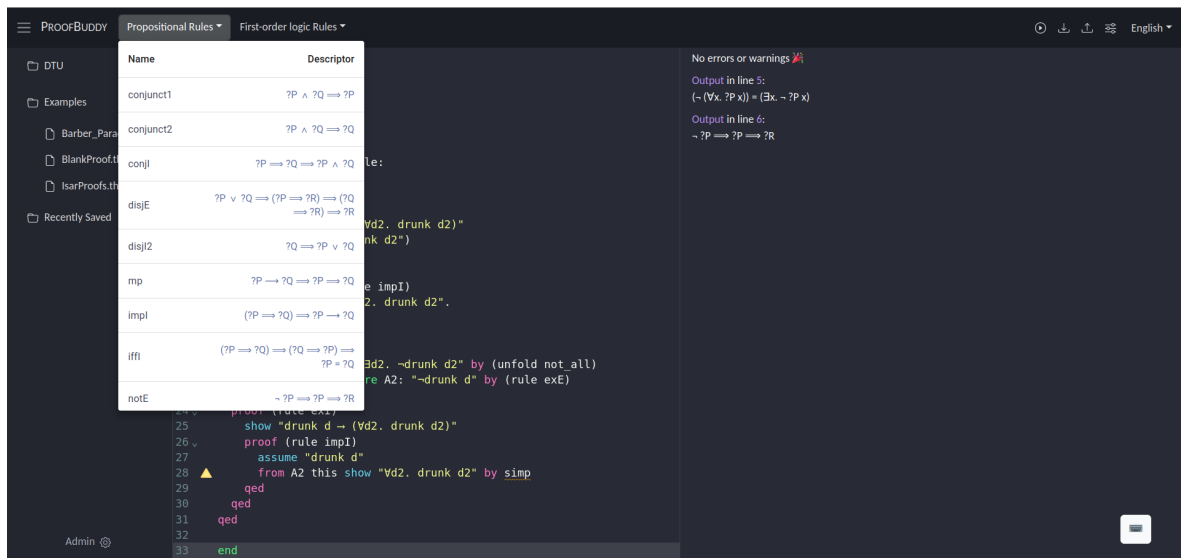


Figure 3: Screenshot of the interface of PROOFBUDDY with a reference list of proof rules. Note the yellow warning triangle and yellow underline of the “simp” tactic from the linter.

Isabelle server. The current version of PROOFBUDDY always uses Isabelle/HOL as the parent session. Hence, PROOFBUDDY makes the entire functionality of Isabelle/HOL available. For PROOFBUDDY (and thus Isabelle) to distinguish theories with the same name that are written by different users, each user has an individual Isabelle session. Adding theories to an Isabelle session automatically results in Isabelle checking the correctness and completeness of the theories. Once the Isabelle server has checked a development, the web server sends feedback from Isabelle directly to the frontend using Socket.IO, where it is formatted and displayed to the user.

4.3 Advantages of PROOFBUDDY

We designed PROOFBUDDY in response to the issues described in Section 3, and we will here highlight some of the ways in which PROOFBUDDY provides opportunities to alleviate these issues.

Due to the design decision of implementing a web interface, students do not need to install anything, and instructors can be certain that all students are using the same version of the tool.

The parser and the linters offer the possibility to restrict the input language used in PROOFBUDDY. The restriction of the input language makes it possible to introduce features one at a time, ensuring that students can only use the features that they are supposed to learn about in a given activity. The dropdown menus containing reminders of proof rules relevant to the activity at hand could also make it easier for students to engage with the content of the course by eliminating time spent looking up proof rules. Restricting the input language additionally could make it easier to design exercises that are harder to solve by brute force, e.g. by disallowing automation or certain proof rules. Restricting the input language could also be used to force students to write proofs in a style that resembles pen-and-paper proofs.

By formatting and specializing the messages from Isabelle in the frontend before displaying them, it is possible to give better feedback as described in Section 3. This would make it possible to add more, and also more useful, feedback to the output, depending on the failure or the previous behavior of the user.

Instructors can develop an interactive tutorial which forces students to complete exercises in a certain

order by structuring the activities in PROOFBUDDY using the built-in file browser and PDF viewer. This also allows for tailoring exercises to the pace and needs of the individual student.

The major benefit of PROOFBUDDY is the ability to monitor the progress and behavior of each student as they learn to program and prove in Isabelle/HOL. Therefore we are able to collect data to evaluate the behavior of students. This data consists of the contents of student theories saved whenever the student sends the theory for checking, annotated with time stamps. Monitoring the individual keystrokes and cursor position could also be possible to get even more detail. A high frequency of checking a theory can indicate that the student tries something without really thinking about why a proof is not accepted. If many students struggle with the same exercise, the instructor can improve the exercises or give a better introduction. When analyzing the failure it is also possible to adapt the next exercises. Using PROOFBUDDY offers two opportunities: (1) educational studies about the impact of proving with proof assistants can be performed with the help of the collected data and (2) learning analytics to improve and adapt the exercises to each student depending on the failures. In the next section, we will evaluate the extent to which PROOFBUDDY is useful for the first of these opportunities.

5 Evaluation of PROOFBUDDY

We evaluate PROOFBUDDY both from the perspective of researchers wanting to carry out didactic studies using data collected by the tool and from the perspective of students using the tool. Since we are not aware of any rigorous studies of how proof assistants impact learning, we are unable to test the adequacy of PROOFBUDDY with regards to concrete research questions from the literature, but instead evaluate PROOFBUDDY from the perspective of researchers by testing the adequacy of the data collected by PROOFBUDDY for answering research questions that we imagine could be part of future didactic studies. We note that PROOFBUDDY can of course only be used to answer questions about the behavior of students when interacting with a proof assistant.

Any comparative study between approaches based on proof assistants and approaches based on pen-and-paper would still need additional work to answer questions about the behavior of students when writing proofs on paper. Note also that this evaluation is not itself a didactic study, but simply an evaluation of the usefulness of the PROOFBUDDY tool for conducting future studies. The evaluation from the perspective of students concerns the usability of PROOFBUDDY, both in general and as compared to the standard editors distributed with Isabelle.

5.1 Didactic Context

We carried out the evaluation of PROOFBUDDY as part of the existing MSc-level course *Automated Reasoning* at the Technical University of Denmark (DTU) in the spring of 2023. This is an elective course for advanced Computer Science students, and the main topic is learning the use and theory of Isabelle. We thus expect students following the course to already be proficient at pen-and-paper proofs, and the main purpose of evaluating PROOFBUDDY using a population of students following the course is not to teach the students, but to collect data about how students interact with PROOFBUDDY. Table 1 contains a brief overview of the course plan and more information about the course can be found at <https://kurser.dtu.dk/course/2022-2023/02256>. We carried out the evaluation in week 7 of the course, at which point the students were already somewhat familiar with the use of Isabelle.

Weeks	Topics
1 to 2	Introduction, programming and proving, sequent calculus
3 to 4	Logic and proof beyond equality, natural deduction (first-order logic)
5 to 6	Isar: a language for structured proofs, natural deduction (higher-order logic)
7 to 9	Simple type theory
10 to 13	Formalized mathematics and computer science

Table 1: Course plan for the spring 2023 version of *Automated Reasoning* at DTU.

5.2 Research Questions

We evaluated the usefulness of PROOFBUDDY by attempting to answer the following research questions (RQ), which cover both perspectives of the evaluation:

RQ1 How many resources does PROOFBUDDY need to be usable by many students at once?

RQ2 What functionality do students miss in PROOFBUDDY compared to Isabelle?

RQ3 What issues did students encounter when interacting with PROOFBUDDY?

RQ4 Is the data collected by PROOFBUDDY useful for conducting didactic studies?

Sample research questions in didactic studies include:

SQ1 Which types of mistakes do students make while writing functional programs?

SQ2 Which types of mistakes do students make while writing proofs?

SQ3 Which types of feedback are immediately useful to students?

SQ4 How often do students use the possibility to get feedback from the type checker?

SQ5 How long do students need for an exercise?

Research question 4 includes a number of sample questions (SQ) which we imagine that researchers could want to answer when conducting a didactic study. It is important to note that obtaining answers to these questions would not by itself show anything about the efficacy of proof assistants in education as compared to pen-and-paper. Instead, answering these questions for multiple groups of students taught using different approaches (e.g. various approaches to using proof assistant or with pen-and-paper) could provide comparable measures which could potentially be used to show differences in efficacy. PROOFBUDDY enables such studies by allowing researchers to collect data about student interaction with proof assistants, but other approaches are still needed to collect data about student behavior using pen-and-paper, e.g. to conduct a randomized controlled trial to determine the efficacy of using proof assistants as compared to pen-and-paper.

5.3 Methods

We conducted the evaluation by asking the participants to solve a number of exercises using PROOFBUDDY and then fill out a short questionnaire. The exercises consisted of two parts: proving formulas directly in the natural deduction system of Isabelle/Pure, and programming and proving the correctness of simple program optimizations for an imperative programming language. The first and second authors instructed and supervised the students during the evaluation.

5.3.1 Population

The population studied in the evaluation consists of the students enrolled in the spring 2023 version of the course *Automated Reasoning*, of which there were 19. We included only those students who were physically present at the exercise session on March 17 in the evaluation. We recruited students by asking them to participate before the exercise session. Participants were not reimbursed financially or otherwise. Before starting the evaluation, we briefed participants about the purpose of PROOFBUDDY and the overall elements of the evaluation, including the research questions and the extent of data collected. Participants were not briefed about the interface of PROOFBUDDY such that we could study the discoverability of the features. Participants were not debriefed after the evaluation. 12 students opted to participate in the evaluation.

5.3.2 Ethical Considerations

We did not collect any personal identifiable information of participants during the evaluation, but did collect all text entered into PROOFBUDDY. We briefed participants about the extent of the data collection before the start of the evaluation. We considered students to have given informed consent to participate in the evaluation when they had listened to the briefing, read a letter describing their rights as participants, and started using PROOFBUDDY. Students in the course were free to choose not to take part in the evaluation and instead solve the exercises using their usual means instead of PROOFBUDDY. The exercises used for the evaluation were not used for grading students.

5.3.3 Threats to validity

Since the population consists of students who have voluntarily selected to follow the course *Automated Reasoning*, we may experience some selection bias. More importantly, the students following the course already had some experience with Isabelle, and might not experience the same issues as complete beginners. Our preliminary usability study is thus not generalizable to beginners. Additional selection bias may be introduced since there may be a correlation between students who actively follow the course (and so are present during the evaluation) and those who do not. The use of a questionnaire where we ask participants to report their own opinions on the importance of missing features and technical issues with PROOFBUDDY may introduce self-report bias. Manually categorizing participant questions asked during the evaluation may introduce researcher bias.

5.3.4 Analyses

To answer RQ1, we monitored the performance of the PROOFBUDDY tool and the resource usage (in terms of CPU and memory usage) of the server hosting the tool while performing the evaluation.

To answer RQ2 and RQ3 we asked participants to fill out a questionnaire about their experiences and any issues they may have encountered while using the PROOFBUDDY tool. We additionally recorded the questions participants asked the instructors while using PROOFBUDDY. The questionnaire contained the following questions (questions 1–3 concerning RQ2 and questions 4–5 concerning RQ3):

1. Do you usually use Isabelle/jEdit or Isabelle/VSCode?
2. Did you miss any features of your usual editor while using PROOFBUDDY?
3. If yes, how important do you think each of those features are on a scale from 1 to 5 (with 1 being not important and 5 being very important)?

4. Did you encounter any technical issues while using PROOFBUDDY?
5. If yes, how much did each of these issues affect your work on a scale from 1 to 5 (with 1 being not at all and 5 being very much)?

To answer RQ4, we used the data collected by PROOFBUDDY during the evaluation to set up mock analyses for the sample questions in Section 5.2. By setting up analyses for each of the sample questions we determined whether the data collected by PROOFBUDDY was adequate to perform analyses in didactic studies. The data collected by PROOFBUDDY includes error messages and warnings from the type checker and the proof checker of Isabelle. The syntactic, type level and tactic level mistakes students make while writing functional programs and proofs can be categorized using this data. The data also includes the actual programs and proofs, and can thus also be used to categorize semantic mistakes, i.e. programs that type check but do not have correct behavior or proofs that prove a different theorem than what was intended. SQ1 and SQ2 could be answered by categorizing the mistakes students make and ranking the categories by frequency. The data collected by PROOFBUDDY contains timestamps and includes both the actual theory and any messages from Isabelle. This data can be used to trace the evolution of the theory over time, noting the messages (e.g. errors and warnings) PROOFBUDDY has given the user between each step, and thus determining which messages lead to theories with fewer mistakes. SQ3 could be answered by categorizing the mistakes and messages, then ordering them chronologically and measuring the association between various messages and the disappearance of mistakes in the proofs. This is of course a simplified view of the causality between messages and mistakes, and studying the actual proof scripts in more detail could enable more refined analyses. The data collected by PROOFBUDDY consists of records of each instance of a student asking the tool for feedback. SQ4 could thus be answered by counting how many records exist in the dataset for each student. If we assume that students progress immediately from one exercise to the next, SQ5 could be answered by estimating the time spent on each exercise.

5.4 Observation Protocol

Nearly all students started the exercise session using PROOFBUDDY. One student had problems logging in, but trying a new account and password resolved the issue. At first all students got familiar with the tool and asked questions about the interface. Most of the students only dealt with the logic exercises, but some started with the program optimization exercise. During the exercise session we collected the questions the students asked the instructors because there was nearly no interaction between the students. We sorted the observed questions into the following categories:

1. **Creating proofs:** Problems during the process of proving, including recognizing and understanding assumptions and subgoals as well as how to start a proof. Questions about whether a proof is complete and correct also belong to this category;
2. **Mathematical inscriptions:** Problems with mathematical inscriptions, like how to write a proof step or argument in formal language;
3. **PROOFBUDDY usability:** Problems with the web interface of PROOFBUDDY, like menu functions and understanding the meaning of displayed elements;
4. **Working with Isabelle:** Problems with Isabelle, like choosing the correct rule, variable or structure and keywords of the Isar language, including typing errors;
5. **Logic:** Problems with aspects of logic, like syntax and semantics of definitions or the usage of quantifiers and other connectives;

6. **Functions:** Problems with aspects of functional programming, like syntax and semantics of functions or the concept of functions.

Next, we summarize the kinds of questions observed in each category:

1. **Creating proofs:** There was only one question related to induction.
2. **Mathematical inscriptions:** The students had no questions in this category.
3. **PROOFBUDDY Usability:** There were 17 question in this category, concerning: whether there is a difference in the syntax of Isabelle and PROOFBUDDY, how to use the special symbol keyboard, how to load and check theories (five questions).
4. **Working with Isabelle:** There were 19 questions in this category, concerning: syntax of Isabelle, the usage of strings in Isabelle and the structure of induction in Isabelle.
5. **Logic:** There were 21 questions in this category, concerning: scope of quantifiers, the freshness of variables when eliminating an existential quantifier (using the obtain keyword) and contradiction.
6. **Functions:** There were two questions in this category, concerning: conceptual understanding of functions, their semantics and programming.

In the first hour, most questions concerned the usage of PROOFBUDDY and the checking of theories. We observed that PROOFBUDDY needed a long time to answer the checking requests and therefore the students had to wait for feedback. For that reason students started using their own installations of Isabelle to check the proofs, but kept writing their proofs using PROOFBUDDY. The students seemed to like interacting with PROOFBUDDY, and several noted especially the possibility to choose a rule from the drop-down menu, without searching for the right rule among many irrelevant rules.

After the first half hour there were many questions about the syntax of Isabelle and logic. After one and a half hours the students worked on their own and asked only a few further questions. The first students left after two hours, and 6 students were still working after two and a half hours when the evaluation ended. All participants except one filled out the questionnaire before leaving.

5.5 Data Analysis

We analyzed the logged communication between frontend, backend and Isabelle Server (see Figure 5). This reveals that the average processing time of a request in the backend takes 16 seconds, where the handling of the request by the web server, i.e. writing the development to a theory file and attaching dependencies to the Isabelle request, takes only one second. The Isabelle Server used the remaining 15 seconds to check the development and compute the feedback. We did not notice any spikes in the processing time. Furthermore we analyzed the memory and CPU utilization of the server. Figure 4 shows the percentage of memory used during the evaluation. There is an increase in memory usage when starting PROOFBUDDY and another one when the students begin to log in. The measurement of the CPU utilization always resulted in the same value (0.64% at the user level).

The questionnaire included two different validated standard questionnaires and questions comparing Isabelle and PROOFBUDDY. 11 of 12 participants completed the survey and the answers support the impression we got by observing the exercise session.

We used the validated short version questionnaire System Usability Scale (SUS) [6] where PROOFBUDDY obtains an average score of 65/100. The students find PROOFBUDDY easy to use but do not see themselves using the tool frequently.

The second questionnaire used is the short version of the User Experience Questionnaire (UEQ) [32]. In this questionnaire, values between -0.8 and 0.8 represent a neutral evaluation of the corresponding

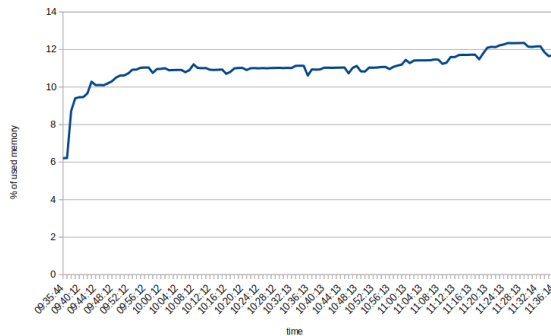


Figure 4: Utilization of the server memory when running PROOFBUDDY.

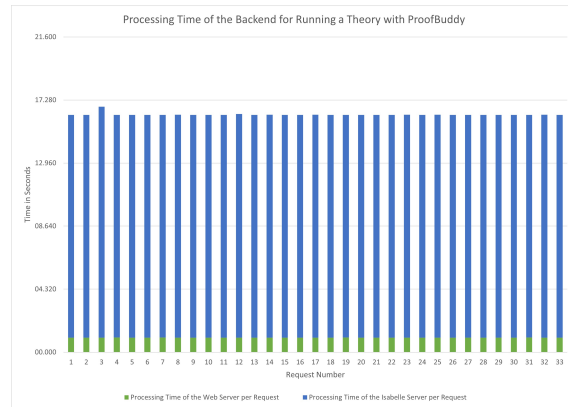


Figure 5: Vertical diagram displaying the processing time of the backend for checking theories with PROOFBUDDY.

scale, values > 0.8 represent a positive evaluation and values < -0.8 represent a negative evaluation. The range of the scales is between -2 (horribly bad) and $+2$ (extremely good). The pragmatic quality has a value of 0.775 where the efficiency has a bad score and the hedonic quality has a value of 0.727 .

The rest of the questionnaire concerned the comparison of PROOFBUDDY and Isabelle. Most of the students normally use Isabelle/jEdit and not Isabelle/VSCoDe. In comparison with Isabelle, the students miss the actual proof state and the immediate feedback and criticize the long waiting time and the error detection. There was a problem switching between the different activities, which overrides the editor content without saving the previous content anywhere. Sometimes the cursor jumps to the beginning of the editor. Often the students do not find the symbol keyboard and sometimes the keyboard overlaps with the editor or output panes. It sometimes happens that PROOFBUDDY does not detect all errors in a theory, even when copying the theory into Isabelle resulted in errors. Some students used PROOFBUDDY to write proofs but checked them in Isabelle/jEdit. Some students mentioned that PROOFBUDDY never gave them any feedback and instead seemed to become stuck when asked to check a theory.

5.6 Discussion

As we observed, students like the web interface of PROOFBUDDY, especially the dropdown menu. They find it intuitive to use the interface but there are some issues.

The main issue observed with the usability of PROOFBUDDY was the slow checking of theories. The bad score in efficiency in the UEQ follows from the long waiting time which was the main usability problem. Therefore students state in the SUS that they will not use PROOFBUDDY frequently. Additionally, some students reported that PROOFBUDDY did not report errors in their theories, indicating another issue with the communication between PROOFBUDDY and Isabelle. We note also that the measurement of CPU utilization always having the same value might have some connection to this issue.

Since the evaluation, the issues concerning the response time have been fixed. Firstly, there was an error in processing and combining chunks of large responses from the Isabelle server, in which case PROOFBUDDY appeared to check the theory indefinitely. Secondly, we specified the headless session option for the delay to consolidate the status of command evaluation (“headless_consolidate_delay”). The “headless_consolidate_delay”, which is by default set to 15 seconds, seems to correlate to the response

time of 15 seconds by the Isabelle server. By reducing the “headless_consolidate_delay” to 0.5 seconds, we observed a much shorter response time from the Isabelle server and therefore from PROOFBUDDY. During the evaluation, users were only able to access recently saved theories through the file browser. Reloading the page thus resulted in losing any progress made, since the users were unable to access the saved theories on the server. Users are now able to access saved theories through the file browser.

Despite the usability issues of PROOFBUDDY, the data collection features worked as expected. All of the data required to answer the sample question as described above was thus present in the PROOFBUDDY database. Unfortunately, the usability issues meant that many students stopped using PROOFBUDDY and returned to using their own Isabelle instead. In a real study this would of course not be acceptable, and so our first priority is remedying the observed usability issues.

6 Future Work

We plan to improve PROOFBUDDY. One approach for optimizing the response time is pre-assessing the Isabelle theories in the frontend. The frontend is already equipped with a parser for the Isabelle language. This opens the possibility to assess the theories for syntactical errors, such as missing brackets, or failures in the Isar proof syntax where special keywords are missing, prior to sending them to the backend. Only sending syntactically correct theories would reduce the server workload, which could improve the scalability of the PROOFBUDDY backend.

We additionally plan to extend the course management functionality by adapting the linter, the drop-down menu and the symbol keyboard to more exercises to support the declared learning objectives of the associated learning activities. Furthermore, we need to improve the file management, such that there is a database which stores courses and teacher profiles and for each student their own files within their profile. PROOFBUDDY already saves the Isabelle theory versions on the server explicitly, if the user presses the upload button, or implicitly, prior to the checking by Isabelle.

Another step will be to analyze the collected data automatically in order to answer the research questions: (1) what kind of feedback do students need during the proof writing process and (2) which parts of the proof are causing the students difficulties. By answering the first research question, the feedback of Isabelle can be extended to give more precise feedback depending on the kind of mistake. The problems discovered by answering the second research question could be solved by a hint to solve an exercise (adaptive learning) or by explaining a concept in the lecture one more time.

We have developed a new BSc-level course at TUB, where we focus on teaching how to create and properly write down proofs. The course is planned for summer 2023. Referring to our research questions above, the concept of the course is to enable substantially more feedback for students. Thus, we use PROOFBUDDY in addition to the help of the instructor and fellow students to provide the feedback much more directly (in real time) and also more individually. The idea is to start with propositional logic and first-order logic, because knowledge about these is a good foundation for structuring proofs [33]. Afterwards, we introduce inductive data structures and prove properties about them. We want to avoid that students just learn to use the tool without actually understanding the proofs that they develop with it. Therefore, the concept of the course aims to strengthen the mutual transformation between formal proofs—as developed with PROOFBUDDY—and traditional pen-and-paper proofs, in both directions. Confronted with these transformations, students are supposed to also learn that there are different degrees of formality to prove propositions. This course offers the opportunity to evolve the kind of feedback a proof assistant should give to learners that it assists during the learning process. We plan to iteratively change the feedback during several semesters and analyze if the students have fewer problems with the

exercises. Furthermore we will test new features in a Bachelor course dealing with graph structure theory at TUB in summer 2023, specifically for exercises about tree width. The students in this course have not used a proof assistant before. The focus of this study lies on the time students need to step into a proof assistant and how exercises have to be prepared such that students can manage the formal language.

There will also be an introductory course in theorem proving with Isabelle at TUB in summer 2023 where we start with PROOFBUDDY to introduce propositional logic, first-order logic and sets. The exercises will be solved in Isabelle/jEdit with the opportunity to use PROOFBUDDY instead.

We would also like to carry out a study about the impact of a proof assistant for learning proofs. This could be measured in an experiment where one group is taught with PROOFBUDDY and a control group only learns the topic with pen-and-paper proofs. But for this kind of study one has to measure the learning of proof competences in a fair way, which can be difficult as detailed above.

7 Conclusion

We introduce PROOFBUDDY, a web-based tool for guiding and monitoring the use of Isabelle/HOL by students. Proof assistants like Isabelle allow students to write proofs about functional programs, and many concepts in, e.g. logic can naturally be encoded as functional programs.

Unfortunately, not much evidence has yet been collected about the efficacy of using proof assistants in education. We have identified that one of the main issues in doing so seems to be a lack of tools for studying the interactions of students with proof assistants. Additionally, instructors have reasonable worries about teaching with very expressive languages, which may be difficult to learn and understand. PROOFBUDDY provides the opportunity to restrict the expressivity of Isabelle depending on the exercise and can therefore be used as a learning and teaching tool. Furthermore PROOFBUDDY allows researchers to collect data about student interactions for conducting studies. We log the data of the communication between frontend, backend and the Isabelle server. The collected data allows us to improve the error messages of Isabelle and analyze at which parts in a proof the students fail. Hence, we can offer hints and additional feedback. Such an approach allows instructors to gain insight into how students try to write and prove properties about inductive definitions and use pattern matching; it also enables us to carry out didactic research on student behavior. We expect that instructors could use the results of such studies to design guided exercises which support the learning progression of individual students.

References

- [1] Vincent A.W.M.M. Aleven & Kenneth R. Koedinger (2002): *An effective metacognitive strategy: learning by doing and explaining with a computer-based Cognitive Tutor*. *Cognitive Science* 26(2), pp. 147–179, doi:10.1207/s15516709cog2602_1.
- [2] Jeremy Avigad (2019): *Learning Logic and Proof with an Interactive Theorem Prover*. In Gila Hanna, David A. Reid & Michael de Villiers, editors: *Proof Technology in Mathematics Research and Teaching, Mathematics Education in the Digital Era* 14, Springer International Publishing, Cham, pp. 277–290, doi:10.1007/978-3-030-28483-1_13.
- [3] Yves Bertot & Pierre Castéran (2004): *Interactive theorem proving and program development. Coq'Art: The Calculus of inductive constructions*. Springer Berlin, Heidelberg, doi:10.1007/978-3-662-07964-5.
- [4] Sebastian Böhne, Maria Knobelsdorf & Christoph Kreitz (2016): *Mathematisches Argumentieren und Beweisen mit dem Theorembeweiser Coq*. In A. Schwill & U. Liuckey, editors: *HDI 2016 – 7. Fachtagung zur Hochschuldidaktik der Informatik, Commentarii informaticae didacticae* 10, Universitätsverlag Potsdam, pp. 69–80. Available at <http://eprints.cs.univie.ac.at/6838/>. (in German).

- [5] Sebastian Böhne & Christoph Kreitz (2018): *Learning how to Prove: From the Coq Proof Assistant to Textbook Style*. In Pedro Quaresma & Walther Neuper, editors: *Theorem proving components for Educational software, Electronic Proceedings in Theoretical Computer Science 267*, Open Publishing Association, pp. 1–18, doi:10.4204/eptcs.267.1.
- [6] John Brooke (1996): *SUS: A quick and dirty usability scale*. In Patrick W. Jordan, B. Thomas, Ian Lyall McClelland & Bernard Weerdmeester, editors: *Usability Evaluation in Industry*, chapter 21, CRC Press, London, England, pp. 189–195, doi:10.1201/9781498710411.
- [7] Esther Brunner (2014): *Mathematisches Argumentieren, Begründen und Beweisen*. Springer Spektrum Berlin, Heidelberg, doi:10.1007/978-3-642-41864-8. (in German).
- [8] Michael B. Burke (2006): *Electronic Media Review: Logic and Proofs (Web-based course)*. *Teaching Philosophy* 29(3), pp. 255–260, doi:10.5840/teachphil200629327.
- [9] Kevin Buzzard & Mohammed Pedramfar (2021): *The Natural Number Game*. Available at https://www.ma.imperial.ac.uk/~7Ebuzzard/xena/natural_number_game/.
- [10] Nathan C. Carter & Kenneth G. Monks (2013): *Lurch: a word processor built on OpenMath that can check mathematical reasoning*. In Christoph Lange, David Aspinall, Jacques Carette, James Davenport, Andrea Kohlhasse, Michael Kohlhasse, Paul Libbrecht, Pedro Quaresma, Florian Rabe, Petr Sojka, Iain Whiteside & Wolfgang Windsteiger, editors: *MathUI, OpenMath, PLMMS and ThEdu Workshops and Work in Progress at the Conference on Intelligent Computer Mathematics, CEUR Workshop Proceedings 1010*, Aachen, pp. 23:1–23:10. Available at <http://ceur-ws.org/Vol-1010/paper-23.pdf>.
- [11] Nathan C. Carter & Kenneth G. Monks (2013): *Lurch: a word processor that can grade students' proofs*. In Christoph Lange, David Aspinall, Jacques Carette, James Davenport, Andrea Kohlhasse, Michael Kohlhasse, Paul Libbrecht, Pedro Quaresma, Florian Rabe, Petr Sojka, Iain Whiteside & Wolfgang Windsteiger, editors: *MathUI, OpenMath, PLMMS and ThEdu Workshops and Work in Progress at the Conference on Intelligent Computer Mathematics, CEUR Workshop Proceedings 1010*, Aachen, pp. 4:1–4:10. Available at <http://ceur-ws.org/Vol-1010/paper-04.pdf>.
- [12] Nathan C. Carter & Kenneth G. Monks (2017): *A Web-Based Toolkit for Mathematical Word Processing Applications with Semantics*. In Herman Geuvers, Matthew England, Osman Hasan, Florian Rabe & Olaf Teschke, editors: *Intelligent Computer Mathematics, Lecture Notes in Computer Science 10383*, Springer International Publishing, Cham, pp. 272–291, doi:10.1007/978-3-319-62075-6_19.
- [13] Ian Fette & Alexey Melnikov (2011): *The WebSocket Protocol*. Technical Report 6455, Internet Engineering Task Force, doi:10.17487/RFC6455.
- [14] Christiane Frede & Maria Knobelsdorf (2018): *Explorative Datenanalyse der Studierendenperformance in der Theoretischen Informatik*. In N. Bergner, R. Röpke, U. Schroeder & D. Krömker, editors: *Hochschuldidaktik der Informatik - HDI 2018 - 8. Fachtagung des GI-Fachbereichs und Ausbildung/Didaktik der Informatik, Frankfurt, Germany, September 12-13, 2018*, Universitätsverlag Potsdam, pp. 135 – 149. Available at <http://eprints.cs.univie.ac.at/6870/>. (in German).
- [15] Asta Halkjær From, Frederik Krogsdal Jacobsen & Jørgen Villadsen (2022): *SeCaV: A Sequent Calculus Verifier in Isabelle/HOL*. In Mauricio Ayala-Rincon & Eduardo Bonelli, editors: *Proceedings 16th Logical and Semantic Frameworks with Applications, Buenos Aires, Argentina (Online), 23rd - 24th July, 2021, Electronic Proceedings in Theoretical Computer Science 357*, Open Publishing Association, pp. 38–55, doi:10.4204/EPTCS.357.4.
- [16] Aiso Heinze & Kristina Reiss (2003): *Reasoning and Proof: Methodological Knowledge as a Component of Proof Competence*. In Maria Alessandra Mariotti, editor: *Proceedings of the Third Conference of the European Society for Research in Mathematics Education*, pp. 1–10. Available at <http://www.lettredelapreuve.org/01dPreuve/CERME3Papers/Heinze-paper1.pdf>. Thematic Working Group 4, paper 5.
- [17] Maxim Hendriks, Cezary Kaliszzyk, Femke van Raamsdonk & Freek Wiedijk (2010): *Teaching logic using a state-of-the-art proof assistant*. *Acta Didactica Napocensia* 3(2), pp. 35–48. Available at http://dppd.ubbcluj.ro/adn/article_3_2_4.pdf.

- [18] Frederik Krogsdal Jacobsen & Jørgen Villadsen (2022): *Teaching Functional Programmers Logic and Metatheory*. In Peter Achten & Elena Machkasova, editors: *Trends in Functional Programming In Education, Electronic Proceedings in Theoretical Computer Science* 363, Open Publishing Association, pp. 74–92, doi:10.4204/eptcs.363.5.
- [19] Felix Kiehn, Christiane Frede & Maria Knobelsdorf (2017): *Was macht Theoretische Informatik so schwierig? Ergebnisse einer qualitativen Einzelfallstudie*. In Maximilian Eibl & Martin Gaedke, editors: *INFORMATIK 2017*, Gesellschaft für Informatik, Bonn, pp. 267–278, doi:10.18420/in2017_20. (in German).
- [20] Maria Knobelsdorf & Christiane Frede (2016): *Analyzing Student Practices in Theory of Computation in Light of Distributed Cognition Theory*. In: *Proceedings of the 2016 ACM Conference on International Computing Education Research, ICER '16*, ACM, New York, NY, USA, pp. 73–81, doi:10.1145/2960310.2960331.
- [21] Maria Knobelsdorf, Christiane Frede, Sebastian Böhne & Christoph Kreitz (2017): *Theorem Provers as a Learning Tool in Theory of Computation*. In: *Proceedings of the 2017 ACM Conference on International Computing Education Research, ICER '17*, Association for Computing Machinery, New York, NY, USA, pp. 83–92, doi:10.1145/3105726.3106184.
- [22] Graham Leach-Krouse (2017): *Carnap: An Open Framework for Formal Reasoning in the Browser*. In Pedro Quaresma & Walther Neuper, editors: *Proceedings 6th International Workshop on Theorem proving components for Educational software, Electronic Proceedings in Theoretical Computer Science* 267, Open Publishing Association, pp. 70–88, doi:10.4204/EPTCS.267.5.
- [23] Christoph Lüth & Martin Ring (2013): *A Web Interface for Isabelle: The Next Generation*. In Jacques Carette, David Aspinall, Christoph Lange, Petr Sojka & Wolfgang Windsteiger, editors: *Intelligent Computer Mathematics, Lecture Notes in Artificial Intelligence* 7961, Springer, pp. 326–329, doi:10.1007/978-3-642-39320-4_22.
- [24] Maria Alessandra Mariotti (2019): *The Contribution of Information and Communication Technology to the Teaching of Proof*. In Gila Hanna, David A. Reid & Michael de Villiers, editors: *Proof Technology in Mathematics Research and Teaching, Mathematics Education in the Digital Era* 14, Springer International Publishing, Cham, pp. 173–195, doi:10.1007/978-3-030-28483-1_8.
- [25] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris Doorn & Jakob Raumer (2015): *The Lean Theorem Prover (System Description)*. In Amy P. Felty & Aart Middeldorp, editors: *Automated Deduction - CADE-25, Lecture Notes in Computer Science* 9195, pp. 378–388, doi:10.1007/978-3-319-21401-6_26.
- [26] Tobias Nipkow (2012): *Teaching Semantics with a Proof Assistant: No more LSD Trip Proofs*. In V. Kuncak & A. Rybalchenko, editors: *Verification, Model Checking, and Abstract Interpretation, Lecture Notes in Computer Science* 7148, Springer, pp. 24–38, doi:10.1007/978-3-642-27940-9_3.
- [27] Tobias Nipkow (2022): *Programming and Proving in Isabelle/HOL*. Available at <https://isabelle.in.tum.de/doc/prog-prove.pdf>.
- [28] Tobias Nipkow, Lawrence C. Paulson & Markus Wenzel (2002): *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. *Lecture Notes in Computer Science* 2283, Springer, doi:10.1007/3-540-45949-9.
- [29] Siddhartha Prasad, Ben Greenman, Tim Nelson, John Wrenn & Shriram Krishnamurthi (2022): *Making Hay from Wheats: A Classsourcing Method to Identify Misconceptions*. In Ilkka Jormanainen & Andrew Petersen, editors: *Proceedings of the 22nd Koli Calling International Conference on Computing Education Research, Koli Calling '22*, Association for Computing Machinery, New York, NY, USA, pp. 2:1–2:7, doi:10.1145/3564721.3564726.
- [30] Martin Ring & Christoph Lüth (2014): *Real-time collaborative Scala development with Clide*. In Heather Miller & Philipp Haller, editors: *Proceedings of the Fifth Annual Scala Workshop*, ACM, pp. 63–66, doi:10.1145/2637647.2637652.
- [31] Anders Schlichtkrull, Jørgen Villadsen & Asta Halkjær From (2018): *Students' Proof Assistant (SPA)*. In Pedro Quaresma & Walther Neuper, editors: *Proceedings 7th International Workshop on Theorem proving components for Educational software, Electronic Proceedings in Theoretical Computer Science* 290, pp. 1–13, doi:10.4204/EPTCS.290.1.

- [32] Martin Schrepp, Andreas Hinderks & Jörg Thomaschewski (2017): *Design and Evaluation of a Short Version of the User Experience Questionnaire (UEQ-S)*. *International Journal of Interactive Multimedia and Artificial Intelligence* 4(6), pp. 103–108, doi:10.9781/ijimai.2017.09.001.
- [33] John Selden & Annie Selden (2009): *Teaching Proving by Coordinating Aspects of Proofs with Students' Abilities*. In Despina A. Stylianou, Maria L. Blanton & Eric J. Knuth, editors: *Teaching and Learning Proof Across the Grades*, chapter 20, Routledge, pp. 339–354, doi:10.4324/9780203882009.
- [34] Wilfried Sieg (2007): *The AProS Project: Strategic Thinking & Computational Logic*. *Logic Journal of the IGPL* 15(4), pp. 359–368, doi:10.1093/jigpal/jzm026.
- [35] Socket.IO (2023): <https://socket.io/>. Online. Accessed April 18 2023.
- [36] Alexander Steen, Max Wisniewski & Christoph Benz Müller (2016): *Einsatz von Theorembeweisern in der Lehre*. *Commentarii informaticae didacticae* 10, pp. 81–92. Available at <https://orbilu.uni.lu/bitstream/10993/40839/1/cid10.pdf>. (in German).
- [37] Bernard Sufrin & Richard Bornat (1996): *User interfaces for generic proof assistants part I: Interpreting gestures*. Available at <https://www.cs.ox.ac.uk/people/bernard.sufrin/jape.org.uk/DOCUMENTS/CURRENT/UITP96-1.pdf>.
- [38] Bernard Sufrin & Richard Bornat (1998): *User interfaces for generic proof assistants part II: Displaying proofs*. Available at <https://www.cs.ox.ac.uk/people/bernard.sufrin/jape.org.uk/DOCUMENTS/CURRENT/UITP96-2.pdf>.
- [39] Athina Thoma & Paola Iannone (2022): *Learning about proof with the theorem prover LEAN: The abundant numbers task*. *International Journal of Research in Undergraduate Mathematics Education* 8, pp. 64–93, doi:10.1007/s40753-021-00140-1.
- [40] Jørgen Villadsen, Andreas Halkjær From & Anders Schlichtkrull (2018): *Natural Deduction and the Isabelle Proof Assistant*. In Pedro Quaresma & Walther Neuper, editors: *Theorem proving components for Educational software, Electronic Proceedings in Theoretical Computer Science* 267, Open Publishing Association, pp. 140–155, doi:10.4204/eptcs.267.9.
- [41] Jørgen Villadsen, Andreas Halkjær From & Anders Schlichtkrull (2019): *Natural Deduction Assistant (NaDeA)*. In Pedro Quaresma & Walther Neuper, editors: *Theorem proving components for Educational software, Electronic Proceedings in Theoretical Computer Science* 290, Open Publishing Association, pp. 14–29, doi:10.4204/eptcs.290.2.
- [42] Jørgen Villadsen, Alexander Birch Jensen & Anders Schlichtkrull (2015): *NaDeA: A Natural Deduction Assistant with a Formalization in Isabelle*. *Journal of Applied Logics* 4(1), pp. 55–82. Available at <http://www.collegepublications.co.uk/downloads/ifcolog00010.pdf>.
- [43] Jelle Wemmenhove, Thijs Beurskens, Sean McCarren, Jan Moraal, David Tuin & Jim Portegies (2022): *Waterproof: educational software for learning how to write mathematical proofs*. arXiv:2211.13513.
- [44] Markus Wenzel (2022): *The Isabelle system manual*. Available at <https://isabelle.in.tum.de/doc/system.pdf>.
- [45] Jack Wrenn & Shriram Krishnamurthi (2021): *Reading Between the Lines: Student Help-Seeking for (Un)Specified Behaviors*. In Otto Seppälä & Andrew Petersen, editors: *Proceedings of the 21st Koli Calling International Conference on Computing Education Research*, Koli Calling '21, Association for Computing Machinery, New York, NY, USA, pp. 14:1–14:6, doi:10.1145/3488042.3488072.
- [46] John Wrenn & Shriram Krishnamurthi (2019): *Executable Examples for Programming Problem Comprehension*. In Robert McCartney, Andrew Petersen, Anthony Robins & Adon Moskal, editors: *Proceedings of the 2019 ACM Conference on International Computing Education Research*, ICER '19, Association for Computing Machinery, New York, NY, USA, pp. 131–139, doi:10.1145/3291279.3339416.
- [47] John Wrenn & Shriram Krishnamurthi (2020): *Will Students Write Tests Early Without Coercion?* In Nick Falkner & Otto Seppälä, editors: *Proceedings of the 20th Koli Calling International Conference on Computing Education Research*, Koli Calling '20, Association for Computing Machinery, New York, NY, USA, pp. 27:1–27:5, doi:10.1145/3428029.3428060.



Verifying a Sequent Calculus Prover for First-Order Logic with Functions in Isabelle/HOL

Asta Halkjær From¹ · Frederik Krogsdal Jacobsen¹

Received: 29 March 2023 / Accepted: 28 March 2024
© The Author(s) 2024

Abstract

We describe the design, implementation and verification of an automated theorem prover for first-order logic with functions. The proof search procedure is based on sequent calculus and we formally verify its soundness and completeness in Isabelle/HOL using an existing abstract framework for coinductive proof trees. Our analytic completeness proof covers both open and closed formulas. Since our deterministic prover considers only the subset of terms relevant to proving a given sequent, we do the same when building a countermodel from a failed proof. Finally, we formally connect our prover with the proof system and semantics of the existing SeCaV system. In particular, the prover can generate human-readable SeCaV proofs which are also machine-verifiable proof certificates. The abstract framework we rely on requires us to fix a stream of proof rules in advance, independently of the formula we are trying to prove. We discuss the efficiency implications of this and the difficulties in mitigating them.

Keywords Isabelle/HOL · SeCaV · First-Order Logic · Prover · Soundness · Completeness

1 Introduction

While there are many automated theorem provers capable of proving theorems involving very large formulas and many lemmas, very few of them have formalized proofs of metatheoretical properties such as soundness and completeness. This leads to issues of trust: how do we know that the answers returned by automated theorem provers are actually correct? And do we know that our automated theorem provers will actually be able to prove what we want them to? Even those provers that can generate proof certificates to support their answers may not always be trustworthy, since some proof techniques lead to proofs that are very difficult to follow for a human, and are thus difficult to *manually* check for correctness. Proof certificates can also be mechanically checked, but doing so means trusting another piece of software to be correct.

✉ Frederik Krogsdal Jacobsen
fkjac@dtu.dk

Asta Halkjær From
ahfrom@dtu.dk

¹ DTU Compute, Section for Algorithms, Logic and Graphs, Technical University of Denmark, Kongens Lyngby, Denmark

Formalizing the soundness and completeness of a prover provides two crucial benefits. With a soundness result, we know that the prover will not erroneously accept an invalid formula and output a wrong proof of the formula. Advanced features and optimizations thus cannot cause unforeseen flaws in the prover, since their correctness is formally proven. Completeness of the prover is especially useful in combination with the possibility of generating readable proof certificates. With formalized completeness, we can use the prover as a tool to generate step-by-step proofs of any valid formula, and the prover can thus also be used to gain understanding, e.g. by students trying to understand why a counter-intuitive formula is valid. While there are some systems with formalized metatheories, they rarely include executable provers, often cannot generate proof certificates, and are often quite limited in their expressive power (cf. Sect. 1.1).

In this paper, we present an automated theorem prover for first-order logic with functions based on sequent calculus. We formalize its soundness and completeness in Isabelle/HOL. We reuse the syntax and semantics of first-order logic from the Sequent Calculus Verifier (SeCaV) system [15] (Sect. 2.1). We state the soundness and completeness of the prover with respect to the SeCaV proof system, its semantics and a bounded semantics that we introduce here. The prover can generate human-readable and machine-verifiable SeCaV proofs for valid formulas.

Our formalization instantiates an abstract framework of coinductive proof trees by Blanchette et al. [11] (Sect. 2.2). By instantiating the framework with concrete functions implementing our sequent calculus, the framework builds a prover for us (Sect. 3). By discharging further proof obligations, the framework proves that any proof tree built by our prover is either finite or contains an infinite path with certain properties. We then build either a SeCaV proof from the finite tree (Sect. 4) or a countermodel from the failed proof attempt (Sect. 5). As far as we are aware, we are the first to use the framework to prove soundness and completeness of an executable prover (as opposed to simply a calculus).

Our prover is deterministic and fair and works on finite sequents. To handle the quantifiers, we must thus build our countermodel in a Herbrand universe that contains only the subset of terms that actually appear in the failed proof. This idea is inspired by Ben-Ari's textbook proof [2], where terms are either variables or constants, and by Ridge and Margetson's Isabelle proof [41], where only variables are considered. We are not aware of any previous formalization of this construction that handles functions. We consider all terms in our Herbrand universe, including those with free variables, yielding completeness for both open and closed formulas.

The prover is free software and the source code is available as supplementary material. This consists of around 3000 lines of Isabelle/HOL and 1300 lines of supplementary Haskell. The supplementary Haskell code is not involved in the proof procedure, but handles parsing of the input formula and conversion of proofs to both human-readable and machine-verifiable formats, and can in some cases shorten proofs after they have been found by merging proof steps. The user is thus only required to trust the Haskell code if the generated proof certificates are used without verifying them.

We summarize our main contributions:

- A formally verified sound and complete automated theorem prover for full first-order logic with functions.
- An analytic proof of completeness for both open and closed formulas for a deterministic prover via a bounded semantics.
- A method of translating the prover-generated certificates of validity into human-readable and machine-verifiable proofs in SeCaV.

- A concrete application of the abstract completeness framework and a demonstration of how to obtain soundness and completeness of an executable prover using the framework as a starting point.

We summarize the results and discuss the generated proofs, the challenges encountered during the verification, the prover limitations, and future work in Sect. 6, before concluding in Sect. 7.

This paper is a revised and extended version of a paper presented at ITP 2022 [17] with a number of improvements. We give a more thorough explanation of the SeCaV syntax, including the formalized definitions of substitution and all of the rules of its proof system. This has allowed us to give full listings throughout, where the previous paper contained only abridged definitions. Moreover, we have included and explained the relevant Isabelle code from the abstract completeness framework, elucidating the actual definitions underlying the prover. We also give a more thorough explanation of the proof search procedure, including previously absent details of the procedures for finding terms in a sequent and determining whether a sequent is an axiom. Using the added detail in the definitions, we explain additional technical and conceptual details in several proofs. This improvement to the presentation includes more explanation of Isabelle syntax, locales and library functionality throughout the paper. Finally, we have extended our discussion of possible optimizations and post-processing of proofs and included an additional example proof.

1.1 Related Work

The present paper is a much improved version of the work started in the second author's Master's thesis [22]. The Sequent Calculus Verifier (SeCaV) is an existing proof system, and both soundness and completeness have been proven for the system [18]. The system has been used to teach students in several courses at the Technical University of Denmark [19, 52]. An online tool called the SeCaV Unshortener has been developed to allow input of proofs in a simple format, which is then translated to an Isabelle proof [15].

Our prover is based on the abstract completeness framework by Blanchette et al. [7, 11]. The framework contains a simple example prover for propositional logic, and the original application of the framework was a technical result (see [7]) in the formalization of the metatheory of the Sledgehammer tool for automated theorem proving within Isabelle/HOL [8]. Blanchette et al. [11] have used the framework to formalize soundness and completeness of a calculus for first-order logic with equality and in negation normal form. Their search is nondeterministic and they do not generate an executable prover like we do. We thus extend their work by using the framework to prove soundness and completeness of an executable prover.

A number of other systems have formally verified metatheories. NaDeA (Natural Deduction Assistant) by Villadsen et al. [51] is a web application that allows users to prove formulas with natural deduction. The metatheory of a model of the system is formalized in Isabelle/HOL, and the application allows export of proofs for verification in Isabelle. The Incredible Proof Machine by Breitner [12] is a web application that allows users to create proofs using a specialized graphical interface. The proof system is as strong as natural deduction, and a model of the system is formalized in Isabelle using the abstract framework by Blanchette et al. [11]. Neither system includes automated theorem provers; they are essentially simple proof assistants designed to aid students in understanding logical systems.

THINKER by Pelletier [38] is a proof system and an attached automated theorem prover. THINKER is a natural deduction system designed to allow for what the author calls “direct proofs”, as opposed to proofs based on reduction to a resolution system. THINKER was perhaps the first automated theorem prover designed specifically with “naturality” in mind, as a reaction to the indirectness of resolution-based proof systems. MUSCADET by Pastre [37] is also an automated theorem prover based on natural deduction. The system distinguishes itself by also supporting usage of prior knowledge such as previously proven theorems through a Prolog knowledge base.

While there are many very advanced automated theorem provers such as Vampire [26], Zipperposition [3] and Z3 [13], their metatheory and implementations are rarely formalized. As a first step towards formally verifying modern provers, Schlichtkrull et al. [43] have formalized an ordered resolution prover for *clausal* first-order logic in Isabelle/HOL. Jensen et al. [23] formalized the soundness, but not the completeness, of a prover for first-order logic with equality in Isabelle/HOL. Villadsen et al. [53] verified a simple prover for first-order logic in Isabelle/HOL with the aim of allowing students to understand both the prover and the formalization. That work was based on an earlier formalization by Ridge and Margetson [41], but it simplified both the prover and the proofs to enable easier understanding by students. Neither of these two provers provide support for functions or generation of proof certificates.

Blanchette [9] gives an overview of a number of verification efforts including the metatheory of SAT solvers [10, 14, 33, 34, 46] and certificate checkers [28, 29], SMT solvers [30, 32, 48], the superposition calculus [39], resolution [40, 42, 45], a number of non-classical logics [20, 21, 44, 50, 54] and a wide range of proof systems for classical propositional logic [35, 36]. Some of these efforts are part of the IsaFoL project (Isabelle Formalization of Logic). Part of the goal is to develop “a methodology for formalizing modern research in automated reasoning”. Our work points in this direction too, by formally verifying a non-saturation-based prover.

2 Background

In this section, we briefly introduce the two existing projects we build on: the Sequent Calculus Verifier (SeCaV) system and the abstract framework by Blanchette et al. [11]. We have not modified these projects in any way for our use, and their designs thus significantly influence the design of our prover.

2.1 The Sequent Calculus Verifier

The SeCaV system is a one-sided sequent calculus for first-order logic with functions. Constants are encoded as functions with arity 0. Figure 1 gives the syntax of terms and formulas (denoted p, q, \dots) as Isabelle/HOL datatypes. Constructor names like *Fun* for function symbols and *Var* for variables are capitalized. Parameterized datatypes are written in postfix notation in Isabelle, e.g. the type *tm list* of lists containing terms. The system uses de Bruijn indices to identify variables, while functions and predicates are named by natural numbers. Besides predicates, the system includes implication, disjunction, conjunction, existential quantification, universal quantification and negation (in that order in Fig. 1). Predicates and functions take their arguments as ordered lists of terms, which may be empty. Sequents (denoted y, z, \dots) are ordered lists of formulas.

datatype $tm = Fun\ nat\ (tm\ list) \mid Var\ nat$

datatype $fm =$
 $Pre\ nat\ (tm\ list) \mid Imp\ fm\ fm \mid Dis\ fm\ fm \mid Con\ fm\ fm \mid Exi\ fm \mid Uni\ fm \mid Neg\ fm$

Fig. 1 The syntax of Sequent Calculus Verifier terms and formulas (parentheses added for clarity)

definition $shift\ e\ v\ x \equiv \lambda n. \text{if } n < v \text{ then } e\ n \text{ else if } n = v \text{ then } x \text{ else } e\ (n - 1)$

primrec semantics-term and semantics-list where

$semantics-term\ e\ f\ (Var\ n) = e\ n \mid$
 $semantics-term\ e\ f\ (Fun\ i\ l) = f\ i\ (semantics-list\ e\ f\ l) \mid$
 $semantics-list\ e\ f\ [] = [] \mid$
 $semantics-list\ e\ f\ (t\ \#\ l) = semantics-term\ e\ f\ t\ \#\ semantics-list\ e\ f\ l$

primrec semantics where

$semantics\ e\ f\ g\ (Pre\ i\ l) = g\ i\ (semantics-list\ e\ f\ l) \mid$
 $semantics\ e\ f\ g\ (Imp\ p\ q) = (semantics\ e\ f\ g\ p \longrightarrow semantics\ e\ f\ g\ q) \mid$
 $semantics\ e\ f\ g\ (Dis\ p\ q) = (semantics\ e\ f\ g\ p \vee semantics\ e\ f\ g\ q) \mid$
 $semantics\ e\ f\ g\ (Con\ p\ q) = (semantics\ e\ f\ g\ p \wedge semantics\ e\ f\ g\ q) \mid$
 $semantics\ e\ f\ g\ (Exi\ p) = (\exists x. semantics\ (shift\ e\ 0\ x)\ f\ g\ p) \mid$
 $semantics\ e\ f\ g\ (Uni\ p) = (\forall x. semantics\ (shift\ e\ 0\ x)\ f\ g\ p) \mid$
 $semantics\ e\ f\ g\ (Neg\ p) = (\neg semantics\ e\ f\ g\ p)$

Fig. 2 The semantics of the Sequent Calculus Verifier (# separates head and tail of a list)

The semantics of a formula is due to Berghofer [4], who models the universe as a type variable, and we do the same for now (we will revisit this choice in Sect. 5.3.1). Besides this implicit universe, the interpretation consists of an environment e for variables, a function denotation f and a predicate denotation g . The semantics of the system is standard and defined using the three recursive functions in Fig. 2. The function *semantics-term* evaluates a term to a member of the universe under an environment and a function denotation, while *semantics-list* does this for a list of terms. The *semantics* function itself defines the meaning of the object-logical connectives using the connectives from the meta-logic in Isabelle/HOL. The *shift* function handles shifting de Bruijn indices when interpreting quantifiers, such that the environment maps 0 to the meta-quantified element of the domain. We say that a sequent is valid when, under all interpretations, some formula in the sequent is satisfied.

The system has a number of proof rules, which are displayed in Fig. 3 (abusing set notation for the membership and inclusion relations on lists—see the formalization for their encodings as primitive recursive functions). The rules should be read from the bottom up, since we generally work backwards from the sequent we wish to prove. The rules are classified according to Smullyan’s uniform notation [47].

The first proof rule, BASIC, terminates the branch and applies when the sequent contains both a formula and its negation. Isabelle/HOL allows pattern matching only on the head of a list, so to simplify the specification of this rule, the positive formula must come first.

The structural EXT rule can be applied to change the position of formulas in a sequent (permutation), duplicate an existing formula (contraction) and remove formulas that are not needed (weakening). This rule is crucial, since most rules in the system work only on the first formula in a sequent. Duplicating a formula is necessary if a quantified formula needs to be instantiated several times, since γ -rules (starting with GAMMA) destroy the original formula.

$$\begin{array}{c}
\frac{\text{Neg } p \in z}{\Vdash p, z} \text{ BASIC} \qquad \frac{\Vdash z \quad z \subseteq y}{\Vdash y} \text{ EXT} \qquad \frac{\Vdash p, z}{\Vdash \text{Neg} (\text{Neg } p), z} \text{ NEGNEG} \\
\\
\frac{\Vdash p, q, z}{\Vdash \text{Dis } p \, q, z} \text{ ALPHADIS} \qquad \frac{\Vdash \text{Neg } p, q, z}{\Vdash \text{Imp } p \, q, z} \text{ ALPHAIMP} \\
\\
\frac{\Vdash \text{Neg } p, \text{Neg } q, z}{\Vdash \text{Neg} (\text{Con } p \, q), z} \text{ ALPHACON} \qquad \frac{\Vdash p, z \quad \Vdash q, z}{\Vdash \text{Con } p \, q, z} \text{ BETA CON} \\
\\
\frac{\Vdash p, z \quad \Vdash \text{Neg } q, z}{\Vdash \text{Neg} (\text{Imp } p \, q), z} \text{ BETAIMP} \qquad \frac{\Vdash \text{Neg } p, z \quad \Vdash \text{Neg } q, z}{\Vdash \text{Neg} (\text{Dis } p \, q), z} \text{ BETADIS} \\
\\
\frac{\Vdash p [\text{Var } 0/t], z}{\Vdash \text{Exi } p, z} \text{ GAMMAEXI} \qquad \frac{\Vdash \text{Neg} (p [\text{Var } 0/t]), z}{\Vdash \text{Neg} (\text{Uni } p), z} \text{ GAMMAUNI} \\
\\
\frac{\Vdash p [\text{Var } 0/\text{Fun } i \ []], z \quad i \text{ fresh}}{\Vdash \text{Uni } p, z} \text{ DELTAUNI} \\
\\
\frac{\Vdash \text{Neg} (p [\text{Var } 0/\text{Fun } i \ []]), z \quad i \text{ fresh}}{\Vdash \text{Neg} (\text{Exi } p), z} \text{ DELTAEXI}
\end{array}$$

Fig. 3 Proof rules for the Sequent Calculus Verifier

The NEGNEG rule removes a double negation from the first formula in a sequent. It can be considered an α -rule, but we keep it separate from the others because it does not generate two formulas. The ALPHADIS rule decomposes disjunctions (and similar for the ALPHAIMP and ALPHACON rules). The BETADIS rule decomposes negated disjunctions and requires that two sequents are proven separately, creating branches in the proof tree (and similar for the BETA CON and BETAIMP). This essentially moves the connective into the proof tree itself, since both branches now need to be proven separately. The GAMMAEXI rule instantiates an existential quantifier with any term t by substituting t for variable 0 in the quantified formula. Due to the use of de Bruijn indices, the definition of substitution is quite complicated. The definition of the substitution function sub can be seen in Fig. 4. The notation $p [\text{Var } 0/t]$ should be interpreted as the function application $sub \ 0 \ t \ p$. The GAMMAUNI rule is similar. The DELTAEXI rule instantiates a negated existential quantifier in the first formula in a sequent with a fresh constant function, with fresh here meaning that the function identifier does not already occur anywhere in the sequent. The fresh constant cannot have any relationship to other terms in the sequent: it is *arbitrary*. Thus, we could have used any other term without affecting the validity of the formula, which is exactly what is needed to prove a universally quantified (“there does not exist”) formula. The freshness condition can easily be checked by recursively going through the formulas in the sequent (see the formalization for the primitive recursive functions implementing this). The DELTAUNI rule is similar.

The proof system in Fig. 3 has been formally verified to be sound and complete with regards to the semantics in Fig. 2 by From et al. [18]. We use these results to relate our prover to SeCaV.

primrec *inc-term* and *inc-list* where
$$\begin{aligned} \text{inc-term } (\text{Var } n) &= \text{Var } (n + 1) \mid \\ \text{inc-term } (\text{Fun } i \ l) &= \text{Fun } i \ (\text{inc-list } l) \mid \\ \text{inc-list } [] &= [] \mid \\ \text{inc-list } (t \ # \ l) &= \text{inc-term } t \ # \ \text{inc-list } l \end{aligned}$$
primrec *sub-term* and *sub-list* where
$$\begin{aligned} \text{sub-term } v \ s \ (\text{Var } n) &= \\ &(\text{if } n < v \ \text{then } \text{Var } n \ \text{else if } n = v \ \text{then } s \ \text{else } \text{Var } (n - 1)) \mid \\ \text{sub-term } v \ s \ (\text{Fun } i \ l) &= \text{Fun } i \ (\text{sub-list } v \ s \ l) \mid \\ \text{sub-list } v \ s \ [] &= [] \mid \\ \text{sub-list } v \ s \ (t \ # \ l) &= \text{sub-term } v \ s \ t \ # \ \text{sub-list } v \ s \ l \end{aligned}$$
primrec *sub* where
$$\begin{aligned} \text{sub } v \ s \ (\text{Pre } i \ l) &= \text{Pre } i \ (\text{sub-list } v \ s \ l) \mid \\ \text{sub } v \ s \ (\text{Imp } p \ q) &= \text{Imp } (\text{sub } v \ s \ p) \ (\text{sub } v \ s \ q) \mid \\ \text{sub } v \ s \ (\text{Dis } p \ q) &= \text{Dis } (\text{sub } v \ s \ p) \ (\text{sub } v \ s \ q) \mid \\ \text{sub } v \ s \ (\text{Con } p \ q) &= \text{Con } (\text{sub } v \ s \ p) \ (\text{sub } v \ s \ q) \mid \\ \text{sub } v \ s \ (\text{Exi } p) &= \text{Exi } (\text{sub } (v + 1) \ (\text{inc-term } s) \ p) \mid \\ \text{sub } v \ s \ (\text{Uni } p) &= \text{Uni } (\text{sub } (v + 1) \ (\text{inc-term } s) \ p) \mid \\ \text{sub } v \ s \ (\text{Neg } p) &= \text{Neg } (\text{sub } v \ s \ p) \end{aligned}$$
Fig. 4 The definition of substitution used in the Sequent Calculus Verifier

2.2 Abstract Frameworks for Soundness and Completeness

Blanchette et al. [11] have formalized an abstract framework to facilitate soundness and completeness proofs by coinductive methods. In particular, they give abstract definitions that can be instantiated to a concrete sequent calculus or tableau prover. They facilitate proofs in the Beth-Hintikka style: the search “builds either a finite deduction tree yielding a proof ... or an infinite tree from which a countermodel ... can be extracted.” The framework consists of a number of Isabelle/HOL locales that must be instantiated and in return provide various definitions and proofs.

Locales [1, 24] allow the abstraction of definitions and proofs over given parameters. As an example, consider groups in algebra defined by a carrier set, a binary operation and the group axioms. With a locale, these can be specified abstractly and a number of operations and results can then be given for the abstraction. Later, we can instantiate the locale with a concrete group by providing the carrier set and binary operation and then proving that the group axioms are fulfilled. We then obtain instantiations of the results for our concrete group.

In this section, we give an overview of the locales provided by the abstract framework: what they require and what they provide. We reproduce their Isabelle code for completeness, but note that the code in this section was not written by us. The full listings can be found in the Archive of Formal Proofs entry by Blanchette et al. [6].

First, two coinductive datatypes are crucial: a *tree* is finitely branching but can be infinitely deep, while a *stream* has no branching but is decidedly infinite (a list with no end).

```
codatatype 'a tree = Node (root: 'a) (cont: 'a tree fset )
codatatype (sset: 'a) stream =
  SCons (shd: 'a) (stl: 'a stream) (infixr ## 65)
```

A *tree* is a *Node* consisting of a *root* element and a finite set (*fset*) of subtrees, which can be extracted with *cont*. Streams are built with the constructor *SCons* and have a head element (*shd*) and a tail stream (*stl*). The function *sset* takes a stream and returns its set of elements.

Tables 1 and 2 cover the two locales *RuleSystem* and *PersistentRuleSystem* which are central for proving completeness. The locale premises are given above each vertical line and the (important) conclusions are given below. For reference, we include the relevant Isabelle code as well, after each dotted line, but reading this is optional. We need a few explanations to understand the Isabelle code. Locales in Isabelle are specified by giving a name followed by a number of *fixed* constants of given types and a number of *assumptions* about those constants, which we can rely on while working inside the locale. Types in Isabelle/HOL include type variables *'a* and function arrows \Rightarrow . Assumptions can be stated with the meta-universal quantifier \bigwedge and meta-logical implication \Longrightarrow . The notation $\llbracket A; B; \dots \rrbracket \Longrightarrow X$ is shorthand for $A \Longrightarrow B \Longrightarrow \dots \Longrightarrow X$. We can also extend another locale by using the $+$ operation explicitly. In this case, it can be useful to give the type variables descriptive names by using the *for* keyword to explicitly specify the types of the constants. The relation $|\in|$ denotes membership of a finite set. Moreover, *SOME* is Hilbert's choice operator and picks an element satisfying the given predicate, when such an element exists. The expression *the None* is an undefined value, since *the* is only defined on the *Some* constructor. The combinators *Not*, *alw*, *ev* and *holds* implement linear temporal logic operators on streams. For instance, the function *trim* uses *sdrop-while* and *Not* to drop rules from a stream as long as they are not enabled in the current state. The predicate *fair* expresses that all rules occur infinitely often: no matter how far down the stream we go, it is always the case that every rule will eventually occur at some later point in the stream. The corecursive function *mkTree* uses *fimage*, which is simply the image of a function over a finite set; in this case it is used to build the subtrees. Its definition uses *coinductive* since it applies to the codatatype of potentially infinite trees.

The locales in Table 1 require us to prove a number of things about three definitions, where two of them are given in the *RuleSystem-Defs* locale. First, the *eff* relation specifies the effect of applying a rule to a state in our proof search. By (proof) state we mean a sequent, potentially coupled with additional information. The nodes of our proof tree will be proof states in this sense. Second, *rules* is a stream of rules for the prover to attempt to apply. Third, *S* is a set of well formed states (in our case simply the set of all states).

For the *RuleSystem* locale, we must prove two things about these definitions. First, *eff-S*, that the set of well formed states *S* is closed under the *eff* relation on rules from the stream *rules*. Second, *enabled-R*, that no matter the proof state we have reached (in *S*), some rule in *rules* applies. In return we get the function *mkTree* which embodies our prover and a proof, *wf-mkTree*, that the tree produced by this prover is well formed (for fair rule streams). A tree is well formed (*wf*) when its children are well formed and the set of child states is *eff*-related to the node's state and applied rule.

For the *PersistentRuleSystem* locale in Table 2, we must additionally prove the assumption *per*. This essentially states that rules do not interfere with each other: when we apply one rule, other rules that were applicable before are still applicable. In return we get a theorem called *epath-completeness-Saturated*. An escape path (*epath*) is an infinite path in a well formed proof tree. Such a path is saturated (*Saturated*) when any rule which is enabled at some point on the path is eventually applied. Thus, this theorem states a completeness property for the *mkTree* function (on well formed input): either it returns a well formed finite tree or a tree containing a saturated escape path (from which we can build a countermodel).

Table 3 covers the *Soundness* locale used to prove the soundness of resulting proof trees. Here, besides *eff* and *rules*, we must provide a set of models, *structure*, and a satisfaction

Table 1 The *RuleSystem* locale with premises above the line and important conclusions below. Corresponding (abridged) Isabelle code below the dotted line

<i>eff</i>	Effect relation between a rule, a state and a finite set of resulting states.
<i>rules</i>	Stream of rules. The set of these is called <i>R</i> .
<i>S</i>	Set of well formed states.
<i>eff-S</i>	Proof that for any proof state in <i>S</i> and rule in <i>R</i> , the <i>eff</i> -related states are in <i>S</i> .
<i>enabled-R</i>	Proof that for any state in <i>S</i> , some rule in <i>R</i> is <i>enabled</i> , i.e. applies to that state.
<hr/>	
<i>mkTree</i>	A function from a stream of rules and a starting state to a tree of states and rules.
<i>wf-mkTree</i>	Proof that the tree generated by <i>mkTree</i> is well formed wrt. <i>eff</i> when the rule stream is <i>fair</i> (every rule keeps appearing).
<hr/>	
.....	
locale <i>RuleSystem-Defs</i> =	
fixes <i>eff</i> :: 'rule \Rightarrow 'state \Rightarrow 'state fset \Rightarrow bool	
and <i>rules</i> :: 'rule stream	
begin	
abbreviation <i>R</i> \equiv sset <i>rules</i>	
abbreviation <i>effStep</i> <i>step</i> \equiv <i>eff</i> (<i>snd</i> <i>step</i>) (<i>fst</i> <i>step</i>)	
definition <i>enabled</i> <i>r</i> <i>s</i> \equiv \exists <i>sl</i> . <i>eff</i> <i>r</i> <i>s</i> <i>sl</i>	
definition <i>pickEff</i> <i>r</i> <i>s</i> \equiv if <i>enabled</i> <i>r</i> <i>s</i> then (SOME <i>sl</i> . <i>eff</i> <i>r</i> <i>s</i> <i>sl</i>) else the None	
definition <i>trim</i> <i>rs</i> <i>s</i> = <i>sdrop-while</i> (λ <i>r</i> . Not (<i>enabled</i> <i>r</i> <i>s</i>)) <i>rs</i>	
primcorec <i>mkTree</i> where	
<i>root</i> (<i>mkTree</i> <i>rs</i> <i>s</i>) = (<i>s</i> , (<i>shd</i> (<i>trim</i> <i>rs</i> <i>s</i>)))	
<i>cont</i> (<i>mkTree</i> <i>rs</i> <i>s</i>) =	
<i>fimage</i> (<i>mkTree</i> (<i>stl</i> (<i>trim</i> <i>rs</i> <i>s</i>))) (<i>pickEff</i> (<i>shd</i> (<i>trim</i> <i>rs</i> <i>s</i>)) <i>s</i>)	
definition <i>fair</i> <i>rs</i> \equiv sset <i>rs</i> \subseteq <i>R</i> \wedge (\forall <i>r</i> \in <i>R</i> . <i>alw</i> (<i>ev</i> (<i>holds</i> ($(=)$ <i>r</i>))) <i>rs</i>)	
coinductive <i>wf</i> where	
<i>wf</i> : \llbracket <i>snd</i> (<i>root</i> <i>t</i>) \in <i>R</i> ; <i>effStep</i> (<i>root</i> <i>t</i>) (<i>fimage</i> (<i>fst</i> <i>o</i> <i>root</i>) (<i>cont</i> <i>t</i>));	
\wedge <i>t'</i> . <i>t'</i> \in <i>cont</i> <i>t</i> \implies <i>wf</i> <i>t'</i> $\rrbracket \implies$ <i>wf</i> <i>t</i>	
end	
locale <i>RuleSystem</i> = <i>RuleSystem-Defs</i> <i>eff</i> <i>rules</i>	
for <i>eff</i> :: 'rule \Rightarrow 'state \Rightarrow 'state fset \Rightarrow bool and <i>rules</i> :: 'rule stream +	
fixes <i>S</i> :: 'state set	
assumes <i>eff-S</i> : \bigwedge <i>s</i> <i>r</i> <i>sl</i> <i>s'</i> . \llbracket <i>s</i> \in <i>S</i> ; <i>r</i> \in <i>R</i> ; <i>eff</i> <i>r</i> <i>s</i> <i>sl</i> ; <i>s'</i> \in <i>sl</i> $\rrbracket \implies$ <i>s'</i> \in <i>S</i>	
and <i>enabled-R</i> : \bigwedge <i>s</i> . <i>s</i> \in <i>S</i> \implies \exists <i>r</i> \in <i>R</i> . \exists <i>sl</i> . <i>eff</i> <i>r</i> <i>s</i> <i>sl</i>	
begin	
lemma <i>wf-mkTree</i> :	
assumes <i>s</i> : <i>s</i> \in <i>S</i> and <i>fair</i> <i>rs</i>	
shows <i>wf</i> (<i>mkTree</i> <i>rs</i> <i>s</i>)	
end	

Table 2 The *PersistentRuleSystem* locale which extends *RuleSystem* from Table 1

<i>per</i>	Proof that if a rule r in R is enabled in a well formed state s and s' is <i>eff</i> -related to s by a rule r' in R distinct from r , then r is enabled in s' .
<i>epath-completeness-Saturated</i>	Proof that for any well formed state s , there exists either a well formed finite tree with s as root or a saturated escape path with s as root.
.....	
Inherited definitions from locale <i>RuleSystem-Defs</i> :	
abbreviation $enabledAtStep\ r\ step \equiv enabled\ r\ (fst\ step)$	
abbreviation $takenAtStep\ r\ step \equiv snd\ step = r$	
definition $saturated\ r \equiv$ $alw\ (holds\ (enabledAtStep\ r)\ impl\ ev\ (holds\ (takenAtStep\ r)))$	
definition $Saturated\ steps \equiv \forall\ r \in R. saturated\ r\ steps$	
coinductive <i>epath</i> where	
$epath: \llbracket snd\ (shd\ steps) \in R; fst\ (shd\ (stl\ steps)) \mid \in \mid sl; effStep\ (shd\ steps)\ sl;$ $epath\ (stl\ steps) \rrbracket \implies epath\ steps$	
Inherited definitions from locale <i>RuleSystem</i> :	
definition $per\ r \equiv \forall\ s\ r1\ sl'\ s'. s \in S \wedge enabled\ r\ s \wedge r1 \in R - \{r\} \wedge$ $eff\ r1\ s\ sl' \wedge s' \mid \in \mid sl' \longrightarrow enabled\ r\ s'$	
The definition of the new locale:	
locale <i>PersistentRuleSystem</i> = <i>RuleSystem</i> <i>eff</i> rules S	
for $eff :: 'rule \Rightarrow 'state \Rightarrow 'state\ fset \Rightarrow bool$ and $rules :: 'rule\ stream$ and $S +$	
assumes $per: \bigwedge\ r. r \in R \implies per\ r$	
begin	
theorem <i>epath-completeness-Saturated</i> :	
assumes $s \in S$	
shows $(\exists\ t. fst\ (root\ t) = s \wedge wf\ t \wedge tfinite\ t) \vee$ $(\exists\ steps. fst\ (shd\ steps) = s \wedge epath\ steps \wedge Saturated\ steps)$	
end	

predicate, *sat*, on sequents and models. The locale then turns a local soundness proof, *local-soundness*, that validity of a sequent follows from validity of its children, into a global result, *soundness*, that any finite, well formed tree has a valid root.

Finally, to generate code we need to instantiate the locale *RuleSystem-Code* in Table 4, where *eff* must now be a deterministic relation (i.e. a function) and *rules* is as before. In return we get an executable version of *mkTree* above, called *i.mkTree* and defined from the code lemmas specified in the table.

RuleSystem-Code provides no guarantees on its own, but we use the same underlying function in all four locales. We export this function to Haskell using Isabelle’s (only partly verified) code generation, code lemmas and a few (unverified) custom code-printing facilities. This step moves us from a verified prover inside Isabelle to a prover in Haskell which is based on a verified prover, but which is not itself verified.

Table 3 The *Soundness* locale

<i>eff</i> , <i>rules</i>	As in Table 1 but states are now called sequents.
<i>structure</i>	Set of models.
<i>sat</i>	Satisfaction predicate on sequents and models.
<i>local-soundness</i>	Proof that the validity of a sequent (as given by <i>sat</i> and <i>structure</i>) follows from the validity of its children (as given by <i>eff</i> and <i>rules</i>).
<i>soundness</i>	Proof that any finite, well formed tree has a valid root.

.....

```

locale Soundness = RuleSystem-Defs eff rules for
  eff :: 'rule  $\Rightarrow$  'sequent  $\Rightarrow$  'sequent fset  $\Rightarrow$  bool
  and rules :: 'rule stream +
fixes structure :: 'structure set
  and sat :: 'structure  $\Rightarrow$  'sequent  $\Rightarrow$  bool
assumes local-soundness:  $\bigwedge r s sl.$ 
   $\llbracket r \in R; \text{eff } r s sl; \bigwedge s'. s' \in | sl \implies \forall S \in \text{structure. sat } S s' \rrbracket \implies$ 
   $\forall S \in \text{structure. sat } S s$ 
begin
abbreviation ssat s  $\equiv \forall S \in \text{structure. sat } S s$ 
theorem soundness:
  assumes f: tfinite t and w: wf t
  shows ssat (fst (root t))
end

```

Table 4 The *RuleSystem-Code* locale

<i>eff</i>	Effect <i>function</i> from a rule and a state to a finite set of resulting states.
<i>rules</i>	Stream of rules.
<i>i.mkTree</i>	Executable version of the <i>mkTree</i> function.

.....

```

locale RuleSystem-Code =
  fixes eff' :: 'rule  $\Rightarrow$  'state  $\Rightarrow$  'state fset option
  and rules :: 'rule stream
begin
lemma enabled r s  $\longleftrightarrow \text{eff}' r s \neq \text{None}$ 
lemma trim rs s = sdrop-while ( $\lambda r. \text{Not } (\text{enabled } r s)$ ) rs
lemma pickEff r s = the (eff' r s)
lemma mkTree rs s =
  (case trim rs s of SCons r s'  $\Rightarrow$  Node (s, r) (fimage (mkTree s') (pickEff r s)))
end

```

3 Prover

In this section we explain the design of the proof search procedure driving our prover. The procedure does not use the proof system of SeCaV directly, but introduces a new set of similar proof rules that apply to entire sequents at once. That is, the prover rule corresponding to ALPHADIS breaks down all formulas $Dis\ p\ q$ in the sequent, not just the first one. This obviates the need for the structural EXT rule, which is therefore not present. It also removes the need for an explicit IDLE rule, needed by Blanchette et al. [11] to prove that there always exists an enabled rule, since we can just let all rules be enabled at all times; sometimes they will simply not have an effect. In general, these *multi-rules* simplify the prover and its completeness proof by removing all concerns about the structure of sequents. If we need to apply a rule to a formula to find a proof, then the multi-rule will apply to that formula immediately, no matter where in the sequent it is. Additionally, we remove the BASIC rule and let the prover close proof branches implicitly.

Before we can define what the rules do, we need a few auxiliary definitions. The functions *listFunTm* and *listFunTms* collect the function names that occur in a term and a list of terms, respectively:

```
primrec listFunTm :: tm  $\Rightarrow$  nat list and listFunTms :: tm list  $\Rightarrow$  nat list where
  listFunTm (Fun n ts) = n # listFunTms ts
| listFunTm (Var n) = []
| listFunTms [] = []
| listFunTms (t # ts) = listFunTm t @ listFunTms ts
```

This is used by *generateNew* to generate a function name that is fresh to a given list of terms:

```
definition generateNew :: tm list  $\Rightarrow$  nat where
  generateNew ts  $\equiv$  1 + foldr max (listFunTms ts) 0
```

The functions *subtermTm* and *subtermFm* compute the list of terms occurring in a term and a formula, respectively:

```
primrec subtermTm :: tm  $\Rightarrow$  tm list where
  subtermTm (Fun n ts) = Fun n ts # remdups (concat (map subtermTm ts))
| subtermTm (Var n) = [Var n]

primrec subtermFm :: fm  $\Rightarrow$  tm list where
  subtermFm (Pre - ts) = concat (map subtermTm ts)
| subtermFm (Imp p q) = subtermFm p @ subtermFm q
| subtermFm (Dis p q) = subtermFm p @ subtermFm q
| subtermFm (Con p q) = subtermFm p @ subtermFm q
| subtermFm (Exi p) = subtermFm p
| subtermFm (Uni p) = subtermFm p
| subtermFm (Neg p) = subtermFm p
```

This is used by *subtermFms* to compute the list of all terms in a list of formulas (i.e. a sequent):

```
abbreviation subtermFms z  $\equiv$  concat (map subtermFm z)
```

We define *subterms* as the list of all terms in a sequent, except that the list contains exactly *Fun 0 []* when it would otherwise be empty. This ensures that we always have some term to instantiate γ -formulas with:

```
definition subterms :: sequent  $\Rightarrow$  tm list where
  subterms z  $\equiv$  case remdups (subtermFms z) of
    []  $\Rightarrow$  [Fun 0 []]
  | ts  $\Rightarrow$  ts
```


The function *sub* (defined in Fig. 4) implements substitution in a standard way using de Bruijn indices. See the formalization [16] or the original SeCaV work [18] for details. The function *branchDone* computes whether a sequent is an axiom, i.e. whether the sequent contains both a formula and its negation:

```
fun branchDone :: sequent  $\Rightarrow$  bool where
  branchDone [] = False
  | branchDone (Neg p # z) = (p  $\in$  set z  $\vee$  Neg (Neg p)  $\in$  set z  $\vee$  branchDone z)
  | branchDone (p # z) = (Neg p  $\in$  set z  $\vee$  branchDone z)
```

The prover uses this to determine when a branch of the proof tree is proven and can be closed. The disjunct $\text{Neg } (\text{Neg } p) \in \text{set } z$ is not necessary for the prover, but makes the proofs easier later on.

We first define which “parts” of a single formula *f* must be proven for a rule *r* to apply:

```
definition parts :: tm list  $\Rightarrow$  rule  $\Rightarrow$  fm  $\Rightarrow$  fm list list where
  parts A r f = (case (r, f) of
    (NegNeg, Neg (Neg p))  $\Rightarrow$  [[p]]
  | (AlphaImp, Imp p q)  $\Rightarrow$  [[Neg p, q]]
  | (AlphaDis, Dis p q)  $\Rightarrow$  [[p, q]]
  | (AlphaCon, Neg (Con p q))  $\Rightarrow$  [[Neg p, Neg q]]
  | (BetaImp, Neg (Imp p q))  $\Rightarrow$  [[p], [Neg q]]
  | (BetaDis, Neg (Dis p q))  $\Rightarrow$  [[Neg p], [Neg q]]
  | (BetaCon, Con p q)  $\Rightarrow$  [[p], [q]]
  | (DeltaExi, Neg (Exi p))  $\Rightarrow$  [[Neg (sub 0 (Fun (generateNew A) []) p)]]
  | (DeltaUni, Uni p)  $\Rightarrow$  [[sub 0 (Fun (generateNew A) []) p]]
  | (GammaExi, Exi p)  $\Rightarrow$  [Exi p # map ( $\lambda$ t. sub 0 t p) A]
  | (GammaUni, Neg (Uni p))  $\Rightarrow$  [Neg (Uni p) # map ( $\lambda$ t. Neg (sub 0 t p)) A]
  | -  $\Rightarrow$  [[f]])
```

The *parts* of a formula under a rule is a list of lists of formulas with an implicit conjunction between lists and disjunction between inner formulas. For instance, the function states that for *AlphaDis* to prove *Dis p q*, we must prove either *p* or *q*. The definition takes a parameter *A*, which should be a list of terms present on the proof branch. For δ -rules, a function which does not appear in *A* is generated to instantiate the quantifier (ensuring soundness), and for γ -rules, the quantifier is instantiated with every term in *A* (ensuring completeness). Note that if the rule and formula do not match, the result simply contains the original formula. This means that rules are always enabled, but that they do nothing to most formulas.

To construct a proof tree, we need a function that computes the result of applying a rule to (all formulas in) a sequent. This is done by the following function (*@* appends two lists and *remdups* removes duplicates):

```
primrec children :: tm list  $\Rightarrow$  rule  $\Rightarrow$  sequent  $\Rightarrow$  sequent list where
  children - - [] = [[]]
  | children A r (p # z) =
    (let hs = parts A r p; A' = remdups (A @ subtermFms (concat hs))
     in list-prod hs (children A' r z))
```

It first computes the effect of applying the rule to the first formula in the sequent (using the definition *parts*) and gives a name to the updated list of terms in the sequent (since δ - and γ -rules may introduce new terms). The function then goes through the rest of the sequent recursively, combining the generated child branches with the function *list-prod*:

```
primrec list-prod :: 'a list list  $\Rightarrow$  'a list list  $\Rightarrow$  'a list list where
  list-prod - [] = []
  | list-prod hs (t # ts) = map ( $\lambda$ h. h @ t) hs @ list-prod hs ts
```

The type variable $'a$ in the type signature means that the function works on lists of lists containing any type of elements. The function *list-prod* behaves in the following way (similar to the Cartesian product):

$$\text{set } (\text{list-prod } hs \ ts) = \{h @ t \mid h \ t. h \in \text{set } hs \wedge t \in \text{set } ts\}$$

For β -rules, the end result is a list of 2^n child branches, where n is the number of β -formulas in the sequent. These branches are ordered such that they correspond to the branches one would have obtained by applying the corresponding SeCaV β -rule n times. For all other rules, the end result is a single child branch. The parameter A to *children* should again be a list of terms present on the proof branch. We should be clear that *children* does not apply rules recursively to sub-formulas, but only to the “top layer.” If the application of a rule reveals a formula that this rule applies to again, this formula is left alone and is only considered the next time *children* is applied to the sequent with that rule. For example, the result of calling *children* with the rule ALPHADIS and the sequent containing only the formula Dis (Dis $p \ q$) r is Dis $p \ q, r$ and not p, q, r .

The prover needs to ensure that bound variables are instantiated with all terms on the current branch when a γ -rule is applied. For this reason, we define the *state* in a proof tree node to be a pair consisting of a list of terms appearing on the branch and a sequent. The list of terms will be used to instantiate the parameter A in the definitions above.

We are now ready to define the effect of applying a proof rule to a proof state:

```

primrec effect :: rule  $\Rightarrow$  state  $\Rightarrow$  state fset where
  effect r (A, z) =
    (if branchDone z then {} else
     fimage ( $\lambda z'$ . (remdups (A @ subterms z @ subterms z'), z'))
     (fset-of-list (children (remdups (A @ subtermFms z)) r z)))

```

To fit the types of the framework, the function returns a finite set (*fset*) instead of a list, and the function *fimage* is used to compute the image of a finite set under a given function. If the sequent is an axiom, the branch is proven, and the function returns an empty set of child nodes (denoted by $\{\}$ for finite sets), closing the branch. Otherwise, the function converts the result of the *children* function to a finite set, and adds any new terms to the list of terms in each child node.

Having defined what rules do, we now need a *stream* of them (*rules* in Table 1). We, somewhat arbitrarily, define a list of rules in the order $\alpha, \delta, \beta, \gamma$ and cycle the list to obtain a stream. For efficiency, we could run, say, all α - and δ -rules to completion before branching with the β -rules, but this cannot be encoded in the simple stream of rules without further machinery: one could imagine having larger “meta-rules” corresponding to groups of SeCaV rules. This would give a notion of “phases” where we would first run all the rules in one group, then all the rules in the next group in the stream, etc. For simplicity (see Sect. 6.4) we apply single rules in a fixed order. This also trivially ensures fairness.

3.1 Applying the Framework

We are now ready to apply the abstract completeness framework to obtain the actual proof search procedure (cf. Sect. 2.2). First, we define a relational version of the *effect* of a rule, called *eff*. To use the framework, we need to prove three properties: that the set of well formed proof states is closed under *eff* (*eff-S*), that it is always possible to apply some rule (*enabled-R*) and that the rules that can be applied are still possible to apply after applying other rules (*per*). We do not need to restrict the set of well formed proof states, so the first

property is trivial. Since all of our rules can always be applied (they simply do nothing if they do not match the sequent), the other two properties are also trivial. We can thus instantiate the framework with our effect relation and stream of rules. This allows us to define the prover using the *mkTree* function from the framework:

definition $secavProver \equiv mkTree\ rules$

This function takes a list of terms and a sequent, and applies the rules in the stream in order to build a proof tree with the given sequent at the root, using our *eff* relation to determine the children of each node. The list of terms is used to collect the terms that occur in the sequents on each branch and should initially be empty (in the exported prover, the function is wrapped in another function to ensure that the list of terms is empty).

We call the sequent at the root of this proof tree the *root sequent*:

abbreviation $rootSequent\ t \equiv snd\ (fst\ (root\ t))$

3.2 Making the Prover Executable

To actually make the prover executable, we need to specify that the stream of rules should be lazily evaluated, or the prover will never terminate. We do this by lazifying the *stream* type using the *code-lazy-type* command [31]. Additionally, we need to define the prover using the code interpretation of the framework to enable computation of some parts of the framework (cf. Table 4). After telling Isabelle how to translate operations on the *option* type to the *Maybe* type, this also allows us to export the prover to Haskell code.

We have implemented a few Haskell modules to drive the exported prover and translate found proofs into the proof system of SeCaV. These modules are not formally verified, but the proofs generated in this manner can be verified by Isabelle. We have written an automated test suite that tests the unverified code for soundness and completeness by applying the prover to a number of valid formulas, then calling Isabelle to verify the generated proofs, and by applying the prover to a number of invalid formulas and confirming that it does not generate a proof (within 10 seconds). While these tests do not give us absolute certainty that the exported code and the hand-written Haskell modules are correct, they provide a reasonable amount of certainty when combined with the formal proofs of correctness of the proof search procedure within Isabelle.

4 Soundness

We use the abstract soundness framework (cf. Sect. 2.2) to prove that any sequent with a well formed and finite proof tree can be proved in the proof system of SeCaV. It follows from the soundness of SeCaV that such sequents for which the prover terminates are semantically valid.

In a well formed proof tree, the sequent in every node can be derived from the sequents in its child nodes. We will need an intermediate lemma stating that the proof trees produced using the *children* function are well formed. For this to hold, it must be the case that the terms occurring in the sequent we are trying to derive are contained in the list of existing terms given to the *children* function such that the δ -rules are used with constants which are actually fresh. To state this condition, we need the following functions, which are similar to *listFunTms* and *subtermFms* but compute sets instead of lists:

primrec $paramst :: tm \Rightarrow nat\ set$ **and** $paramsts :: tm\ list \Rightarrow nat\ set$ **where**

User-Friendly Formal Methods

$$\begin{aligned} \text{paramst } (\text{Var } n) &= \{ \} \quad | \\ \text{paramst } (\text{Fun } a \text{ ts}) &= \{a\} \cup \text{paramsts } \text{ts} \quad | \\ \text{paramsts } [] &= \{ \} \quad | \\ \text{paramsts } (t \# \text{ts}) &= (\text{paramst } t \cup \text{paramsts } \text{ts}) \end{aligned}$$

primrec $\text{params} :: \text{fm} \Rightarrow \text{nat set}$ **where**

$$\begin{aligned} \text{params } (\text{Pre } b \text{ ts}) &= \text{paramsts } \text{ts} \quad | \\ \text{params } (\text{Imp } p \ q) &= \text{params } p \cup \text{params } q \quad | \\ \text{params } (\text{Dis } p \ q) &= \text{params } p \cup \text{params } q \quad | \\ \text{params } (\text{Con } p \ q) &= \text{params } p \cup \text{params } q \quad | \\ \text{params } (\text{Exi } p) &= \text{params } p \quad | \\ \text{params } (\text{Uni } p) &= \text{params } p \quad | \\ \text{params } (\text{Neg } p) &= \text{params } p \end{aligned}$$

abbreviation $\text{paramss } z \equiv \bigcup p \in \text{set } z. \text{params } p$

The following lemma then comprises the core of the result:

Lemma 1 *If for all sequents z' in children $A \ r \ z$, we can derive $\Vdash \text{pre} @ z'$ and the term list A contains all parameters of pre and z , then we can derive $\Vdash \text{pre} @ z$ itself:*

assumes $\forall z' \in \text{set } (\text{children } A \ r \ z). (\Vdash \text{pre} @ z')$
and $\text{paramss } (\text{pre} @ z) \subseteq \text{paramsts } A$
shows $\Vdash \text{pre} @ z$

Proof By induction on z for arbitrary pre and A .

For the empty sequent, we can immediately derive $\Vdash \text{pre}$ from the assumption and the definition of *children*.

For the non-empty sequent with formula p as head and z as tail we have the following induction hypothesis (for any pre and A):

then have ih: $\forall z' \in \text{set } (\text{children } A \ r \ z). (\Vdash \text{pre} @ z') \implies (\Vdash \text{pre} @ z)$
if $\text{paramss } (\text{pre} @ z) \subseteq \text{paramsts } A$ **for** $\text{pre } A$

We abbreviate the term list that the prover actually recurses on as $?A$. From the first assumption and the definition of *list-prod* we then have:

$$\forall \text{hs} \in \text{set } (\text{parts } A \ r \ p). \forall \text{ts} \in \text{set } (\text{children } ?A \ r \ z). (\Vdash \text{pre} @ \text{hs} @ \text{ts}) \quad (*)$$

The proof continues by examining the possible cases for *parts*.

Take first the case where $r = \text{AlphaDis}$ and $p = \text{Dis } q \ r$. Then (*) states that we can derive $\Vdash \text{pre} @ q \# r \# z'$ for all z' in *children* $?A \ r \ z$. We apply the induction hypothesis at pre extended with q and r , which is allowed since they are subformulas of p . We then get the derivation $\Vdash \text{pre} @ q \# r \# z$. By the EXT and ALPHADIS rules from SeCaV we obtain the desired derivation $\Vdash \text{pre} @ \text{Dis } q \ r \# z$.

The remaining α - and β -cases are similar. In the δ -cases we prove that the constant used by the prover is new to the sequent, as required by the SeCaV δ -rules. To apply the induction hypothesis, we need that the constant is already included in our term list $?A$, but this is guaranteed by constructing $?A$ via *parts*.

In the γ -cases we get a derivation that includes both the γ -formula and all instantiations of it using terms from the list A . Here we induct on A to generalize each instantiation into the corresponding γ -formula and use EXT to contract this γ -formula with the existing occurrence.

When *parts* $A \ r \ p$ returns p , the thesis holds from (*) and the induction hypothesis. \square

We only need pre in the above lemma to make the induction hypothesis strong enough for the proof, so we can instantiate it afterwards.

Corollary 1 (*Proof tree to SeCaV*) *We derive a sequent from derivations of its children:*

assumes $\forall z' \in \text{set}(\text{children } A \ r \ z). (\Vdash z')$ **and** $\text{paramss } z \subseteq \text{paramsts } A$
shows $\Vdash z$

We obtain the following soundness theorem from the abstract soundness framework. Note that the derivation in SeCaV follows the steps in the well-formed proof tree, which means that the derivation corresponds exactly to the generated proof certificate.

Theorem 1 (Prover soundness wrt. *SeCaV*) *The root sequent of any finite, well formed proof tree has a derivation in SeCaV:*

assumes $t \text{ finite } t$ **and** wft
shows $\Vdash \text{rootSequent } t$

5 Completeness

The completeness proof is heavily based on the abstract completeness framework. As noted in Sect. 2.2, however, the framework only helps us with part of the proof. First, we duplicate the output of Table 2, since the *mkTree* function is unhelpfully abstracted away by an existential quantifier. This could easily be changed in the framework and should be considered for the next release.

Lemma 2 (Prover cases) *The proof tree generated by the prover is either finite and well formed or there exists a saturated escape path with our initial state as root:*

defines $t \equiv \text{secavProver } (A, z)$
shows $(fst(\text{root } t) = (A, z) \wedge wft \wedge t \text{ finite } t) \vee$
 $(\exists \text{ steps. } fst(\text{shd steps}) = (A, z) \wedge \text{epath steps} \wedge \text{Saturated steps})$

In the first case, the sequent has a proof (cf. Sect. 4). In the second case, we need to build a countermodel from the saturated escape path to contradict the validity of the sequent. The rest of this section does exactly that. Inspired by Ben-Ari [2] and Ridge and Margetson [41], we start off by giving a definition of Hintikka sets over a restricted set of terms (Sect. 5.1). We show that the set of formulas on saturated escape paths fulfills all Hintikka requirements when we take the set of terms to be the terms on the path (Sect. 5.2). We then define a countermodel for any formula in such a set using a new semantics that bounds quantifiers by an explicit set rather than by types alone (Sect. 5.3). Finally we tie these results together to show that the prover terminates for all sequents that are valid under our new semantics (Sect. 5.4). In Sect. 6.1 we use existing results to prove completeness of the prover wrt. the SeCaV semantics.

5.1 Hintikka

First, by the *terms* of a set of formulas H we mean all the subterms of formulas in H , unless there are none, in which case we mean a designated singleton set:

definition
 $\text{terms } H \equiv \text{if } (\bigcup p \in H. \text{set}(\text{subtermFm } p)) = \{\} \text{ then } \{\text{Fun } 0 \ []\}$
 $\text{else } (\bigcup p \in H. \text{set}(\text{subtermFm } p))$

This set contains an arbitrary (but fixed) constant, $\text{Fun } 0 \ []$, when H itself contains no terms. Otherwise it contains all subterms of all formulas in H . This mirrors the definition of *subterms* in Sect. 3.

```

locale Hintikka =
  fixes H :: fm set
  assumes
    Basic: Pre n ts ∈ H ⇒ Neg (Pre n ts) ∉ H and
    AlphaDis: Dis p q ∈ H ⇒ p ∈ H ∧ q ∈ H and
    AlphaImp: Imp p q ∈ H ⇒ Neg p ∈ H ∧ q ∈ H and
    AlphaCon: Neg (Con p q) ∈ H ⇒ Neg p ∈ H ∧ Neg q ∈ H and
    BetaCon: Con p q ∈ H ⇒ p ∈ H ∨ q ∈ H and
    BetaImp: Neg (Imp p q) ∈ H ⇒ p ∈ H ∨ Neg q ∈ H and
    BetaDis: Neg (Dis p q) ∈ H ⇒ Neg p ∈ H ∨ Neg q ∈ H and
    GammaExi: Exi p ∈ H ⇒ ∀ t ∈ terms H. sub 0 t p ∈ H and
    GammaUni: Neg (Uni p) ∈ H ⇒ ∀ t ∈ terms H. Neg (sub 0 t p) ∈ H and
    DeltaUni: Uni p ∈ H ⇒ ∃ t ∈ terms H. sub 0 t p ∈ H and
    DeltaExi: Neg (Exi p) ∈ H ⇒ ∃ t ∈ terms H. Neg (sub 0 t p) ∈ H and
    Neg: Neg (Neg p) ∈ H ⇒ p ∈ H

```

Fig. 5 Requirements for a set of formulas H to be a Hintikka set

Figure 5 contains a definition of a Hintikka set H . Here, we use a locale slightly differently to the previous ones, in that we specify no conclusions, only premises: the formula set H and the requirements *Basic*, *AlphaDis*, etc. This use simply allows us to assume *Hintikka* H in a theorem and know that the set H then fulfills the stated requirements. Similarly, we can prove that a set H is *Hintikka* by proving that it fulfills the requirements. It is important to note that in the γ - and δ -cases, the quantifiers only range over the *terms* of H .

5.2 Saturated Escape Paths are Hintikka

The following definition forgets all structure of a path and reduces it to a set of formulas:

definition $tree\text{-}fms\ steps \equiv \bigcup ss \in sset\ steps.\ set\ (pseq\ ss)$

The function *pseq* extracts the sequent from each step.

Given a saturated escape path *steps*, we want to prove that *tree-fms steps* is a Hintikka set. For instance, if *Dis p q* appears on the path, then both *p* and *q* should too. The prover is designed to make this property of its proof trees as evident as possible: formulas unaffected by a given rule are easily shown to be preserved by the application of that rule and any rule immediately applies to all its affected formulas, regardless of their position in the sequent.

We will need a number of intermediate results.

5.2.1 Unaffected Formulas

We define the predicate *affects* to hold for a rule and a formula when that rule does not preserve the formula (thus no rule *affects* a γ -formula, since the γ -rules of the prover, unlike those of SeCaV, preserve the original formula). For instance, *affects AlphaDis (Dis p q)* holds while *affects BetaCon (Dis p q)* does not.

We then prove the following key preservation lemma:

Lemma 3 (*effect preserves unaffected formulas*)

Assume formula p occurs in sequent z and the rule r does not affect p. Then p also occurs in all children of z as given by effect:

assumes $p \in \text{set } z$ **and** $\neg \text{affects } r \ p$ **and** $(B, z') \mid \in \mid \text{effect } r \ (A, z)$
shows $p \in \text{set } z'$

Proof The function *parts* preserves unaffected formulas (proof by cases) so *children* does as well (proof by induction on the sequent) and thus *effect* does too. \square

We lift this to escape paths:

Lemma 4 (Escape paths preserve unaffected formulas) *Assume formula p occurs in some sequent at the head of an escape path which consists of a prefix, pre , where none of the rules affect p , and a suffix, suf . Then p occurs at the head of suf (the operator $@-$ prepends a list to a stream):*

assumes $p \in \text{set } (pseq \ (shd \ steps))$ **and** $epath \ steps$ **and** $steps = pre \ @- \ suf$
and $list\text{-all} \ (not \ (\lambda step. \text{affects} \ (snd \ step) \ p)) \ pre$
shows $p \in \text{set } (pseq \ (shd \ suf))$

Next, notice the following property of streams:

Lemma 5 (Eventual prefix) *When a property P eventually holds of a stream, then the stream is comprised of a prefix of n (possibly zero) elements for which P does not hold and then a suffix that starts with an element for which P does hold:*

assumes $ev \ (holds \ P) \ xs$
shows $\exists n. list\text{-all} \ (not \ P) \ (stake \ n \ xs) \wedge holds \ P \ (sdrop \ n \ xs)$

Saturation states that a rule is eventually applied and Lemmas 4 and 5 combine to state that any affected formulas are preserved until then.

5.2.2 Affected Formulas

Knowing that formulas are preserved as desired, we need to know that they are broken down as desired. The following lemma (proof omitted here) states this in general via *parts*:

Lemma 6 (Parts in effect) *For any formula p in a sequent z , the effect of rule r on z includes some part of r 's effect on p :*

assumes $p \in \text{set } z$ **and** $(B, z') \mid \in \mid \text{effect } r \ (A, z)$
shows $\exists C \ xs. set \ A \subseteq set \ C \wedge xs \in set \ (parts \ C \ r \ p) \wedge set \ xs \subseteq set \ z'$

This is easier to understand when we specialize the rule and the formula:

Corollary 2 *Effect of the NegNeg rule on a double-negated formula p :*

shows $Neg \ (Neg \ p) \in \text{set } z \implies (B, z') \mid \in \mid \text{effect } NegNeg \ (A, z) \implies p \in \text{set } z'$

5.2.3 Hintikka Requirements

We then need to prove the following:

Theorem 2 (Hintikka escape paths) *Saturated escape paths fulfill all Hintikka requirements:*

assumes $epath \ steps$ **and** $Saturated \ steps$
shows $Hintikka \ (tree\text{-fms} \ steps)$

Proof This boils down to proving each requirement of Fig. 5. We give a couple of examples and refer to the formalization for the full details.

For *Basic*, assume towards a contradiction that both a predicate and its negation appear on the branch. By preservation of formulas (Lemma 4), both appear in the same sequent at some point. But then *branchDone* holds for that sequent, so it has no children and the branch would terminate. This contradicts that escape paths are infinite, so *Basic* must hold.

For *AlphaDis*, assume that $Dis\ p\ q$ appears on the branch. Then it appears at some step n . By saturation of the escape path, *AlphaDis* is eventually applied at some (earliest) step $n + k$. By Lemma 4, $Dis\ p\ q$ is preserved until then. So by the effect of rule *AlphaDis*, both p and q appear at step $n + k + 1$. The cases for the β - and δ -requirements are very similar.

For *GammaExi* assume that $Exi\ p$ occurs at step n . We need to show that it is instantiated with all terms that (eventually) appear on the branch. Fix an arbitrary such term t . There must be some step m where t appears in a sequent. Thus at every step after m , term t appears in the term list which is part of the proof state. By saturation, at some step greater than $n + m + 1$, rule *GammaExi* is applied. The formula $Exi\ p$ is preserved until this stage (Lemma 4) and the term list only grows, so t is too. Thus, at the next step, $sub\ 0\ t\ p$ occurs on the branch as desired. \square

5.3 Countermodel

We need to build a countermodel for any formula in a Hintikka set to contradict the validity of any formula on a saturated escape path. We do this in the usual term model with a (bounded) Herbrand interpretation. Unfortunately, we cannot build a countermodel in the original semantics where the universe is specified as a type, since we cannot form the type of terms in a given Hintikka set (we cannot use the *typedef* mechanism of Isabelle to parameterize the universe on the terms in the Hintikka set since the set depends on the formula we are attempting to prove). Instead, we introduce a custom bounded semantics.

5.3.1 Bounded Semantics

The bounded semantics is exactly like the usual semantics (cf. Fig. 2) except for an extra argument u , standing for the universe, which bounds the range of the quantifiers in the following cases:

$$\begin{aligned} | \quad usemantics\ u\ e\ f\ g\ (Exi\ p) &= (\exists x \in u. usemantics\ u\ (SeCaV.shift\ e\ 0\ x)\ f\ g\ p) \\ | \quad usemantics\ u\ e\ f\ g\ (Uni\ p) &= (\forall x \in u. usemantics\ u\ (SeCaV.shift\ e\ 0\ x)\ f\ g\ p) \end{aligned}$$

This leads to the following natural requirements on environments e and function denotations f , namely that they must stay inside u :

definition $is-env\ u\ e \equiv \forall n. e\ n \in u$

definition $is-fdenot\ u\ f \equiv \forall i\ l. list-all\ (\lambda x. x \in u)\ l \longrightarrow f\ i\ l \in u$

In general, we only consider environments and function denotations that satisfy these requirements and call them (and any model based on them) *well formed*. When $u = UNIV$ (the universal set), the quantifiers are not actually bounded and the two semantics coincide.

The SeCaV proof system (cf. Fig. 3) is sound for the bounded semantics too.

Theorem 3 (*SeCaV is sound for the bounded semantics*) *Given a SeCaV derivation of sequent z and a well formed model, some formula p in z is satisfied in that model:*

assumes $\Vdash z$ **and** $is\text{-}env\ u\ e$ **and** $is\text{-}fdenot\ u\ f$
shows $\exists p \in set\ z. usemantics\ u\ e\ f\ g\ p$

Proof The proof closely resembles the original soundness proof (cf. [18]). \square

We abbreviate validity of a sequent in the bounded semantics as *uvalid*:

abbreviation $uvalid\ z \equiv \forall u (e :: nat \Rightarrow tm) f\ g. is\text{-}env\ u\ e \longrightarrow is\text{-}fdenot\ u\ f$
 $\longrightarrow (\exists p \in set\ z. usemantics\ u\ e\ f\ g\ p)$

Namely, for all universes and well formed models, some formula in the sequent is satisfied in the bounded semantics at that universe by that model.

5.3.2 Model Construction

Our countermodel is given by a bounded Herbrand interpretation where terms are interpreted as themselves when they appear in the universe *terms H* and as an arbitrary term otherwise.

Definition 1 (Countermodel induced by Hintikka set *S*)

We abbreviate the model as *M S*:

abbreviation $E\ S\ n \equiv if\ Var\ n \in terms\ S\ then\ Var\ n\ else\ SOME\ t. t \in terms\ S$
abbreviation $F\ S\ i\ l \equiv if\ Fun\ i\ l \in terms\ S\ then\ Fun\ i\ l\ else\ SOME\ t. t \in terms\ S$
abbreviation $G\ S\ n\ ts \equiv Neg\ (Pre\ n\ ts) \in S$
abbreviation $M\ S \equiv usemantics\ (terms\ S)\ (E\ S)\ (F\ S)\ (G\ S)$

The definition of *G* is what makes this a countermodel rather than a model: a predicate is satisfied exactly when its negation is present in the Hintikka set.

Importantly, these definitions are *well formed*:

Lemma 7 (Well formed countermodel)

Definition 1 is well formed:

shows $is\text{-}env\ (terms\ S)\ (E\ S)$
shows $is\text{-}fdenot\ (terms\ S)\ (F\ S)$

Proof By the construction of *E* and *F* and the nonemptiness of *terms S*. \square

Theorem 4 (Model existence)

The given model falsifies any formula *p* in Hintikka set *S*:

assumes *Hintikka S*
shows $(p \in S \longrightarrow \neg M\ S\ p) \wedge (Neg\ p \in S \longrightarrow M\ S\ p)$

Proof By induction on the size of the formula *p* (substitution instances are smaller than the quantified formulas they arise from since Isabelle's notion of size is the constructor depth of the formula). The second part of the thesis is needed when the Hintikka requirements concern negated formulas. We show a few cases here and refer to the formalization for the full details. The cases omitted here are similar to those shown.

Assume $p = Pre\ n\ ts$ occurs in *S*. We need to show that the given model falsifies *p*. Since *terms S* is downwards closed by construction, *ts* is interpreted as itself by the bounded Herbrand interpretation. Moreover, by the *Basic* requirement, we know that *Neg p* is not in *S* and is therefore satisfied. Thus, *p* is falsified.

Assume $p = Dis\ q\ r$ occurs negated in *S*. Then by the *BetaDis* requirement, either *Neg q* or *Neg r* occurs in *S*. The induction hypothesis applies to these, so *p* is satisfied as desired.

Assume $p = \text{Uni } q$ occurs in S . By the *DeltaUni* requirement, so does some instance $\text{sub } 0 \ t \ q$ for a term t in $\text{terms } S$. By the induction hypothesis, this is falsified by $M \ S$, and because t came from $\text{terms } S$, it is interpreted as itself. Thus, we have a counterexample that falsifies p .

Assume $p = \text{Exi } q$ occurs in S . By the *GammaExi* requirement, so do all instantiations using terms from S . Thus, these are all falsified by the model. These terms from S are interpreted as themselves by definition, so we have no witness for p in $\text{terms } S$ and $M \ S$ falsifies it. \square

We note that the above proof works for open and closed formulas alike because we consider both bound and free variables to be subterms of a formula.

5.4 Result

We start off by proving completeness for *uvalid* sequents. We need to relate these to saturated escape paths.

Lemma 8 (Saturated escape paths contradict uvalidity) *A sequent z on a saturated escape path, steps, cannot be uvalid:*

assumes $\text{fst}(\text{shd steps}) = (A, z)$ **and** epath steps **and** Saturated steps
shows $\neg \text{uvalid } z$

Proof Assume towards a contradiction that z is *uvalid*. By Theorem 2 the formulas on *steps* form a Hintikka set S . Every formula p in z also occurs in S , so by Theorem 4, the well formed model $M \ S$ (Lemma 7) falsifies all of them. This contradicts the *uvalidity* of z . \square

This leads to completeness for *uvalid* sequents:

Theorem 5 (Completeness wrt. *uvalid*) *The prover terminates for uvalid sequents:*

assumes $\text{uvalid } z$
defines $t \equiv \text{secavProver}(A, z)$
shows $\text{fst}(\text{root } t) = (A, z) \wedge \text{wft } t \wedge \text{tfinite } t$

Proof From the abstract framework (Lemma 2), either the thesis holds or a saturated escape path exists for our sequent, but assumed *uvalidity* and Lemma 8 contradict the latter. \square

Corollary 3 (Completeness wrt. SeCaV) *Termination for sequents derivable in SeCaV:*

assumes $\Vdash z$
defines $t \equiv \text{secavProver}(A, z)$
shows $\text{fst}(\text{root } t) = (A, z) \wedge \text{wft } t \wedge \text{tfinite } t$

Proof By the soundness of SeCaV (Theorem 3) and Theorem 5 for *uvalid* sequents. \square

6 Results and Discussion

We have presented an automated theorem prover for the Sequent Calculus Verifier system. The prover is capable of proving a number of selected exercise formulas very quickly, including formulas which are quite difficult for humans to prove. The prover does have some limitations, mostly related to performance and length of the generated proofs, since our proof search procedure is not very optimized for either of these metrics. In particular, our prover always instantiates quantified formulas with all terms in the sequent and breaks down all formulas as much as possible, even when some formulas are “obviously” irrelevant to the proof.

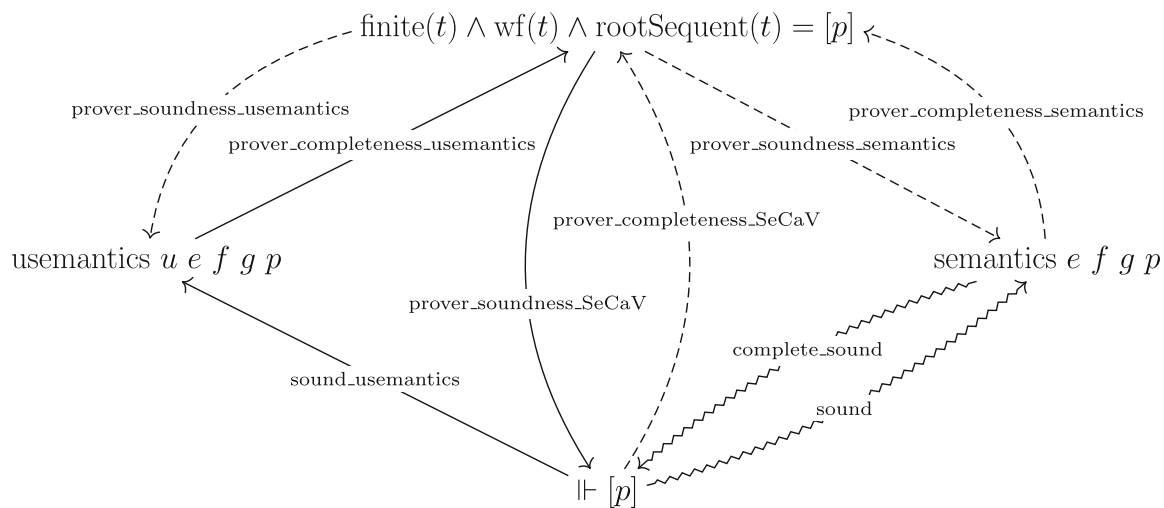


Fig. 6 Overview of our results. Solid arrows represent our main contributions, squiggly arrows represent theorems of the existing SeCaV system, and dashed arrows represent easy corollaries

6.1 Summary of Theorems

We have proven soundness and completeness of the proof search procedure with regards to the proof system of SeCaV (see Fig. 3). For soundness, this was done directly (in Theorem 1), while we took a detour through our notion of a bounded semantics to prove completeness (in Theorems 3 and 5, which led to Corollary 3). To justify the introduction of our bounded semantics, we can use the existing soundness and completeness theorems of the SeCaV proof system [18] and our results to prove that validity in the two semantics coincide. Additionally, a number of easy corollaries further linking the prover, the proof system and the two semantics follow from our results, and have been collected in Fig. 6 (refer to the formalization for the proofs). In the figure, the interpretations are implicitly universally quantified and for the bounded semantics we only consider well formed interpretations.

6.2 Example Proofs

As Knuth famously remarked [25], we must beware of a program that has only been proven correct, but not tested. To demonstrate that the automated theorem prover works, we examine some simple generated proofs. The prover generates proofs in the SeCaV Unshortener format [15]: first comes the formula to be proven, then the names of proof rules to apply and the resulting sequent after each application, with each formula in a sequent on its own line. Arguments to predicates and functions are given in square brackets and parentheses are used to disambiguate formulas.

We start with perhaps the simplest possible classical example, that $\neg p \vee p$. Figure 7 shows the proof generated by the prover on the left. This is the shortest possible proof of the formula in the SeCaV system, and the prover is thus on par with a human in this very simple case.

The next example is $\neg p(a) \vee \exists x.p(x)$. Figure 7 contains the generated proof on the right. It can be shortened since the quantified formula only needs to be instantiated once, by a . However, the prover always duplicates a γ -formula before instantiating it with *all* terms on the branch.

Finally, Fig. 8 shows a proof of $\forall x.A(f(x)) \longrightarrow \exists x.A(x)$ generated by the prover. This formula contains a function f , which the prover needs to use to instantiate the existentially

$\neg p \vee p$	$\neg p(a) \vee \exists x.p(x)$	
Dis (Neg p) p	Dis (Neg (p [a])) (Exi (p [0]))	Ext Exi (p [0]) Exi (p [0]) Neg (p [a]) p [a]
AlphaDis Neg p	AlphaDis Neg (p [a])	GammaExi[0] p [0]
Ext p	Ext Exi (p [0])	Ext Exi (p [0])
Ext p	Ext Exi (p [0])	Neg (p [a])
Neg p	Neg (p [a])	p [a]
Basic	GammaExi[a] p [a]	Ext p [a]
	Exi (p [0])	Neg (p [a])
	Exi (p [0])	Exi (p [0])
	Neg (p [a])	p [0]
	GammaExi[a] p [a]	Basic
	Exi (p [0])	
	Neg (p [a])	

Fig. 7 Proofs generated by the prover in SeCaV Unshortener format. Note that the proof on the right spans two columns

Imp (Uni (A [f [0]])) (Exi (A [0]))	GammaExi[0] A [0]	Ext Neg (Uni (A [f [0]]))
AlphaImp Neg (Uni (A [f [0]]))	Exi (A [0])	Neg (Uni (A [f [0]]))
Ext Exi (A [0])	Neg (Uni (A [f [0]]))	Exi (A [0])
Ext Exi (A [0])	A [f [0]]	A [f [0]]
Neg (Uni (A [f [0]]))	Ext Neg (Uni (A [f [0]]))	A [0]
GammaExi[f [0]] A [f [0]]	Exi (A [0])	Neg (A [f [f [0]]])
Exi (A [0])	A [f [0]]	GammaUni[0] Neg (A [f [0]])
Neg (Uni (A [f [0]]))	GammaUni[f [0]] Neg (A [f [f [0]]])	Neg (Uni (A [f [0]]))
Ext Exi (A [0])	Neg (Uni (A [f [0]]))	Exi (A [0])
Ext Exi (A [0])	Exi (A [0])	A [f [0]]
Neg (Uni (A [f [0]]))	A [f [0]]	A [0]
A [f [0]]	A [0]	Neg (A [f [f [0]]])
		Ext A [f [0]]
		A [0]
		Neg (Uni (A [f [0]]))
		Neg (A [f [f [0]]])
		Neg (A [f [0]])
		Exi (A [0])
		Basic

Fig. 8 Proof of the formula $\forall x.A(f(x)) \rightarrow \exists x.A(x)$ generated by the prover in SeCaV Unshortener format. Note that the prover applies a series of γ -rules to instantiate quantifiers with all possible terms and that these terms include free variables

quantified variable. Proving the formula also requires instantiating variables occurring as arguments of an application of f . This proof can be shortened significantly, since the prover instantiates both quantifiers with both x (i.e. de Bruijn index 0) and $f(x)$, even though only the instantiations leading to formulas involving $A(f(x))$ are needed for the proof. Note that the prover is able to instantiate formulas with free variables (i.e. de Bruijn indices with no corresponding quantifier) to overcome the lack of concrete constant names.

6.3 Verification Challenges

While verifying the prover, we discovered that our initial version was unsound due to a missing update of the term list when applying (multiple) δ -rules to a sequent. The attempted soundness proof failed in exactly this case, pointing us directly to the issue. Thus, the formal verification caught a critical flaw that we had missed in our testing and helped us fix it.

We have designed the prover to be easily verified and it mostly was. Especially the abstract framework worked well for our novel case with a deterministic prover for first-order logic. One obstacle, however, was in using a type to represent the domain in the SeCaV semantics (cf. Fig. 2). To build the countermodel, we need the domain to contain only the terms on the saturated escape path, but we cannot form this type, which depends on a local variable, in Isabelle/HOL. Here we would benefit from Isabelle integration of the work by Kunčar and Popescu [27] which adds exactly this capability to higher-order logic. Instead we introduced the bounded semantics (“the set-based relativization” in their terminology [27]) and proved a new soundness result for it (cf. Sect. 5.3.1). Otherwise the largest issue was dealing with substitutions using de Bruijn indices. We are excited to see how recent work by Blanchette et al. [5] for reasoning about syntax with bindings improves matters in this area.

6.4 Limitations and Future Work

There are a number of limitations and possibilities for optimization in the proof search itself. Most importantly, the focus of the procedure is on completeness, not performance. Our prover is much slower than state-of-the-art provers such as Vampire [26], and in particular, our prover is not fast enough to prove any but the easiest problems in the TPTP database [49]. Our goal was not to compete on speed, but simply to show that formal verification of provers with advanced features such as generation of proof certificates and support for functions is possible. The prover also cannot output counterexamples, even though these could be detected in some cases: our prover simply never terminates on invalid formulas.

We believe that the approach used for our prover is extendable to more sophisticated and optimized proof search procedures, albeit with considerably more work needed to formally verify them. The most obvious opportunity for optimization is controlling the order of proof rules. In systems with unordered sequents, it is generally better to apply as many α -rules as possible before applying β -rules to avoid duplicating work, but the prover simply applies rules in a fixed order. As mentioned in Sect. 3, this optimization can be done by working with “meta-rules” corresponding to groups of SeCaV rules such that a meta-rule e.g. applies as many α -rules as possible before continuing to the next “phase” of the proof. We have attempted to implement this, but found that it complicates the proofs considerably since this idea makes it much harder to determine when a proof rule is actually applied. In the proof of fairness and the proof that the formulas on saturated escape paths form Hintikka sets, we need to know that certain formulas are preserved until proof rules are eventually applied to them. By introducing phases in the proof, proving this becomes much more difficult, since we then need to prove that each phase actually ends (requiring some measure which depends on the specific sequents in question), and to locate each rule within the meta-rule it is part of. We thus leave optimizations in this vein as future work. We note that, since the SeCaV system requires application of the EXT rule to permute sequents, and proof rules only apply to the first formula in a sequent, the optimization described above may not always reduce the number of SeCaV proof steps needed to prove a formula, and some heuristics would probably be needed to produce reasonably short proofs in all cases.

Instead of introducing phases by working with meta-rules, we could also imagine allowing the proof strategy to depend on the actual formula being proven more directly, by choosing a rule based on the current sequent or possibly even the entire proof tree. The framework used for the prover does not allow this (since the rules used for the proof search must be given as a fixed stream of rules) and a new framework would have to be developed to implement optimizations in this vein. Such a framework would also make it possible to avoid useless rule applications and to easily implement proof strategies based on heuristics. It is unclear how the additional flexibility of allowing rules to depend on formulas would impact the difficulty of the proofs of soundness and completeness, and we thus leave developing such a framework as future work.

Another optimization could be to only support closed formulas and thus reduce the number of subterms of a given formula. For our current Herbrand interpretation, we need variables to be subterms, but if we only considered closed terms, we could do away with this. This would however require a change to the proof strategy to make the prover invent new names instead of using free variable in cases such as the proof in Fig. 8.

The length of proofs could also be optimized by performing more post-processing of the found proofs, for example by removing unnecessary instantiations or rule applications that do not contribute to proving a branch. This would not improve the performance in the sense that the prover would still spend the same amount of time finding the proof, but it could reduce the length of some proofs significantly. The proof trees generated by the prover already require some (unverified) post-processing to obtain proofs in the SeCaV system. The existing post-processing also removes rule applications that do nothing to the sequent, and merges consecutive EXT rules, of which there may be many after removing other rule applications. The current post-processing does not, however, consider the usefulness of a rule in the context of the larger proof, and this could be implemented to eliminate entire lines of reasoning that end up amounting to nothing. This analysis can of course only be performed after seeing the entire proof, and would thus not be able to increase the speed of the prover. It would also be interesting to move these steps from Haskell into Isabelle/HOL and extend the proofs to cover them.

Another way to shorten the generated proofs would be to disable rules that do not actually change the sequent. For instance, an ALPHADIS rule can affect a dozen formulas in a sequent at once or zero, when there are no disjunctions, but the rule is applied regardless. This can lead to a lot of useless rule applications in the final proof, which are removed by the above mentioned post-processing. In terms of how much work the prover does, we have to check the sequent anyway, to see if a given rule applies to it, so there is no immediate benefit compared to simply applying the rule with no effect. Moreover, the current design allows us, as mentioned, to forgo an always-enabled IDLE rule, otherwise needed to satisfy the framework, since all rules are always enabled. It could, however, have a positive impact on the memory usage of the prover, since the proof tree would not grow as much, taking up unnecessary space. This should be investigated in future work.

7 Conclusion

We have designed, implemented and verified an automated theorem prover for first-order logic with functions in Isabelle/HOL. We have used an existing framework in a novel way to get us part of the way towards completeness and we have extended existing techniques on countermodels over restricted domains to reach our destination. We build on the existing

SeCaV system and contribute an automatic way of finding derivations to the project. Thus, we have demonstrated the utility of Isabelle/HOL for implementing and verifying executable software and the strength of its libraries in doing so. Our prover handles the full syntax of first-order logic with functions and constructs human-readable proof certificates in a sequent calculus. We hope our work inspires others to verify more sophisticated provers in the same vein.

Supplementary Information The online version contains supplementary material available at <https://doi.org/10.1007/s10817-024-09697-3>.

Acknowledgements We would like to thank Agnes Moesgård Eschen, Alexander Birch Jensen, Anders Schlichtkrull, Simon Tobias Lund and Jørgen Villadsen for comments on drafts. We are very grateful to the anonymous reviewers at ITP 2022 and the Journal of Automated Reasoning for their thoughtful comments.

Declarations

Conflict of interest The authors have no relevant financial or non-financial interests to disclose.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

References

1. Ballarín, C.: Locales: a module system for mathematical theories. *J. Autom. Reason.* **52**(2), 123–153 (2014). <https://doi.org/10.1007/s10817-013-9284-7>
2. Ben-Ari, M.: *Mathematical Logic for Computer Science*, pp. 149–150. Springer, London (2012). <https://doi.org/10.1007/978-1-4471-4129-7>
3. Bentkamp, A., Blanchette, J., Tourret, S., Vukmirović, P.: Superposition for full higher-order logic. In: Platzer, A., Sutcliffe, G. (eds.) *Automated Deduction – CADE 28. Lecture Notes in Computer Science*, vol. 12699, pp. 396–412. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-79876-5_23
4. Berghofer, S.: First-order logic according to Fitting. *Archive of Formal Proofs. Formal proof development* (2007). <https://isa-afp.org/entries/FOL-Fitting.html>
5. Blanchette, J.C., Gheri, L., Popescu, A., Traytel, D.: Bindings as bounded natural functors. *Proc. ACM Program. Lang.* **3**(POPL, Article 22), 1–34 (2019). <https://doi.org/10.1145/3290335>
6. Blanchette, J.C., Popescu, A., Traytel, D.: Abstract completeness. *Archive of Formal Proofs. Formal proof development* (2014). https://isa-afp.org/entries/Abstract_Completeness.html
7. Blanchette, J.C., Popescu, A., Traytel, D.: Unified classical logic completeness. In: Demri, S., Kapur, D., Weidenbach, C. (eds.) *Automated Reasoning. Lecture Notes in Computer Science*, vol. 8562, pp. 46–60. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08587-6_4
8. Blanchette, J.C., Popescu, A.: Mechanizing the metatheory of Sledgehammer. In: Fontaine, P., Ringeissen, C., Schmidt, R.A. (eds.) *Frontiers of Combining Systems. Lecture Notes in Computer Science*, vol. 8152, pp. 245–260. Springer, Berlin (2013). https://doi.org/10.1007/978-3-642-40885-4_17
9. Blanchette, J.C.: Formalizing the metatheory of logical calculi and automatic provers in Isabelle/HOL (invited talk). In: Mahboubi, A., Myreen, M.O. (eds.) *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019*, pp. 1–13. ACM, New York (2019). <https://doi.org/10.1145/3293880.3294087>
10. Blanchette, J.C., Fleury, M., Lammich, P., Weidenbach, C.: A verified SAT solver framework with learn, forget, restart, and incrementality. *J. Autom. Reason.* **61**(1–4), 333–365 (2018). <https://doi.org/10.1007/s10817-018-9455-7>

11. Blanchette, J.C., Popescu, A., Traytel, D.: Soundness and completeness proofs by coinductive methods. *J. Autom. Reason.* **58**(1), 149–179 (2017). <https://doi.org/10.1007/s10817-016-9391-3>
12. Breitner, J.: Visual theorem proving with the Incredible Proof Machine. In: Blanchette, J., Merz, S. (eds.) *Interactive Theorem Proving. Lecture Notes in Computer Science*, vol. 9807, pp. 123–139. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-43144-4_8
13. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) *Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science*, vol. 4963, pp. 337–340. Springer, Berlin (2008). https://doi.org/10.1007/978-3-540-78800-3_24
14. Fleury, M.: Optimizing a verified SAT solver. In: Badger, J.M., Rozier, K.Y. (eds.) *NASA Formal Methods. Lecture Notes in Computer Science*, vol. 11460, pp. 148–165. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-20652-9_10
15. From, A.H., Jacobsen, F.K., Villadsen, J.: SeCaV: a sequent calculus verifier in Isabelle/HOL. In: Ayala-Rincón, M., Bonelli, E. (eds.) *16th Logical and Semantic Frameworks with Applications (LSFA 2021). Electronic Proceedings in Theoretical Computer Science*, vol. 357, pp. 38–55 (2022). <https://doi.org/10.4204/EPTCS.357.4>
16. From, A.H., Jacobsen, F.K.: A sequent calculus prover for first-order logic with functions. *Archive of Formal Proofs. Formal proof development* (2022). https://isa-afp.org/entries/FOL_Seq_Calc2.html
17. From, A.H., Jacobsen, F.K.: Verifying a sequent calculus prover for first-order logic with functions in Isabelle/HOL. In: Andronick, J., de Moura, L. (eds.) *13th International Conference on Interactive Theorem Proving (ITP 2022). Leibniz International Proceedings in Informatics (LIPIcs)*, vol. 237, pp. 1–22. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, Dagstuhl (2022). <https://doi.org/10.4230/LIPIcs.ITP.2022.13>
18. From, A.H., Jensen, A.B., Schlichtkrull, A., Villadsen, J.: Teaching a formalized logical calculus. In: Quaresma, P., Neuper, W., Marcos, J. (eds.) *Theorem Proving Components for Educational Software (ThEdu'19). Electronic Proceedings in Theoretical Computer Science*, vol. 313, pp. 73–92 (2020). <https://doi.org/10.4204/EPTCS.313.5>
19. From, A.H., Villadsen, J., Blackburn, P.: Isabelle/HOL as a meta-language for teaching logic. In: Marcos, J., Neuper, W., Quaresma, P. (eds.) *Theorem Proving Components for Educational Software (ThEdu'20). Electronic Proceedings in Theoretical Computer Science*, vol. 328, pp. 18–34 (2020). <https://doi.org/10.4204/EPTCS.328.2>
20. From, A.H.: Epistemic logic: Completeness of modal logics. *Archive of Formal Proofs. Formal proof development* (2018). https://isa-afp.org/entries/Epistemic_Logic.html
21. From, A.H.: Formalized soundness and completeness of epistemic logic. In: Silva, A., Wassermann, R., de Queiroz, R. (eds.) *Logic, Language, Information, and Computation. Lecture Notes in Computer Science*, vol. 13038, pp. 1–15. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-88853-4_1
22. Jacobsen, F.K.: Formalization of logical systems in Isabelle: an automated theorem prover for the Sequent Calculus Verifier. Master's thesis, Technical University of Denmark (June 2021). <https://findit.dtu.dk/en/catalog/2691928304>
23. Jensen, A.B., Larsen, J.B., Schlichtkrull, A., Villadsen, J.: Programming and verifying a declarative first-order prover in Isabelle/HOL. *AI Commun.* **31**(3), 281–299 (2018). <https://doi.org/10.3233/AIC-180764>
24. Kammüller, F., Wenzel, M., Paulson, L.C.: Locales - A sectioning concept for Isabelle. In: Bertot, Y., Dowek, G., Hirschowitz, A., Paulin-Mohring, C., Théry, L. (eds.) *Theorem Proving in Higher Order Logics. Lecture Notes in Computer Science*, vol. 1690, pp. 149–165. Springer, Berlin (1999). https://doi.org/10.1007/3-540-48256-3_11
25. Knuth, D.E., van Emde Boas, P.: The correspondence between Donald E. Knuth and Peter van Emde Boas on priority dequeues during the spring of 1977. Facsimile edition (1977). <https://staff.fnwi.uva.nl/p.vanemdeboas/knuthnote.pdf>
26. Kovács, L., Voronkov, A.: First-order theorem proving and Vampire. In: Sharygina, N., Veith, H. (eds.) *Computer Aided Verification. Lecture Notes in Computer Science*, vol. 8044, pp. 1–35. Springer, Berlin (2013). https://doi.org/10.1007/978-3-642-39799-8_1
27. Kunčar, O., Popescu, A.: From types to sets by local type definition in higher-order logic. *J. Autom. Reason.* **62**(2), 237–260 (2019). <https://doi.org/10.1007/s10817-018-9464-6>
28. Lammich, P.: The GRAT tool chain. In: Gaspers, S., Walsh, T. (eds.) *Theory and Applications of Satisfiability Testing—SAT 2017. Lecture Notes in Computer Science*, vol. 10491, pp. 457–463. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66263-3_29
29. Lammich, P.: Efficient verified (UN)SAT certificate checking. *J. Autom. Reason.* **64**(3), 513–532 (2020). <https://doi.org/10.1007/s10817-019-09525-z>
30. Lescuyer, S.: Formalizing and Implementing a Reflexive Tactic for Automated Deduction in Coq. PhD thesis, Université Paris Sud - Paris XI (January 2011). <https://tel.archives-ouvertes.fr/tel-00713668>













31. Lochbihler, A., Stoop, P.: Lazy algebraic types in Isabelle/HOL. In: Isabelle Workshop 2018 (2018). https://files.sketis.net/Isabelle_Workshop_2018/Isabelle_2018_paper_2.pdf
32. Marić, F., Spasić, M., Thiemann, R.: An incremental simplex algorithm with unsatisfiable core generation. *Archive of Formal Proofs. Formal proof development* (2018). <https://isa-afp.org/entries/Simplex.html>
33. Marić, F.: Formal verification of modern SAT solvers. *Archive of Formal Proofs. Formal proof development* (2008). <https://isa-afp.org/entries/SATSolverVerification.html>
34. Marić, F.: Formal verification of a modern SAT solver by shallow embedding into Isabelle/HOL. *Theor. Comput. Sci.* **411**(50), 4333–4356 (2010). <https://doi.org/10.1016/j.tcs.2010.09.014>
35. Michaelis, J., Nipkow, T.: Formalized proof systems for propositional logic. In: Abel, A., Forsberg, F.N., Kaposi, A. (eds.) *23rd International Conference on Types for Proofs and Programs (TYPES 2017)*. *Leibniz International Proceedings in Informatics (LIPIcs)*, vol. 104, pp. 1–16. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, Dagstuhl (2018). <https://doi.org/10.4230/LIPIcs.TYPES.2017.5>
36. Michaelis, J., Nipkow, T.: Propositional proof systems. *Archive of Formal Proofs. Formal proof development* (2017). https://isa-afp.org/entries/Propositional_Proof_Systems.html
37. Pastre, D.: Muscadet 2.3: A knowledge-based theorem prover based on natural deduction. In: Gore, R., Leitsch, A., Nipkow, T. (eds.) *Automated Reasoning. Lecture Notes in Computer Science*, vol. 2083, pp. 685–689. Springer, Berlin (2001). https://doi.org/10.1007/3-540-45744-5_56
38. Pelletier, F.J.: Automated natural deduction in THINKER. *Stud. Logica* **60**(1), 3–43 (1998). <https://doi.org/10.1023/A:1005035316026>
39. Peltier, N.: A variant of the superposition calculus. *Archive of Formal Proofs. Formal proof development* (2016). <https://isa-afp.org/entries/SuperCalc.html>
40. Peltier, N.: Propositional resolution and prime implicates generation. *Archive of Formal Proofs. Formal proof development* (2016). <https://isa-afp.org/entries/PropResPI.html>
41. Ridge, T., Margetson, J.: A mechanically verified, sound and complete theorem prover for first order logic. In: Hurd, J., Melham, T. (eds.) *Theorem Proving in Higher Order Logics (TPHOLs 2005)*. *Lecture Notes in Computer Science*, vol. 3603, pp. 294–309. Springer, Berlin (2005). https://doi.org/10.1007/11541868_19
42. Schlichtkrull, A., Blanchette, J.C., Traytel, D., Waldmann, U.: Formalizing Bachmair and Ganzinger’s ordered resolution prover. In: Galmiche, D., Schulz, S., Sebastiani, R. (eds.) *Automated Reasoning. Lecture Notes in Computer Science*, vol. 10900, pp. 89–107. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-94205-6_7
43. Schlichtkrull, A., Blanchette, J.C., Traytel, D.: A verified prover based on ordered resolution. In: *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs. CPP 2019*, pp. 152–165. Association for Computing Machinery, New York (2019). <https://doi.org/10.1145/3293880.3294100>
44. Schlichtkrull, A., Villadsen, J.: Paraconsistency. *Archive of Formal Proofs. Formal proof development* (2016). <https://isa-afp.org/entries/Paraconsistency.html>
45. Schlichtkrull, A.: Formalization of the resolution calculus for first-order logic. *J. Autom. Reason.* **61**(1–4), 455–484 (2018). <https://doi.org/10.1007/s10817-017-9447-z>
46. Shankar, N., Vaucher, M.: The mechanical verification of a DPLL-based satisfiability solver. In: Haeusler, E.H., del Cerro, L.F. (eds.) *Proceedings of the Fifth Logical and Semantic Frameworks, with Applications Workshop (LSFA 2010)*. *Electronic Notes in Theoretical Computer Science*, vol. 269, pp. 3–17 (2011). <https://doi.org/10.1016/j.entcs.2011.03.002>
47. Smullyan, R.M.: *First-Order Logic*. Dover, Mineola (1995)
48. Spasić, M., Marić, F.: Formalization of incremental simplex algorithm by stepwise refinement. In: Gianakopoulou, D., Méry, D. (eds.) *FM 2012: Formal Methods. Lecture Notes in Computer Science*, vol. 7436, pp. 434–449. Springer, Berlin (2012). https://doi.org/10.1007/978-3-642-32759-9_35
49. Sutcliffe, G.: The TPTP problem library and associated infrastructure. From CNF to TH0, TPTP v6.4.0. *J. Autom. Reason.* **59**(4), 483–502 (2017). <https://doi.org/10.1007/s10817-017-9407-7>
50. Villadsen, J., From, A.H., Jensen, A.B., Schlichtkrull, A.: Interactive theorem proving for logic and information. In: Loukanova, R. (ed.) *Natural Language Processing in Artificial Intelligence—NLPinAI 2021*. *Studies in Computational Intelligence*, vol. 999, pp. 25–48. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-90138-7_2
51. Villadsen, J., From, A.H., Schlichtkrull, A.: Natural Deduction Assistant (NaDeA). In: Quaresma, P., Neuper, W. (eds.) *Theorem Proving Components for Educational Software (ThEdu’18)*. *Electronic Proceedings in Theoretical Computer Science*, vol. 290, pp. 14–29 (2019). <https://doi.org/10.4204/EPTCS.290.2>
52. Villadsen, J., Jacobsen, F.K.: Using Isabelle in two courses on logic and automated reasoning. In: Ferreira, J.F., Mendes, A., Menghi, C. (eds.) *Formal Methods Teaching. Lecture Notes in Computer Science*, vol. 13122, pp. 117–132. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-91550-6_9

53. Villadsen, J., Schlichtkrull, A., From, A.H.: A verified simple prover for first-order logic. In: Konev, B., Urban, J., Rümmer, P. (eds.) Practical Aspects of Automated Reasoning. CEUR Workshop Proceedings, vol. 2162, pp. 88–104. CEUR-WS, Aachen (2018). <https://ceur-ws.org/Vol-2162/paper-08.pdf>
54. Villadsen, J., Schlichtkrull, A.: Formalizing a paraconsistent logic in the Isabelle proof assistant. In: Hameurlain, A., Küng, J., Wagner, R., Decker, H. (eds.) Transactions on Large-Scale Data- and Knowledge-Centered Systems XXXIV: Special Issue on Consistency and Inconsistency in Data-Centric Applications. Lecture Notes in Computer Science, vol. 10620, pp. 92–122. Springer, Berlin (2017). https://doi.org/10.1007/978-3-662-55947-5_5

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.



The Concurrent Calculi Formalisation Benchmark

Marco Carbone¹ , David Castro-Perez² , Francisco Ferreira³ ,
Lorenzo Gheri⁴ , Frederik Krogsdal Jacobsen⁵ , Alberto Momigliano⁶ ,
Luca Padovani⁷ , Alceste Scalas⁵ , Dawit Tirore¹ , Martin Vassor⁸ ,
Nobuko Yoshida⁸ , and Daniel Zackon⁹ 

¹ IT University of Copenhagen, Copenhagen, Denmark
`{maca,dati}@itu.dk`

² University of Kent, Canterbury, UK
`d.castro-perez@kent.ac.uk`

³ Royal Holloway, University of London, Egham, UK
`francisco.ferreiraruiz@rhul.ac.uk`

⁴ University of Liverpool, Liverpool, UK
`lorenzo.gheri@liverpool.ac.uk`

⁵ Technical University of Denmark, Kgs. Lyngby, Denmark
`{fkjac,alcsj}@dtu.dk`

⁶ Università degli Studi di Milano, Milan, Italy
`momigliano@di.unimi.it`

⁷ Università di Camerino, Camerino, Italy
`luca.padovani@unicam.it`

⁸ University of Oxford, Oxford, UK
`{martin.vassor,nobuko.yoshida}@cs.ox.ac.uk`

⁹ McGill University, Montreal, Canada
`daniel.zackon@mail.mcgill.ca`

Abstract. POPLMark and POPLMark Reloaded sparked a flurry of work on machine-checked proofs, and fostered the adoption of proof mechanisation in programming language research. Both challenges were purposely limited in scope, and they do not address concurrency-related issues. We propose a new collection of benchmark challenges focused on the difficulties that typically arise when mechanising formal models of concurrent and distributed programming languages, such as process calculi. Our benchmark challenges address three key topics: linearity, scope extrusion, and coinductive reasoning. The goal of this new benchmark is to clarify, compare, and advance the state of the art, fostering the adoption of proof mechanisation in future research on concurrency.

Keywords: Mechanisation · Process calculi · Benchmark · Linearity · Scope extrusion · Coinduction

1 Introduction

The POPLMark challenge [4] played a pivotal role in advancing the field of proof assistants, libraries, and best practices for the mechanisation of programming

language research. By providing a shared framework for systematically evaluating mechanisation techniques, it catalysed a significant shift towards publications that include mechanised proofs within the programming language research community. POPLMark Reloaded [1] introduced a similar programme for proofs using logical relations. These initiatives had a narrow focus, and their authors recognised the importance of addressing topics such as coinduction and linearity in the future.

In this spirit, we introduce a new collection of benchmarks crafted to tackle common challenges encountered during the mechanisation of formal models of concurrent and distributed programming languages. We focus on process calculi, as they provide a simple but realistic showcase of these challenges. Concurrent calculi are notably subtle: for instance, it took some years before an incorrect subject reduction proof in the original paper on session subtyping [25] was discovered and then rectified in the extended journal version [26] with the use of polarities. Similarly, other key results in papers on session types have subsequently been proven incorrect [27, 45], demonstrating the need for machine-checked proofs.

While results about concurrent formalisms have already been mechanised (as we will discuss further below), our experience is that choosing appropriate mechanisation techniques and tools remains a significant challenge and that their trade-offs are not well understood. This often leads researchers toward a trial-and-error approach, resulting in sub-optimal solutions, wasted mechanisation efforts, and techniques and results that are hard to reuse. For example, Cruz-Filipe et al. [17] note that the high-level parts of mechanised proofs closely resemble the informal ones, while the main challenge lies in getting the infrastructure right.

Our benchmark challenges (detailed on our website linked below) consider *in isolation* three key aspects that frequently pose difficulties when mechanising concurrency theory: *linearity*, *scope extrusion*, and *coinductive reasoning*, as we will discuss in more detail in the next section. Mechanisations must often address several of these aspects at the same time; however, we see the combination of techniques as a next step, as argued in Sect. 3.

We have begun collecting solutions to our challenges on our website:

<https://concurrentbenchmark.github.io/>

We intend to use the website to promote best practices and tutorials derived from solutions to our challenges. We encourage readers to try the challenges using their favourite techniques, and to send us their solutions and experience reports.

2 Overview and Design Considerations

In this section, we outline the factors considered when designing the benchmark challenges. We begin with some general remarks, then describe the individual design considerations for each challenge, and the criteria for evaluating solutions.

Similarly to the authors of POPLMark, we seek to answer several questions about the mechanisation of the meta-theory of process calculi:

- (Q1) What is the current state of the art?
- (Q2) Which techniques and best practices can be recommended?
- (Q3) What improvements are needed to make mechanisation tools more user-friendly?

To provide a framework in which to answer these questions, our benchmark is designed to satisfy three main design goals:

- (G1) To enable the comparison of proof mechanisation approaches by making the challenges accessible to mechanisation experts who may be unfamiliar with concurrency theory;
- (G2) To encourage the development of guidelines and tutorials demonstrating and comparing existing proof mechanisation techniques, libraries, and proof assistant features; and
- (G3) To prioritise the exploration of reusable mechanisation techniques.

We also aim at strengthening the culture of mechanisation, by rallying the community to collaborate on exploring and developing new tools and techniques.

To achieve design goal (G1), our challenges explore the three aspects (linearity, scope extrusion, coinduction) independently, so that they may be solved individually and in any order; each challenge is small and easily understandable with basic knowledge of textbook concurrency theory, process calculi, and type theory. For mechanisation experts, our challenges should thus be accessible even without any prior understanding of process calculi. The process calculi used in the challenges focus on the features that we want to emphasise, and omit all constructs that would complicate the mechanisation without bringing tangible insights. For concurrency experts venturing into mechanisation, our challenges thus serve as good first steps. The minimality and uniformity of the calculi also allows us to target design goal (G2). For experts in both mechanisation and concurrency, our challenges serve as a framework in which to consider and share best practices and tutorials. Aligned with design goal (G3), our challenges concern the fundamental meta-theory of process calculi. Our challenges centre around well-established results, showcasing proof techniques that can be leveraged in many applications (as we will further discuss in Sect. 3).

2.1 Linearity

Linear typing systems enable the tracking of resource usage in a program. In the case of typed (in particular, session-typed) process calculi, linearity is widely used for checking if and how a channel is used to send or receive values. This substructurality [39, Ch. 1] gives rise to mechanisation difficulties: *e.g.* deciding how to *split the typing context* in a parallel composition.

The goal of our challenge on linear reasoning is to prove a type safety theorem for a process calculus with session types, by combining subject reduction with the absence of errors. For simplicity we model only linear (as opposed to shared) channels. Inspired by Vasconcelos [49], we define a syntax where a restriction (νab) binds two dual names a and b as opposite endpoints of the same channel;

their duality is reflected in the type system. We model a simple notion of error: well-typed processes must never use dual channel endpoints in a non-dual way (*e.g.* by performing concurrent send/receive operations on the same endpoint, or two concurrent send operations on dual endpoints). The operational semantics is a standard reduction relation. Proving subject reduction thus requires proving type preservation for structural congruence.

We designed this challenge to focus on linear reasoning while minimising definitions and other concerns. We therefore forgo name passing: send/receive operations only support values that do not include channel names, so the topology of the communication network described by a process cannot change. We do not allow recursion or replication, hence infinite behaviours cannot be expressed. We also forgo more sophisticated notions of error-freedom (*e.g.* deadlock freedom) as proving them would distract from the core linear aspects of the challenge.

In mechanised meta-theory, addressing linearity means choosing an appropriate representation of a linear context. While the latter is perhaps best seen as a multiset, most proof assistants have better support for lists. This representation is intuitive, but may require establishing a large number of technical lemmata that are orthogonal to the problem under study (in our case, proving type safety for session types). Several designs are possible: one can label occurrences of resources to constrain their usage (*e.g.* [16]), or impose a multiset structure over lists (*e.g.* [15, 19]). Alternatively, contexts can be implemented as finite maps (as in [12]), whose operations are sensitive to a linear discipline. In all these cases, the effort required to develop the infrastructure is significant. One alternative strategy is to bypass the problem of context splitting by adopting ideas from algorithmic linear type checking. One such approach, known as “typing with leftovers,” is exemplified in [51]. Similarly, context splitting can be eliminated by delegating linearity checks to a *linear predicate* defined on the process structure (*e.g.* [44]). These checks serve as additional conditions within the typing rules. Whatever the choice, list-based encodings can be refined to be intrinsically-typed if the proof assistant supports dependent types (see [16, 42, 47]).

A radically different approach is to adopt a *substructural* meta-logical framework, which handles resource distribution implicitly, including splitting and substitution: users need only map their linear operations to the ones offered by the framework. The only such framework is *Celf* [46] (see the encoding of session types in [8]); unfortunately, *Celf* does not yet fully support the verification of meta-theoretic properties. A compromise is the *two-level* approach, *i.e.* encoding a substructural specification logic in a mainstream proof assistant and then using that logic to state and prove linear properties (for a recent example, see [23]).

2.2 Scope Extrusion

This challenge revolves around the mechanisation of scope extrusion, by which a process can send restricted names to another process, as long as the restriction can safely be extruded to include the receiving process. The setting for this challenge is a “classic” untyped π -calculus, where (unlike the calculi in the other

challenges) names can be sent and received, and bound by input constructs. We define two different semantics for our system:

1. A reduction system: this avoids explicit reasoning about scope extrusion by using structural congruence, allowing implementers to explore different ways to encode the latter (*e.g.* via process contexts or compatible refinement);
2. An (early) labelled transition system.

The goal of our challenge on scope extrusion is to prove that the two semantics are equivalent up to structural congruence.

This is the challenge most closely related to POPLMark, as it concerns the properties of binders, whose encoding has been extensively studied with respect to λ -calculi. Process calculi present additional challenges, typically including several different binding constructs: inputs bind a received name or value, recursive processes bind recursion variables, and restrictions bind names. The first two act similarly to the binders in λ -calculi, but restrictions may be more challenging due to scope extrusion. Scope extrusion requires reasoning about free variables, so approaches that identify α -equivalent processes cannot be directly applied.

Given those peculiarities, the syntax and semantics of π -calculi have been mechanised from an early age (see [37]) with many proof assistants and in many encoding styles. Despite this, almost all of these mechanisations rely on ad-hoc solutions to encode scope extrusion. They range from concrete encodings based on named syntax [37] to basic de Bruijn [30,38] and locally-nameless representation [12]. Nominal approaches are also common (see [6]), but they may be problematic in proof assistants based on constructive type theories. An overall comparison is still lacking, but the case study [3] explores four approaches to encoding binders in Coq in the context of higher-order process calculi. The authors report that working directly with de Bruijn indices was easiest since the approaches developed for λ -calculus binders worked poorly with scope extrusion.

Higher-order abstract syntax (HOAS) has seen extensive use in formal reasoning in this area [13,14,22,32,48]. Its weak form aligns reasonably well with mainstream inductive proof assistants, significantly simplifying the encoding of typing systems and operational semantics. However, when addressing more intricate concepts like bisimulation, extensions to HOAS are needed. These extensions may take the form of additional axioms [32] or require niche proof assistants such as Abella, which features a special quantifier for handling properties related to names [24].

2.3 Coinduction

Process calculi typically include constructs that allow processes to adopt infinite behaviours. Coinduction serves as a fundamental method for the definition and analysis of infinite objects, enabling the examination of their behaviours.

The goal of our challenge on coinductive reasoning is to prove that *strong barbed bisimilarity* can be turned into a congruence by making it sensitive to substitution and parallel composition. The crux of our challenge is the effective

use of coinductive up-to techniques. The intention is that the result should be relatively easy to achieve once the main properties of bisimilarity are established.

The setting for our challenge is an untyped π -calculus augmented with process replication in order to enable infinite behaviours. We do not include name passing since it is orthogonal to our aim of exploring coinductive proof techniques. We base our definition of bisimilarity on a labelled transition system semantics and an observability predicate describing the communication steps available to a process. The description of strong barbed bisimulation is one of the first steps when studying the behaviour of process calculi, both in textbooks (*e.g.* [43]) and in existing mechanisations. Though weak barbed congruence is a more common behavioural equivalence, we prefer strong equivalences to simplify the theory by avoiding the need to abstract over the number of internal transitions in a trace.

While many proof assistants support coinductive techniques, they do so through different formalisms. Some systems even offer multiple abstractions for utilising coinduction. For instance, Agda offers musical notation, co-patterns and productivity checking via sized types [2]; Coq features guarded recursion and refined fixed point approaches via libraries for *e.g.* parameterised coinduction [34], coinduction up-to [41] and interaction trees [50].

When reasoning over bisimilarity many authors rely on the native coinduction offered by the chosen proof assistant [7, 27, 35, 47], while others prefer a more “set-theoretic” approach [6, 30, 36, 40]. Some use both and establish an internal adequacy [32]. Few extend the proof assistant foundations to allow, *e.g.*, reasoning about bisimilarity up-to [14].

2.4 Evaluation Criteria

The motivation behind our benchmark is to obtain evidence towards answering questions (Q1) to (Q3). We are therefore interested not only in the solutions, but also in the experience of solving the challenges with the chosen approach. Solutions to our challenges should be compared on three axes:

1. Mechanisation overhead: the amount of manually-written infrastructure and setup needed to express the definitions in the mechanisation;
2. Adequacy of the formal statements in the mechanisation: whether the proven theorems are easily recognisable as the theorems from the challenge; and
3. Cost of entry for the tools and techniques employed: the difficulty of learning to use the techniques.

Solutions to our challenges need not strictly follow the definitions and lemmata set out in the challenge text, but solutions which deviate from the original challenges should present more elaborate argumentation for their adequacy.

3 Future Work and Conclusions

Our benchmark challenges do not cover all issues in the field, but focus on the fundamental aspects of linearity, scope extrusion, and coinduction. Many mech-

anisations need to combine techniques to handle several of these aspects, and some may also need to handle aspects that are not covered by our benchmark.

Combining techniques for mechanising the fundamental aspects covered in our benchmark is non-trivial. While we focus on the aspects individually to simplify the challenges, we are also interested in exploring how techniques interact.

Much current research on concurrent calculi includes aspects that are not covered by our benchmark challenges, for example constructs such as choice and recursion. Some interesting research topics that build on the fundamental aspects in our benchmark include multiparty session types [31], choreographies [11], higher-order calculi [29], conversation types [10], psi-calculi [5], and encodings between different calculi [20,28]. The meta-theory of these topics includes aspects—*e.g.* well-formedness conditions on global types, partiality of end-point projection functions, *etc.*—that we do not address.

Our coinduction challenge only treats two notions of process equivalence, but many more exist in the literature. Coinduction may also play a role in recursive processes and session types: recursive session types can be expressed in *infinitary form* by interpreting their typing rules coinductively [21,33].

Unlike POPLMark, we consider *animation* of calculi (as in [13]) out of scope for our benchmark. Finally, our challenges encourage, but do not require, exploring proof automation, as offered by *e.g.* the *Hammer* protocol [9,18].

Ultimately, the fundamental aspects covered by our benchmark serve as the building blocks for most current research on concurrent calculi. It is our hope and aim that exploring and comparing solutions to our challenges will move the community closer to a future where the key basic proof techniques for concurrent calculi are as easy to mechanise as they are to write on paper.

Acknowledgments. The work is partially supported by the UK Engineering and Physical Sciences Research Council (EPSRC) grants EP/T006544/2, EP/V000462/1 and EP/X015955/1; Independent Research Fund Denmark RP-1 grant “Hyben”; and Independent Research Fund Denmark RP-1 grant “MECHANIST”.

Disclosure of Interests. Alberto Momigliano is a member of the Gruppo Nazionale Calcolo Scientifico-Istituto Nazionale di Alta Matematica (GNCS-INdAM).

References

1. Abel, A., et al.: POPLMark reloaded: mechanizing proofs by logical relations. *J. Funct. Program.* **29**, 19 (2019). <https://doi.org/10.1017/S0956796819000170>
2. Abel, A., Pientka, B., Thibodeau, D., Setzer, A.: Copatterns: programming infinite structures by observations. In: *POPL 2013: Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 27–38. ACM, New York (2013). <https://doi.org/10.1145/2429069.2429075>
3. Ambal, G., Lenglet, S., Schmitt, A.: $HO\pi$ in Coq. *J. Autom. Reason.* **65**(1), 75–124 (2021). <https://doi.org/10.1007/S10817-020-09553-0>
4. Aydemir, B.E., et al.: Mechanized metatheory for the masses: the POPLMARK challenge. In: Hurd, J., Melham, T. (eds.) *TPHOLs 2005*. LNCS, vol. 3603, pp. 50–65. Springer, Heidelberg (2005). https://doi.org/10.1007/11541868_4

5. Bengtson, J., Johansson, M., Parrow, J., Victor, B.: Psi-calculi: a framework for mobile processes with nominal data and logic. *Log. Methods Comput. Sci.* **7** (2011). [https://doi.org/10.2168/LMCS-7\(1:11\)2011](https://doi.org/10.2168/LMCS-7(1:11)2011)
6. Bengtson, J., Parrow, J.: Formalising the pi-calculus using nominal logic. *Log. Methods Comput. Sci.* **5** (2009). [https://doi.org/10.2168/LMCS-5\(2:16\)2009](https://doi.org/10.2168/LMCS-5(2:16)2009)
7. Bengtson, J., Parrow, J., Weber, T.: Psi-calculi in Isabelle. *J. Autom. Reason.* **56**, 1–47 (2016). <https://doi.org/10.1007/s10817-015-9336-2>
8. Bock, P., Murawska, A., Bruni, A., Schürmann, C.: Representing session types (2016). <https://pure.itu.dk/en/publications/representing-session-types>, in Dale Miller’s Festschrift
9. Böhme, S., Nipkow, T.: Sledgehammer: judgement day. In: Giesl, J., Hähnle, R. (eds.) *IJCAR 2010. LNCS (LNAI)*, vol. 6173, pp. 107–121. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14203-1_9
10. Caires, L., Vieira, H.T.: Conversation types. *Theor. Comput. Sci.* **411**(51–52), 4399–4440 (2010). <https://doi.org/10.1016/j.tcs.2010.09.010>
11. Carbone, M., Montesi, F.: Deadlock-freedom-by-design: multiparty asynchronous global programming. In: *POPL 2013: Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 263–274. ACM, New York (2013). <https://doi.org/10.1145/2429069.2429101>
12. Castro, D., Ferreira, F., Yoshida, N.: EMTST: engineering the meta-theory of session types. In: Biere, A., Parker, D. (eds.) *TACAS 2020. LNCS*, vol. 12079, pp. 278–285. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45237-7_17
13. Castro-Perez, D., Ferreira, F., Gheri, L., Yoshida, N.: Zooid: a DSL for certified multiparty computation: from mechanised metatheory to certified multiparty processes. In: *PLDI 2021: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pp. 237–251. ACM, New York (2021). <https://doi.org/10.1145/3453483.3454041>
14. Chaudhuri, K., Cimini, M., Miller, D.: A lightweight formalization of the metatheory of bisimulation-up-to. In: Leroy, X., Tiu, A. (eds.) *CPP 2015: Proceedings of the 4th ACM SIGPLAN Conference on Certified Programs and Proofs*, pp. 157–166. ACM (2015). <https://doi.org/10.1145/2676724.2693170>
15. Chaudhuri, K., Lima, L., Reis, G.: Formalized meta-theory of sequent calculi for linear logics. *Theor. Comput. Sci.* **781**, 24–38 (2019). <https://doi.org/10.1016/j.tcs.2019.02.023>
16. Ciccone, L., Padovani, L.: A dependently typed linear π -calculus in Agda. In: *PPDP 2020: 22nd International Symposium on Principles and Practice of Declarative Programming*, pp. 8:1–8:14. ACM (2020). <https://doi.org/10.1145/3414080.3414109>
17. Cruz-Filipe, L., Montesi, F., Peressotti, M.: Formalising a turing-complete choreographic language in Coq. In: Cohen, L., Kaliszyk, C. (eds.) *ITP 2021: Proceedings of the 12th International Conference on Interactive Theorem Proving. Leibniz International Proceedings in Informatics, Dagstuhl, Germany*, vol. 193, pp. 15:1–15:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik (2021). <https://doi.org/10.4230/LIPIcs.ITP.2021.15>
18. Czajka, L., Kaliszyk, C.: Hammer for Coq: automation for dependent type theory. *J. Autom. Reason.* **61**(1–4), 423–453 (2018). <https://doi.org/10.1007/S10817-018-9458-4>
19. Danielsson, N.A.: Bag equivalence via a proof-relevant membership relation. In: Beringer, L., Felty, A. (eds.) *ITP 2012. LNCS*, vol. 7406, pp. 149–165. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32347-8_11

20. Dardha, O., Giachino, E., Sangiorgi, D.: Session types revisited. *Inf. Comput.* **256**, 253–286 (2017). <https://doi.org/10.1016/j.ic.2017.06.002>
21. Derakhshan, F., Pfenning, F.: Circular proofs as session-typed processes: a local validity condition. *Log. Methods Comput. Sci.* **18**(2) (2022). [https://doi.org/10.46298/LMCS-18\(2:8\)2022](https://doi.org/10.46298/LMCS-18(2:8)2022)
22. Despeyroux, J.: A higher-order specification of the π -calculus. In: van Leeuwen, J., Watanabe, O., Hagiya, M., Mosses, P.D., Ito, T. (eds.) *TCS 2000*. LNCS, vol. 1872, pp. 425–439. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-44929-9_30
23. Felty, A.P., Olarte, C., Xavier, B.: A focused linear logical framework and its application to metatheory of object logics. *Math. Struct. Comput. Sci.* **31**(3), 312–340 (2021). <https://doi.org/10.1017/S0960129521000323>
24. Gacek, A., Miller, D., Nadathur, G.: Nominal abstraction. *Inf. Comput.* **209**(1), 48–73 (2011). <https://doi.org/10.1016/J.IC.2010.09.004>
25. Gay, S., Hole, M.: Types and subtypes for client-server interactions. In: Swierstra, S.D. (ed.) *ESOP 1999*. LNCS, vol. 1576, pp. 74–90. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-49099-X_6
26. Gay, S.J., Hole, M.: Subtyping for session types in the pi calculus. *Acta Informatica* **42**(2–3), 191–225 (2005). <https://doi.org/10.1007/S00236-005-0177-Z>
27. Gay, S.J., Thiemann, P., Vasconcelos, V.T.: Duality of session types: the final cut. In: *Proceedings of the PLACES 2020*. *Electronic Proceedings in Theoretical Computer Science*, vol. 314, pp. 23–33. Open Publishing Association (2020). <https://doi.org/10.4204/eptcs.314.3>
28. Gorla, D.: Towards a unified approach to encodability and separation results for process calculi. *Inf. Comput.* **208**(9), 1031–1053 (2010). <https://doi.org/10.1016/j.ic.2010.05.002>
29. Hirsch, A.K., Garg, D.: Pirouette: Higher-order typed functional choreographies. *Proc. ACM Program. Lang.* **6** (2022). <https://doi.org/10.1145/3498684>
30. Hirschhoff, D.: A full formalisation of π -calculus theory in the calculus of constructions. In: Gunter, E.L., Felty, A. (eds.) *TPHOLs 1997*. LNCS, vol. 1275, pp. 153–169. Springer, Heidelberg (1997). <https://doi.org/10.1007/BFb0028392>
31. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. *J. ACM* **63**(1) (2016). <https://doi.org/10.1145/2827695>
32. Honsell, F., Miculan, M., Scagnetto, I.: π -calculus in (co)inductive-type theory. *Theor. Comput. Sci.* **253**(2), 239–285 (2001). [https://doi.org/10.1016/S0304-3975\(00\)00095-5](https://doi.org/10.1016/S0304-3975(00)00095-5)
33. Horne, R., Padovani, L.: A logical account of subtyping for session types. In: Castellani, I., Scalas, A. (eds.) *Proceedings of the 14th Workshop on Programming Language Approaches to Concurrency and Communication-Centric Software*. *EPTCS*, vol. 378, pp. 26–37. Open Publishing Association (2023). <https://doi.org/10.4204/EPTCS.378.3>
34. Hur, C.K., Neis, G., Dreyer, D., Vafeiadis, V.: The power of parameterization in coinductive proof. In: *POPL 2013: Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 193–206. ACM, New York (2013). <https://doi.org/10.1145/2429069.2429093>
35. Kahsai, T., Miculan, M.: Implementing Spi calculus using nominal techniques. In: Beckmann, A., Dimitracopoulos, C., Löwe, B. (eds.) *CiE 2008*. LNCS, vol. 5028, pp. 294–305. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-69407-6_33

36. Maksimović, P., Schmitt, A.: HOCore in Coq. In: Urban, C., Zhang, X. (eds.) ITP 2015. LNCS, vol. 9236, pp. 278–293. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-22102-1_19
37. Melham, T.F.: A mechanized theory of the π -calculus in HOL. *Nordic J. Comput.* **1**(1), 50–76 (1994)
38. Perera, R., Cheney, J.: Proof-relevant π -calculus: a constructive account of concurrency and causality. *Math. Struct. Comput. Sci.* **28**(9), 1541–1577 (2018). <https://doi.org/10.1017/S096012951700010X>
39. Pierce, B.C. (ed.): *Advanced Topics in Types and Programming Languages*. MIT Press, London (2004)
40. Pohjola, J., Gómez-Londoño, A., Shaker, J., Norrish, M.: Kalas: a verified, end-to-end compiler for a choreographic language. In: Andronick, J., de Moura, L. (eds.) ITP 2022: Proceedings of the 13th International Conference on Interactive Theorem Proving. Leibniz International Proceedings in Informatics, Dagstuhl, Germany, vol. 237, pp. 27:1–27:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik (2022). <https://doi.org/10.4230/LIPIcs.ITP.2022.27>
41. Pous, D.: Coinduction all the way up. In: Grohe, M., Koskinen, E., Shankar, N. (eds.) Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2016, New York, NY, USA, 5–8 July 2016, pp. 307–316. ACM (2016). <https://doi.org/10.1145/2933575.2934564>
42. Rouvoet, A., Poulsen, C.B., Krebbers, R., Visser, E.: Intrinsically-typed definitional interpreters for linear, session-typed languages. In: Proceedings of the CPP 2020, pp. 284–298. ACM (2020). <https://doi.org/10.1145/3372885.3373818>
43. Sangiorgi, D., Walker, D.: *The π -calculus: A Theory of Mobile Processes*. Cambridge University Press, Cambridge (2001)
44. Sano, C., Kavanagh, R., Pientka, B.: Mechanizing session-types using a structural view: enforcing linearity without linearity. *Proc. ACM Program. Lang.* **7**(OOPSLA), 235:374–235:399 (2023). <https://doi.org/10.1145/3622810>
45. Scalas, A., Yoshida, N.: Less is more: Multiparty session types revisited. *Proc. ACM Program. Lang.* **3** (2019). <https://doi.org/10.1145/3290343>
46. Schack-Nielsen, A., Schürmann, C.: Celf – a logical framework for deductive and concurrent systems (system description). In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 320–326. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-71070-7_28
47. Thiemann, P.: Intrinsically-typed mechanized semantics for session types. In: Proceedings of the 21st International Symposium on Principles and Practice of Declarative Programming, PPDP 2019. ACM, New York (2019). <https://doi.org/10.1145/3354166.3354184>
48. Tiu, A., Miller, D.: Proof search specifications of bisimulation and modal logics for the π -calculus. *ACM Trans. Comput. Logic* **11**(2) (2010). <https://doi.org/10.1145/1656242.1656248>
49. Vasconcelos, V.T.: Fundamentals of session types. *Inf. Comput.* **217**, 52–70 (2012). <https://doi.org/10.1016/j.ic.2012.05.002>
50. Xia, L.Y., et al.: Interaction trees: representing recursive and impure programs in Coq. *Proc. ACM Program. Lang.* **4** (2019). <https://doi.org/10.1145/3371119>
51. Zalakain, U., Dardha, O.: π with leftovers: a mechanisation in Agda. In: Peters, K., Willemse, T.A.C. (eds.) FORTE 2021. LNCS, vol. 12719, pp. 157–174. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-78089-0_9

List of Publications Reproduced in this Thesis

SeCaV: A Sequent Calculus Verifier in Isabelle/HOL

This article was first published in Mauricio Ayala-Rincon and Eduardo Bonelli (eds): Proceedings 16th Logical and Semantic Frameworks with Applications (LSFA 2021), Buenos Aires, Argentina (Online), 23rd – 24th July, 2021, Electronic Proceedings in Theoretical Computer Science 357, pp. 38–55.

The version of record is available online at: <https://doi.org/10.4204/EPTCS.357.4>

Using Isabelle in Two Courses on Logic and Automated Reasoning

This article was first published in Ferreira, J.F., Mendes, A., Menghi, C. (eds): Formal Methods Teaching. FMTea 2021. Lecture Notes in Computer Science, volume 13122, pp. 117–132, by Springer, Cham. The article is reproduced with permission from Springer Nature.

The version of record is available online at: https://doi.org/10.1007/978-3-030-91550-6_9

Teaching Functional Programmers Logic and Metatheory

This article was first published in Peter Achten and Elena Machkasova (eds): Proceedings Tenth and Eleventh International Workshop on Trends in Functional Programming In Education (TFPIE 2021/22), Kraków, Poland (online), 16th February 2021/16th March 2022, Electronic Proceedings in Theoretical Computer Science 363, pp. 74–92.

The version of record is available online at: <https://doi.org/10.4204/EPTCS.363.5>

On Exams with the Isabelle Proof Assistant

This article was first published in Pedro Quaresma, João Marcos and Walther Neuper (eds): Proceedings 11th International Workshop on Theorem Proving Components for Educational Software (ThEdu'22), Haifa, Israel, 11 August 2022, Electronic Proceedings in Theoretical Computer Science 375, pp. 63–76.

The version of record is available online at: <https://doi.org/10.4204/EPTCS.375.6>

ProofBuddy: A Proof Assistant for Learning and Monitoring

This article was first published in Elena Machkasova (ed): Proceedings Twelfth International Workshop on Trends in Functional Programming in Education (TFPIE 2023), Boston, Massachusetts, USA, 12th January 2023, Electronic Proceedings in Theoretical Computer Science 382, pp. 1–21.

The version of record is available online at: <https://doi.org/10.4204/EPTCS.382.1>

Verifying a Sequent Calculus Prover for First-Order Logic with Functions in Isabelle/HOL

This article was first published in the Journal of Automated Reasoning, volume 68, article number 15, 2024, by Springer Nature.

The version of record is available online at: <https://doi.org/10.1007/s10817-024-09697-3>

The Concurrent Calculi Formalisation Benchmark

This article was first published in Castellani, I., Tiezzi, F. (eds): Lecture Notes in Computer Science, volume 14676, pp. 149–158, 2024, by Springer Nature. The article is reproduced with permission from Springer Nature.

The version of record is available online at: https://doi.org/10.1007/978-3-031-62697-5_9

Formal methods for software engineering make it possible to verify the correctness of software using logical and mathematical techniques. Unfortunately, formal methods have a reputation for being difficult to understand and use. This thesis investigates approaches to improving the ease of use and usefulness of formal methods.

The thesis has a focus on proof assistants and mechanization of proofs about concurrent calculi, and investigates three research questions:

- What are effective approaches to learning proof competence with computer assistance?
- How can we improve the helpfulness of proof assistants and related tools?
- What is the state of the art in efficiently mechanizing proofs about concurrent calculi, and how can we improve it?

Towards answering these questions, this thesis contains:

- A sequent calculus verifier for learning proof competence, leveraging the proof assistant Isabelle to check proofs.
- An overview of the computer science curriculum at the Technical University of Denmark, contextualizing the situations in which students learn proof competence.
- A study providing preliminary evidence for the efficacy of learning logic and proof competence via functional programming in a proof assistant.
- A tutorial-style investigation of approaches to testing learning outcomes with proof assistants in the context of written exams.
- An instrumented proof assistant for conducting studies of the efficacy of using proof assistants to facilitate learning of proof competence.
- An automated sequent calculus theorem prover for first-order logic with functions, with a formally verified soundness and completeness proof that considers the search strategy of the prover.
- A set of benchmark challenges addressing three key issues for formal reasoning about concurrent calculi.

The thesis thus provides a broad foundational contribution towards rebuilding the reputation of formal methods in software engineering.

Technical
University of
Denmark

DTU Compute
Richard Petersens Plads, Building 324
2800 Kgs. Lyngby

www.compute.dtu.dk