

# Formalization of Logical Systems in Isabelle

An automated theorem prover for the Sequent Calculus Verifier

Frederik Krogsdal Jacobsen

Master's Thesis

## **Formalization of Logical Systems in Isabelle**

An automated theorem prover for the Sequent Calculus Verifier

Master's Thesis

June 2021

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.

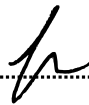
Cover photo:    Arvind Shakya, 2017

Published by:   DTU, Department of Applied Mathematics and Computer Science,  
Richard Petersens Plads, Building 324, 2800 Kgs. Lyngby, Denmark  
[www.compute.dtu.dk](http://www.compute.dtu.dk)

## Preface

This thesis has been prepared from January 25th to June 25th of 2021 at the Section for Algorithms, Logic and Graphs at the Department of Applied Mathematics and Computer Science at the Technical University of Denmark, DTU, granting 30 ECTS points in partial fulfilment of the requirements for the degree of Master of Science in Computer Science and Engineering.

Frederik Krogsdal Jacobsen - s163949



.....  
*Signature*

25/06/2021

.....  
*Date*

## Abstract

We describe the design and implementation of an automated theorem prover for the proof system of the Sequent Calculus Verifier. The prover is designed to generate “natural” proofs in a one-sided sequent calculus, and the algorithm underlying the prover is a formalization of an intuitive approach to proving formulas. The automated theorem prover is implemented in Isabelle/HOL and Haskell, and we additionally present an unfinished proof of completeness of the prover and a sketch of a proof of soundness in Isabelle/HOL. Both the prover and the proofs about it are useful as a learning tool for students to experiment with and see how properties about programs can be proven. While formalized proofs of soundness and completeness are unfinished, confidence that the prover works correctly has been obtained through automated tests.

Die Logik muss für sich selber sorgen.

[...]

Wir können uns, in gewissem Sinne, nicht in der Logik irren.

---

*Tractatus Logico-Philosophicus*  
LUDWIG WITTGENSTEIN

A mathematician is a machine for turning coffee into theorems.  
It follows by duality that a comathematician is a machine for turning cotheorems into ffee.

---

ALFRÉD RÉNYI and UNKNOWN

## Acknowledgements

I would like to thank my advisors Jørgen Villadsen, Asta Halkjær From, and Dmitriy Traytel for many helpful discussions, pointers and comments. I would also like to thank Robin Andreas Clausen Swann and Alexander Birch Jensen and for comments on a draft.

I extend my gratitude to the developers of Isabelle, Haskell, the Archive of Formal Proof, and the Haskell ecosystem of libraries, without which this project would not have been possible.

Many thanks to Kultureliten, everyone at KK400, my family, an anonymous intern at the To Øl brewery, and everyone else who has had my back in some way or another.

A very special blessing—wherever you may be—to Charlotte, Chinua, Damini, Dan, Dave, Elvir, Jacob, Johanne, John, Josefine, Magnus, Maya, Mikayla, Ryan, Thomas, Thor, Tom, Tommy, Veronika Katinka, and Vladimir for letting me in on your ideas in a pleasantly distracting manner.

Apologies to anyone forgotten – I'll fix you a pint on the other side.

# Contents

Preface . . . . .	ii
Abstract . . . . .	iii
Acknowledgements . . . . .	iv
<b>1 Introduction</b>	<b>1</b>
1.1 Related work . . . . .	3
<b>2 Background</b>	<b>6</b>
2.1 First-order logic . . . . .	6
2.2 Induction and coinduction . . . . .	8
2.3 Mechanized proof . . . . .	11
2.4 The Sequent Calculus Verifier . . . . .	13
2.5 Abstract frameworks for completeness and soundness . . . . .	16
<b>3 Prover</b>	<b>18</b>
3.1 The proof search algorithm . . . . .	18
3.2 Implementing the algorithm . . . . .	25
3.3 The Haskell program . . . . .	31
<b>4 Soundness</b>	<b>35</b>
<b>5 Completeness</b>	<b>37</b>
5.1 Enabledness . . . . .	37
5.2 Persistency . . . . .	41
5.3 Putting it all together . . . . .	55
<b>6 Results and discussion</b>	<b>58</b>
6.1 Example proofs . . . . .	58
6.2 Performance . . . . .	61
6.3 Tests . . . . .	61
6.4 Limitations in the prover . . . . .	62
6.5 Missing proofs . . . . .	63
6.6 Sources of complexity . . . . .	64
<b>7 Conclusion</b>	<b>65</b>
<b>Bibliography</b>	<b>66</b>

# 1 Introduction

Logic can be used to formally reason about mathematics, software, and almost any other topic. In some sense, the foundation of logic is propositional logic.

Propositional logic allows reasoning about propositions and relations between them. A proposition can be either true or false, and more complicated propositions can be constructed using logical connectives such as conjunction, disjunction, negation, or implication. Propositional logic allows formalization of arguments such as:

**Premise 1** If it is raining, then the ground is wet.

**Premise 2** It is raining.

**Conclusion** The ground is wet.

The conclusion obviously follows from the two premises, but we would like to formalize the reasoning used in the argument. If we let the symbol  $P$  stand for the proposition “it is raining”, and the symbol  $Q$  stand for the proposition “the ground is wet”, we can write the argument above as follows:

**Premise 1**  $P \implies Q$

**Premise 2**  $P$

**Conclusion**  $Q$

Here, the symbol  $\implies$  means implication, and premise 1 can be read “if  $P$ , then  $Q$ ”. The line of reasoning used for this argument is the *inference rule of modus ponens*, which may formally be written as follows:

$$\frac{P \implies Q \quad P}{Q} \text{MP}$$

This notation should be read as follows: “if the propositions above the line are true, so is the proposition below the line”.

This example is almost silly in its simplicity, but for more complicated propositions, it is rarely so obvious how to reason correctly. The situation gets even worse when we introduce the ability to *quantify over variables*, which results in what is commonly called *first-order logic* or *predicate logic*. Predicate logic allows formalization of arguments such as:

**Premise 1** Any day on which it rains is a day on which the ground is wet

**Premise 2** There is a day on which it rains

**Conclusion** There is a day on which the ground is wet

This is allowed by the introduction of *quantifiers*, of which there are typically two, and *predicates*, which are like propositions, except they are allowed to depend on variables. The quantifier symbol  $\forall$  is read “for all”, and the quantifier symbol  $\exists$  is read “there exists”.

Using these symbols, and letting the symbols  $P$  and  $Q$  now be predicates representing the state of rain and wetness on specific days, we can formalize the argument above as follows:

**Premise 1**  $\forall x.P(x) \implies Q(x)$

**Premise 2**  $\exists x.P(x)$

**Conclusion**  $\exists x.Q(x)$

This argument may be read:

**Premise 1** for all days  $x$ : if it rains on day  $x$ , then the ground is wet on day  $x$

**Premise 2** there is some day  $x$  on which it rains

**Conclusion** there is some day  $x$  on which the ground is wet

The argument works because Premise 2 “produces” a name of a day on which it rains, which may then be “plugged into” Premise 1 to obtain the conclusion. This is essentially the same argument as in the propositional case above, but now we have the additional task of managing the variable.

Managing a single variable is easy, but managing a million variables—or in many cases, even just three or four—is not. It is very easy to accidentally mix up the variables, or forget which variables and names even exist in the first place. For even slightly larger examples than the ones shown above, it can consequently be very difficult to prove that some relationships hold. Even worse, verifying whether a proof is correct can be almost as difficult (and much more boring!). This can make learning and using the rules of logic a gruelling endeavour for students.

One solution to this problem is to lift the burden of verifying proofs from the shoulders of the student by making a computer system do the boring work. This can be accomplished by the use of a *proof assistant*, which is a computer program designed to help humans by checking that each rule is applied properly. Proof assistants often also have many more features such as automatically determining which parts of the conclusion have not yet been proven, automated proof search, and the ability to generate computer programs out of the parts of proofs that correspond to computations.

*Isabelle* is a generic proof assistant in the sense that it allows reasoning using a number of *logical frameworks*. The most popular of these is *higher-order logic*, which gives rise to the Isabelle/HOL system. This system has been used to prove various theorems and properties in a wide variety of projects. Unfortunately, the wide applicability of the Isabelle proof assistant means that it is quite complex and difficult to learn. For beginners, a simpler system is easier to use.

One such system is the *Sequent Calculus Verifier* (SeCaV), which has been developed specifically for students to learn logic. SeCaV is embedded in Isabelle/HOL, but is much less powerful and thus much simpler to use, while still being able to leverage the automated tools of Isabelle to verify that proofs are correct, or point out mistakes in incorrect ones.

A popular feature of Isabelle/HOL is the *Sledgehammer* tool, which allows users of the proof assistant to feed their premises and conclusions to a number of automated provers such as Vampire, SPASS, and the E prover. These provers can then attempt to find a proof. Unfortunately, these provers are optimized such that they typically generate proofs which are essentially incomprehensible to human minds, and especially to beginners. It is thus often difficult to understand why a proof works, and what the exact steps actually are.

The purpose of this project is to develop an automated theorem prover for the Sequent Calculus Verifier, designed in such a way that it produces “natural” proofs. This notion of naturality is not precise, but essentially means that the proofs look similar to the proofs a



human being might write, and especially that the generated proofs and the proof search procedure can be understood by students.

Additionally, the prover should in some sense be *complete*, such that it will—given enough time—find a proof of any provable formula. To prove this property, the prover will instance a general framework of *abstract completeness* on coinductive proof trees. What this precisely means will be explained later on. The main idea is that the prover will search for a proof by generating a possibly infinite tree of rule applications, from which a proof can be extracted if and only if the tree turns out to be finite.

This prover can be used by students to prove more complicated formulas than what is feasible by hand or with machine assistance, and the search procedure and the associated proofs can be studied to see how the intuition used when proving formulas by hand can be organized into a complete search procedure.

## 1.1 Related work

This project builds on top the Sequent Calculus Verifier and the abstract completeness framework for provers. The next two sections will discuss other projects related to these two projects.

Since this project concerns the implementation of an automated theorem prover, we will also locate the project in the contexts of automated theorem proving in general, automated theorem proving in systems with “natural” proofs, and formal verification of logical systems. We will see that the project is uniquely positioned, in that it concerns a formalized automated theorem prover for a “natural” proof system.

### 1.1.1 SeCaV

The Sequent Calculus Verifier is already a well established proof system, and both soundness and completeness have been proven for the system [1]. The system has also been used to teach students in a course at the Technical University of Denmark [2].

To simplify the use of the system, an online tool called the SeCaV Unshortener has been developed. This tool allows students to input proofs in a very simple format, after which they are automatically translated to Isabelle proofs, which can then be verified [3].

### 1.1.2 Abstract soundness and completeness

The abstract completeness framework developed by Blanchette, Popescu and Traytel [4] [5] contains a simple example application in the form of an automated theorem prover for propositional logic. The original application of the framework was the mechanization of the metatheory of the Sledgehammer tool [6]. The ideas of the framework have also been used to formalize the proof system of the Incredible Proof Machine, which will be explained in more detail in the next section.

### 1.1.3 Finding natural proofs

There are a large number of existing proof systems and tools meant to allow “natural” proofs. Most of these systems are based on either natural deduction or sequent calculi. We will mention only a small selection of related systems.

NaDeA (Natural Deduction Assistant) [7] is a web application which 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.

Logitext (logitext.mit.edu) is a web application that allows users to prove sequents in a sequent calculus system. The system works by allowing users to click the connective they want to eliminate, thus making the actual proof rules of the system implicit. AXolotl [8] is an Android app designed to allow self-study of proof techniques in a number of logical calculi. The system restricts itself to the quantifier-free fragment of first-order logic and is thus significantly less powerful than SeCaV.

The Incredible Proof Machine [9] is a web application that allows users to create proofs using a specialized graphical interface. The proof system has been shown to be as strong as natural deduction, and a model of the system has been formalized in Isabelle as mentioned above.

None of the systems above include automated theorem provers; they are essentially simple proof assistants designed to aid students in understanding logical systems.

The Sequent Calculus Trainer [10] [11] is a tool designed to aid students in learning proof techniques for a sequent calculus. The tool distinguishes itself by including an automated theorem prover that constantly runs in the background, so that students can in many cases immediately be informed if they apply a proof rule that makes their sequent unprovable. The logic of the system is a first-order logic with equality.

THINKER [12] 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 “naturalness” in mind, as a reaction to the indirectness of resolution-based proof systems. The author argues that using resolution-based systems obscures the difficulties of proving a formula and was, at the time, actually less efficient than even the most naïve human proof strategies in a natural deduction system.

MUSCADET [13] 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.

#### **1.1.4 Formal verification of automated theorem provers**

While there are many very advanced automated theorem provers, few attempts at formal verification have been made. This probably owes mostly to the complexity of modern automated theorem provers, and to the fact that most theorem provers are oriented towards practical use and are tested on numerous problems in e.g. the TPTP Problem Library [14].

As a first step towards formally verifying modern provers, an ordered resolution prover for first-order logic has been verified in Isabelle/HOL [15].

A prover for first-order logic with equality has also been formally verified in Isabelle/HOL [16]. The prover is for a declarative system, but also includes a prover for a tableau system, allowing intermixing of declarative-style proofs and proofs by tableaux. The verification of the prover is based on a small verified kernel on top of which the rest of the prover is implemented.

A simple prover for first-order logic has also been formalized in Isabelle/HOL with the

aim of allowing students to understand both the prover and the formalization [17]. This work was based on an earlier formalization [18], but simplified both the prover and the proofs to enable easier understanding by students.

## 2 Background

This section will introduce the background necessary to understand how the automated theorem prover works and why it is useful. The prover and the proofs about it integrate logic, programming, and automated theorem proving, so the contents of this section range quite widely. Some readers may already be knowledgeable about some of the topics covered in this section, and may prefer to skip the subsections concerning these topics. Sections 2.4 and 2.5 contain definitions specific to SeCaV and the abstract completeness framework, and should not be skipped.

We will begin by introducing first-order logic and some theoretical properties of it. The abstract completeness framework uses a coinductive approach, so the next topic is induction and coinduction. Next is an introduction to mechanized proof including why such an approach is useful, and why we have chosen the Isabelle/HOL proof assistant for this project. Finally, we introduce the Sequent Calculus Verifier which is the concrete system we will be implementing an automated theorem prover for, and the abstract completeness framework which we use to implement the automated theorem prover.

### 2.1 First-order logic

First-order logic extends propositional logic by allowing quantification over a specified domain and the use of variables with values in this domain [19]. The quantifiers make it possible to construct logical expressions of the form “there is an object such that ...” and “for all objects it holds that ...” with variables representing these quantified objects. The quantifier “there is an object” is represented by the symbol  $\exists$ , while the quantifier “for all objects” is represented by the symbol  $\forall$ .

We will consider a logic with only one infinite set of non-logical symbols. As such, for every natural number  $n$  there is an infinite set of  $n$ -ary predicate symbols representing relations between  $n$  elements. We may identify predicate symbols with valence 0 with propositional variables. Additionally, for every natural number  $n$  there is an infinite set of  $n$ -ary function symbols which represent functions of  $n$  elements. Function symbols of arity 0 are constants which represent specific elements of the domain. Our logic will not include a notion of equality.

The set of terms of the logic is then defined as containing the set of variables and any expression consisting of an  $n$ -ary function symbol applied to  $n$  terms. The set of formulas of the logic is defined as containing any expression consisting of an  $n$ -ary predicate symbol applied to  $n$  terms, any formula quantified by  $\exists$  or  $\forall$ , and any expression consisting of formulas connected by logical quantifiers such as  $\wedge$ ,  $\vee$ ,  $\neg$ ,  $\implies$  and so on. In a formula, a variable is called bound if it is quantified and free if it is not. A formula with no free variables is called a sentence.

A structure specifies a (non-empty) domain for the quantifiers to range over and an interpretation of each non-logical symbol such that each term is assigned an object to represent, each predicate symbol is assigned a property of objects to represent, and each sentence is assigned a truth value.

Given an interpretation, a sentence can be evaluated to be either true or false. If an interpretation makes a sentence true, the interpretation is said to satisfy the sentence, and the

interpretation itself is then called a model of the sentence. Conversely, an interpretation which makes a sentence false is called a countermodel. A sentence is satisfiable if there is an interpretation which makes it true. A sentence is called valid if it is true under every interpretation. A valid sentence is also called a *theorem*. Conversely, a sentence is unsatisfiable if it is false under every interpretation and invalid if there is an interpretation which makes it false.

### 2.1.1 Deductive systems

To prove that a formula is valid, we need a deductive system in which to formally encode our proof. There are many deductive systems for first-order logic, including natural deduction [20], the method of tableaux [21], resolution [22], and Hilbert calculi [23] [24].

We will use a sequent calculus [20] as our deductive system since we are designing an automated theorem prover for the Sequent Calculus Verifier. A sequent calculus works with multiple formulas at a time, with the collection of formulas known as a sequent. A two-sided sequent calculus consists of a conjunction of formulas which serve as “assumptions” and a disjunction of formulas which serve as “goals”. A one-sided sequent calculus merges the two parts of the sequent by negating “assumptions” such that there are only “goals”.

In a sequent calculus, sequents are manipulated by application of proof rules. Some proof rules may generate multiple branches that need to be proven separately, while some proof rules finish branches of the proof when applied.

### 2.1.2 Soundness

If any formula that can be derived in a deductive system is valid, the system is called sound. Proving that a deductive system is sound is typically quite simple, since all that is needed is to prove that the formulas in each sequent logically follow from the formulas of the previous sequent. This can be done by induction on the number of proof rules applied, showing that some proof rule results in the next sequent for all possible sequents in the inductive step.

### 2.1.3 Completeness

If every valid formula can be derived in a deductive system, the system is called complete. For first-order logic, it is possible to construct deductive systems which are both sound and complete [25].

Completeness of a deductive system is typically much harder to show than soundness. This is complicated further by the fact that we need to not only show that every valid formula *can* be derived, but also *how* to do so in order to obtain an automated theorem prover.

One approach, described in [5], is to first show that the prover will either terminate producing a finite proof or produce an infinite “failed proof” when given a formula  $p$ . Next, we show that it is possible to extract a countermodel for  $p$  from an infinite “failed proof”. Now assume that the prover produced an infinite “failed proof” for a valid formula  $p$ . It is then possible to extract a countermodel for  $p$ , which contradicts the fact that  $p$  is valid since the deductive system is sound (which must of course first be proven separately). By the principle of excluded middle it must then be the case that the prover produces a finite proof for  $p$ . Since  $p$  was an arbitrary valid formula, we have then proven that every valid

formula can be derived in the deductive system using the algorithm implemented in the prover.

### 2.1.4 Automated theorem provers

The goal of this project is to construct an automated theorem prover for a first-order logic. Automated theorem provers are of course useful for attempting to prove mathematical or logical theorems as implemented in e.g. the Sledgehammer tool for Isabelle [6], but they are also useful for a range of verification problems. As an example, floating point units in integrated circuits are often verified using automated theorem provers to verify that the circuits correctly implement mathematical operations such as division, which have previously caused problems such as the Pentium FDIV bug [26].

Most automated theorem provers are very focused on performance, since modern verification problems can be extremely large. Unfortunately, this often means that understanding how a given prover works is quite difficult, since they integrate multiple phases such as preprocessing into normal forms and various heuristics for optimizing proof search.

Additionally, many automated theorem provers such as Vampire [27], the E prover [28], iProver [29], and Prover9 [30] use deductive systems based on resolution, which can make it quite difficult for students to understand the found proofs. Using a sequent calculus as our deductive system guarantees that proofs will be readable by humans, since each individual rule is quite simple to understand and proofs are organized in a linear order with a clear and simple relationship between each sequent.

## 2.2 Induction and coinduction

Many of the proofs needed for this project, such as those of soundness and completeness, rely on the method of induction. The abstract completeness framework used for the prover uses a coinductive datastructure to represent potentially infinite proof trees, which is necessary because the prover may not terminate. Coinductive techniques are generally applicable for infinite datastructures and potentially infinite computations, but they are also less intuitive and less widely understood than inductive techniques. The next sections contain a very brief overview of induction and coinduction. More information may be found in e.g. [31].

### 2.2.1 What is induction, really?

The proof technique of mathematical induction is widely known: if we can prove a property  $P$  for the base case  $P(0)$  and prove that  $P(n)$  implies  $P(n + 1)$ , we obtain a “domino effect” through the natural numbers since  $P(0)$  implies  $P(1)$ , which in turn implies  $P(2)$ , and so on. But in fact this technique is much more general and applies to any recursively defined structure. In particular, we can use the technique to prove properties of proof rules, terms, and formulas, and datastructures such as recursively defined trees and lists.

More formally, induction can be captured by the notion of initial algebras of functors, which may be considered as a generalization of the concept of least fixed points of monotone functions. Consider for example the natural numbers defined inductively by a zero function  $0 : \text{Unit} \rightarrow \mathbb{N}$  and a successor function  $\text{succ} : \mathbb{N} \rightarrow \mathbb{N}$ . These functions can be combined into a function  $[0, \text{succ}] : \text{Unit} + \mathbb{N} \rightarrow \mathbb{N}$  which forms the initial algebra of the functor  $T(X) = \text{Unit} + X$ .

Another example is the type  $A^*$  of lists of elements of some type  $A$  which can be captured by the empty list  $\text{nil} : \text{Unit} \rightarrow A^*$  and the cons function  $\text{cons} : A \times A^* \rightarrow A^*$ . These functions can be combined into a function  $[\text{nil}, \text{cons}] : \text{Unit} + (A \times A^*) \rightarrow A^*$  which is the initial algebra of the functor  $T(X) = \text{Unit} + (A \times X)$ .

Using these two definitions, we can define e.g. the length function on lists by induction. The function has to arise from the algebra  $\text{Unit} + A \times \mathbb{N} \rightarrow \mathbb{N}$  which is the cotuple of the zero function  $0 : \text{Unit} \rightarrow \mathbb{N}$  and the function  $\text{succ} \circ \pi_2 : A \times \mathbb{N} \rightarrow \mathbb{N}$  which combines the successor function and the second projection function on pairs. This function is defined by the clauses

$$\begin{aligned} \text{length}(\text{nil}) &= 0 \\ \text{length}(\text{cons}(a, b)) &= \text{succ}(\text{length}(b)) = \text{succ}(\text{length}(\pi_2(a, b))) \\ &= \text{succ}(\pi_2(\text{id} \times \text{length})(a, b)). \end{aligned}$$

The length function is then defined as the (unique) function in the following initiality diagram:

$$\begin{array}{ccc} \text{Unit} + (A \times A^*) & \xrightarrow{\text{id} + (\text{id} \times \text{length})} & \text{Unit} + (A \times \mathbb{N}) \\ \downarrow [\text{nil}, \text{cons}] \cong & & \downarrow [0, \text{succ} \circ \pi_2] \\ A^* & \xrightarrow{\text{length}} & \mathbb{N} \end{array}$$

We can then do proofs by induction by exploiting the uniqueness of functions out of initial algebras. Consider for example the “replicate2” function which doubles the length of a list by repeating each element twice and can be defined inductively:

$$\begin{aligned} \text{replicate2}(\text{nil}) &= \text{nil} \\ \text{replicate2}(\text{cons}(a, b)) &= \text{cons}(a, \text{cons}(a, \text{replicate2}(b))) \end{aligned}$$

We can show that the length of the list returned by this function is twice the length of the original list (i.e. that  $\text{length}(\text{replicate2}(l)) = 2 \cdot \text{length}(l)$ ) by a simple proof by induction. But defining our functions by initiality, we can prove this in an even simpler and more precise way by showing that both sides of the equation are homomorphisms from the initial algebra  $(A^*, [\text{nil}, \text{cons}])$  to the algebra  $(\mathbb{N}, [0, \text{succ} \circ \text{succ} \circ \pi_2])$ . Since functions out of initial algebras are unique, the two sides of the equation must then be equal. This type of proof generalizes nicely to any algebraic datatype, since all we need to know to use the technique is the initial algebra associated to the functor describing the datatype. For most inductive datatypes, this is trivially done by translating the constructors of the datatype into a polynomial functor.

Another important advantage is that this approach easily dualizes to the notion of definitions and proofs by coinduction by replacing initial algebras with final coalgebras.

## 2.2.2 Coinduction

While induction lets us define and prove properties about finite structures defined by a number of constructors, coinduction lets us define and prove properties about infinite structures defined by a number of destructors. Finite lists, for example, can be defined in terms of a constructor for the empty list and a constructor adding an element to an existing list, as we have seen in the previous section. Infinite lists, also known as streams, can dually be defined in terms of a destructor taking the first element of the list and a destructor taking the remaining (potentially infinite) list. The basic idea is that, instead

of constructing objects by adding elements, we are destructing objects by removing elements.

Just as induction can be obtained in terms of initial algebras, coinduction can be obtained in terms of final coalgebras, which may be considered as a generalization of the concept of greatest fixed points of monotone functions. A coalgebra is the dual of an algebra in the sense that the arrows in the functions and diagrams are reversed. While functions out of initial algebras are unique, functions into final coalgebras are unique. Final coalgebras thus lead to the proof principle of coinduction, which is in this sense dual to the principle of induction.

Let us return to the example of the datatype  $A^{\mathbb{N}}$  of infinite lists of element of type  $A$ . The datatype can be defined by the destructors

$$\begin{aligned}\text{head}(s) &= s(0) \\ \text{tail}(s) &= \lambda x.s(x + 1)\end{aligned}$$

This datatype is the final coalgebra of the functor  $T(X) = A \times X$  with coalgebra structure  $\langle \text{head}, \text{tail} \rangle : A^{\mathbb{N}} \rightarrow A \times A^{\mathbb{N}}$ . Once we know that  $A^{\mathbb{N}}$  does carry this final coalgebra structure, the finality of the structure can be used to define functions into  $A^{\mathbb{N}}$  just as initiality was used to define functions out of initial algebras.

One datastructure we will need is a potentially infinite tree with labelled nodes, so we will use it as an example. Consider the functor  $T(X) = \text{fset}(X)$  where  $\text{fset}$  is the type of finite sets. It takes a function  $f : X \rightarrow Y$  and turns it into a function with signature  $\text{fset}(X) \rightarrow \text{fset}(Y)$  which takes  $\langle (x_1), \dots, (x_n) \rangle$  to  $\langle (f(x_1)), \dots, (f(x_n)) \rangle$ . A function  $X \rightarrow \text{fset}(X)$  mapping a state in  $X$  to a finite set of successor states in  $X$  is a coalgebra of this functor.

We will use such a coalgebra to describe a potentially infinite proof tree for our sequent calculus by setting  $X$  to be the set of possible proof states (i.e. sequents and any other information necessary for proofs). Each node in the proof tree will have a finite set of successors, which will represent the branches generated by applying a proof rule to the sequent that labels the node. We need this potentially infinite tree because our prover will not terminate when called with an unprovable formula, but instead produce an infinite “failed proof”.

### 2.2.3 Coinduction and functional programming

Most functional programming languages integrate inductive definitions and recursive functions as core parts of the language. Unfortunately, very few languages support coinductive definitions and corecursion as well.

Haskell [32] is a pure functional programming language which takes the approach of *lazy* datatypes. These types are neither inductive nor coinductive, but can be used to implement both, and Haskell does not distinguish data and codata. The idea is that terms are not fully evaluated until it is necessary to do so to return a result to the user. This means that e.g. an infinite list can be represented as a lazy datatype and treated as an inductive datatype so long as the entire list is never needed. Simultaneously, coinductive functions can be used to manipulate the infinite list, which then acts as a coinductive datatype. This allows programmers to easily work with coinductive and inductive datatypes and recursive and corecursive functions.

The Isabelle proof assistant [33], which will be used to implement most of the prover and all of the proofs about it, also supports both inductive and coinductive definitions and



recursive and corecursive functions. Isabelle does distinguish between data and codata, but allows function definitions that mix recursion and corecursion.

## 2.3 Mechanized proof

This project is not only about designing an automated theorem prover and proving it correct, but also about *mechanically verifying* that the proofs of e.g. soundness and completeness are *correct*. This is done using the Isabelle proof assistant [33], which can be used with higher-order logic (HOL) to prove properties about programs written in the accompanying programming language ML. The Isabelle proof assistant can then automatically verify that the user-supplied proofs are correct. But why is this useful, why should we trust Isabelle more than a human reviewer, and why not use a different proof assistant?

### 2.3.1 Why mechanize proofs?

When mathematicians and logicians develop new results, they will almost always base their work on existing results. We do this without worry because we know that someone else has proven the existing results correct. For many important results, there are even multiple different proofs, which increases confidence in the result since each individual proof needs to be trusted less.

Unfortunately, many interesting results are so complicated that it is difficult to ensure that a proof of the result is actually correct. It is thus no surprise that proofs are sometimes found to be incorrect even after peer review. The algebraic topologist Vladimir Voevodsky, who is now remembered for initiating the univalent foundations program for computer verification of proofs, began the venture after discovering errors in multiple papers he had written several decades before. In one case he discovered that his main result was plainly incorrect with no chance of mending the proof. Writing about the origins of the univalent foundations program, Voevodsky explains [34]:

“[...] multiple groups of mathematicians studied my paper at seminars and used it in their work and none of them noticed the mistake. And it clearly was not an accident. A technical argument by a trusted author, which is hard to check and looks similar to arguments known to be correct, is hardly ever checked in detail.”

This leads us to one of the main benefits of mechanized proof: a computer can be much more rigorous than even the most pedantic human reviewer while also being much faster, given the necessary amount of processing power.

Another major benefit is that computers can not only check proofs, but also try to find them. This is especially useful to take care of “boring” proof steps such as trivial cases which may otherwise be forgotten and complicated applications of existing lemmas to ensure that all steps in a derivation are supported.

The final major benefit of mechanized proofs are that encoding proofs on a computer allows for integration between programs and proofs. This makes it possible to prove properties about programs such that programs can be formally verified. It is even possible to exploit the deep correspondence between programming and proof theory known as the Curry–Howard isomorphism to create certified programs which include proofs of their own correctness.

### 2.3.2 Proof assistants

A proof assistant is an interactive computer program which implements a proof checker and a system to aid the user while attempting to prove theorems. Many proof assistants also include a programming language which is integrated with the proof system. It is important to note that a proof assistant is not an automated theorem prover; a proof assistant is a tool that assists the user, and creativity is still needed on part of the user to actually prove anything but the most basic results. The role of the proof assistant is to remove the risk of forgetting a step in a proof and to aid the user in applying proof techniques such as induction and case analysis correctly.

Proof assistants work by letting the user define functions, predicates, theorems and so on in a formal language, which the computer can then reason about to verify that their proofs are correct. This reasoning is based on the basic axioms of the proof assistant, but this is no more problematic than the need to trust the basic axioms of any other system, and most proof assistants are based on widely agreed upon and well-understood formal systems. The most popular formal systems to base proof assistants on include the calculus of constructions [35] [36], first-order logic [37], and higher-order logic [33] [38].

One of the main benefits of using a proof assistant is that the user *only* needs to trust the basic axioms of the proof verifier. The proof assistant prevents users from accidentally making any assumptions that cannot be justified from the basic axioms since all proofs must be verifiable in the formal system of the proof assistant.

### 2.3.3 Proofs about programs

The goal of this project is to implement an automated theorem prover and prove that it actually works. To do so we will be exploiting the ability of proof assistants to reason about programs implemented in the language of the proof assistant.

There are two ways in which proof assistants can integrate a programming language. The first is to integrate the programming language into the proof checker itself, exploiting the Curry–Howard isomorphism [39] to use the programming language itself to write proofs. This is typically the approach taken by proof assistants based on the calculus of constructions and related systems. This means that programs are in a sense the proof of their own correctness. Most proof assistants taking this approach distinguish between program code needed for execution of the program and code needed only for proofs. This makes it easy to extract an executable program in a trustworthy manner by simply deleting the code needed only for proofs.

The other approach is to write programs in a “normal” programming language, and then write proofs about them in a formal system which can somehow represent programs in the “normal” programming language. This complicates things a bit, since the user now needs to trust that the representation of the program in the formal proof system is equivalent to the actual program. By choosing a programming language with relatively simple syntax and semantics, the issue of trust can be alleviated since the translation between the program and its representation in proofs can be very simple.

### 2.3.4 Why Isabelle/HOL?

We will use the proof assistant Isabelle for this project. The Isabelle proof assistant can be used with a number of formal systems, but we will use higher-order logic, which is the most popular system, resulting in what is called Isabelle/HOL.

$$\begin{array}{l}
tm ::= \text{Fun } n [tm] \\
\quad | \text{Var } n \\
fm ::= \text{Pre } n [tm] \\
\quad | \text{Imp } fm fm \\
\quad | \text{Dis } fm fm \\
\quad | \text{Con } fm fm \\
\quad | \text{Exi } fm \\
\quad | \text{Uni } fm \\
\quad | \text{Neg } fm
\end{array}$$

Figure 2.1: The syntax of the Sequent Calculus Verifier.

We use Isabelle/HOL for a number of reasons. First, Isabelle’s support for mixing of recursion and corecursion when defining functions is needed for the abstract completeness framework which we use to prove soundness and completeness. Additionally, the Sequent Calculus Verifier is currently implemented in Isabelle/HOL. One major benefit of Isabelle over many other proof assistants is that it supports writing of proofs in a manner that is quite similar to pen-and-paper proofs using the Isar proof language.

Isabelle also allows extraction of programs, including those using coinductive datatypes, to the Haskell programming language. The Haskell programming language not only has good support for coinductive datatypes, but also boasts a large ecosystem of libraries and tools which make implementing a driver program for the automated theorem prover easy.

## 2.4 The Sequent Calculus Verifier

The automated theorem prover developed in this project works in the system of the Sequent Calculus Verifier (SeCaV). This system is a one-sided sequent calculus for first-order logic with terms consisting of functions and variables and formulas consisting of predicates, logical connectives, and the usual quantifiers for first-order logic. Constants can be encoded as functions with arity 0. The syntax of the system is given in Backus Normal Form in fig. 2.1. The system uses de Bruijn indices to identify variables, while functions and predicates are named by natural numbers. The system includes implication, disjunction, conjunction, existential quantification, universal quantification, and negation. Predicates and functions take their arguments as ordered lists of terms, which may be empty to obtain propositional predicates and constant terms, respectively. Sequents are represented as ordered lists of formulas.

The semantics of the a formula are given in terms of an interpretation consisting of a variable environment interpretation  $e$ , a function interpretation  $f$ , and a predicate interpretation  $g$ . The semantics of the system are defined using three recursive functions: one for the semantics of a term, one for the semantics of a list of terms, and one for the semantics of a formula. The semantics of a free variable is looked up in the environment, while the semantics of a function is looked up in the function interpretation with the semantics of the arguments defined using the semantics of the list of arguments. The semantics of an empty list of terms is empty. The semantics of non-empty list of terms is the semantics of the first term followed by the semantics of the rest of the list. The semantics of a predicate

$$\begin{array}{c}
\frac{\text{Neg } p \in z}{\Vdash p, z} \text{ BASIC} \qquad \frac{\Vdash z \quad z \subseteq y}{\Vdash y} \text{ EXT} \qquad \frac{\Vdash p, z}{\Vdash \text{Neg} (\text{Neg } p), z} \text{ NEG} \\
\\
\frac{\Vdash p, q, z}{\Vdash \text{Dis } p \, q, z} \text{ ALPHADIS} \qquad \frac{\Vdash \text{Neg } p, q, z}{\Vdash \text{Imp } p \, q, z} \text{ ALPHAIMP} \qquad \frac{\Vdash \text{Neg } p, \text{Neg } q, z}{\Vdash \text{Neg} (\text{Con } p \, q), z} \text{ ALPHACON} \\
\\
\frac{\Vdash p, z \quad \Vdash q, z}{\Vdash \text{Con } p \, q, z} \text{ BETA CON} \qquad \frac{\Vdash p, z \quad \Vdash \text{Neg } q, z}{\Vdash \text{Neg} (\text{Imp } p \, q), z} \text{ BETA IMP} \\
\\
\frac{\Vdash \text{Neg } p, z \quad \Vdash \text{Neg } q, z}{\Vdash \text{Neg} (\text{Dis } p \, q), z} \text{ BETA DIS} \qquad \frac{\Vdash p [\text{Var } 0/t], z}{\Vdash \text{Exi } p, z} \text{ GAMMA EXI} \\
\\
\frac{\Vdash \text{Neg} (p [\text{Var } 0/t]), z}{\Vdash \text{Neg} (\text{Uni } p), z} \text{ GAMMA UNI} \qquad \frac{\Vdash p [\text{Var } 0/\text{Fun } i []], z \quad i \text{ fresh}}{\Vdash \text{Uni } p, z} \text{ DELTA UNI} \\
\\
\frac{\Vdash \text{Neg} (p [\text{Var } 0/\text{Fun } i []]), z \quad i \text{ fresh}}{\Vdash \text{Neg} (\text{Exi } p), z} \text{ DELTA EXI}
\end{array}$$

Figure 2.2: Proof rules for the Sequent Calculus Verifier.

is looked up in the predicate interpretation with the semantics of the arguments defined using the semantics of the list of arguments. The semantics of the logical connectives are defined using the connectives from the meta-logic in Isabelle/HOL. The semantics of the quantifiers are defined using the quantifiers from the meta-logic, using a “shift” function to handle the shift in de Bruijn-indices for the variable environment.

The system has a number of proof rules, which are displayed in fig. 2.2. The rules should be read from the bottom up, since we generally work backwards from a formula we wish to prove when using the system. The rules are classified according to Smullyan’s uniform notation [40].

The first proof rule is the BASIC rule, which is the only rule that terminates a branch of the proof. The BASIC rule can be applied if the sequent contains both a formula and its negation. Since the sequent is interpreted as a disjunction of the formulas it contains, this rule is essentially using that  $P \vee \neg P$  is always valid to terminate the branch. To simplify the implementation of the system, the positive formula must be the first formula in the sequent to apply the rule.

The EXT rule is used to modify a sequent without modifying any of the formulas in it. It is the only structural rule in the system, and it can be applied to change the position of formulas in a sequent (permutation), to duplicate a formula which already exists in a sequent (contraction), and to remove formulas that are not needed (weakening). It is very often required to change the position of formulas, 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 the rules working on quantifiers do not preserve the original formula when applied. Removing formulas that are not needed is never necessary, but can sometimes shorten proofs significantly and make them easier to understand.

The **NEG** rule is used to remove a double negation from the first formula in a sequent. It can be considered an  $\alpha$ -rule, but we keep it separate from the others because it does not generate two formulas.

The **ALPHADIS**, **ALPHAIMP**, and **ALPHA CON** rules are used to decompose disjunctions, implications, and negated conjunctions, respectively. They can all only be applied to the first formula in a sequent. Since a sequent is a disjunction of formulas, the **ALPHADIS** rule decomposes a disjunction by replacing the formula with two formulas containing the subformulas of the disjunction. The **ALPHAIMP** rule decomposes an implication by replacing it with the negation of the left subformula and the right subformula. Since a sequent is a disjunction of formulas, this corresponds to the usual equivalence  $P \implies Q \equiv \neg P \vee Q$ . The **ALPHA CON** rule decomposes a negated conjunction by replacing it with the negation of both subformulas of the conjunction. This corresponds to the de Morgan equivalence  $\neg(P \wedge Q) \equiv \neg P \vee \neg Q$ .

The **BETA CON**, **BETAIMP**, and **BETA DIS** rules are used to decompose conjunctions, negated implications, and negated disjunctions, respectively. All of these rules require that two sequents are proven separately, and thus create branches in the proof tree. The **BETA CON** rule decomposes a conjunction by creating two branches, replacing the formula with one of the subformulas in one branch and the other subformula in the other branch. This essentially moves the conjunction into the proof tree itself, since both branches now need to be proven separately. The **BETAIMP** rule decomposes a negated implication by creating two branches, replacing the formula with the left formula in one branch and the negation of the right subformula in the other branch. This corresponds to the equivalence

$$\neg(P \implies Q) \equiv P \wedge \neg Q$$

with the conjunction again moved into the proof tree itself. The **BETA DIS** rule decomposes a negated disjunction by creating two branches, replacing the formula with the negation of one subformula in one branch and the negation of the other subformula in the other branch. This corresponds to the equivalence  $\neg(P \vee Q) \equiv \neg P \wedge \neg Q$  with the conjunction again moved into the proof tree itself.

The **GAMMA EXI** rule instantiates an existential quantifier in the first formula in a sequent with a term  $t$  by replacing variable  $0$  by  $t$  and removing the quantifier. The term  $t$  can be any term, since proving the formula with any term will prove that some term for which the formula is valid exists. The **GAMMA UNI** rule instantiates a negated universal quantifier in the first formula in a sequent with a term  $t$  by replacing variable  $0$  by  $t$  and removing the quantifier such that the negation is applied to the quantified subformula. The term  $t$  can again be any term, since proving the negated subformula with any term will prove that the quantified formula is not valid for all terms.

The **DELTA UNI** rule instantiates a universal 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. This is done by replacing variable  $0$  by the fresh constant function and removing the quantifier. Introducing a fresh constant function means that the function cannot have any relationship to any other terms in the sequent, since it does not occur anywhere else in the sequent. This means that the constant function can be replaced by any term without affecting the validity of the formula, which is exactly what is needed to prove a universally quantified formula.

The **DELTA EXI** rule instantiates a negated existential quantifier in the first formula in a sequent with a fresh constant function. This is done by replacing variable  $0$  by the fresh constant function and removing the quantifier such that the negation is applied to the

quantified subformula. Introducing a fresh constant function means that proving the negation of the quantified subformula will prove that the original formula is not valid for any term, since the constant function could be replaced by any term without affecting the validity of the formula.

## 2.5 Abstract frameworks for completeness and soundness

The prover will be implemented using an abstract framework for completeness [4] [5]. The abstract completeness framework represents generic proofs coinductively as potentially infinite trees. It includes facilities for creating an automated theorem prover and proving that provers created in this way are complete. The framework needs a function specifying when rules can be applied and what they do when this happens, i.e. their effect on the proof state, an infinite list of proof rules to try in order, and a set of possible proof states.

The abstract completeness framework consists of a number of definitions and lemmas, which build on top of each other to obtain a final completeness result for coinductive proof trees. The definitions and lemmas are collected in a number of locales which each assume various properties of the effect function, the infinite list of proof rules, and the set of possible proof states. These properties must be proven to specialize the abstract completeness result to the specific proof system.

The first property is *closedness of the set of possible proof states under the effect function*. This ensures that the set of possible proof states actually includes every possible proof state that the prover may encounter while attempting a proof.

The second property is *enabledness*, which means that some proof rule is enabled in every possible proof state. This is used to prove that the prover will never get stuck during a proof.

The third property is *persistency*. This means once a proof rule becomes enabled, it will stay enabled even if another proof rule is applied first. The property is stated as an assumption about the “next” states obtained from applying a proof rule in some proof state. The property is then that any proof rule enabled in a proof state must also be enabled in all next states of that proof state, except those obtained from applying the original proof rule.

When these three properties have been proven, the abstract completeness framework produces a theorem of completeness. This theorem states that for any initial proof state, there is either a finite and well-formed proof tree proving the sequent in the initial proof state, or a saturated proof tree containing an *escape path*. A proof tree is *well-formed* if every child of every node can be generated by applying a proof rule to the node. This ensures that the proof tree can actually be constructed using the proof rules of the system. *Saturation* is a very strong form of fairness which states that if a proof rule is ever enabled, it will eventually be applied. This ensures that the “failed” proof tree has actually tried all possible proof rules when attempting to prove the initial sequent. An *escape path* is a well-formed infinite path within a proof tree. If an escape path exists within a saturated proof tree, the proof tree must contain an unprovable branch, since a saturated infinite well-formed path implies that the prover must have tried to apply every possible proof rule to the branch for an infinitely long time without succeeding. It is important to note that this result does not conflict with the fact that first-order logic is undecidable, since it of course not possible to actually run the prover for an infinitely long time.

A separate abstract soundness framework [5] builds on top of the abstract completeness framework to allow proving soundness of the prover given local soundness of the proof

system. The abstract soundness framework again needs the effect function and the infinite list of proof rules to attempt to apply. It additionally needs a universe of interpretations and a function specifying the semantics of the proof system. The soundness result simply states that the existence of a finite, well-formed proof tree ensures that the sequent in the initial proof state is valid.

## 3 Prover

The main contribution of this project is an automated theorem prover for the SeCaV proof system. The proof search algorithm itself is implemented in Isabelle, while a number of auxiliary functions such as parsing of formulas and extraction of proofs into the SeCaV Unshortener syntax is implemented as a Haskell library. The prover is implemented in the Isabelle modules Prover and Export, while the rest of the modules contain proofs of soundness and completeness.

The Isabelle algorithm is extracted into a Haskell library, which is combined with the parsing and extraction library and a small command-line interface in Haskell to obtain an executable automated theorem prover.

### 3.1 The proof search algorithm

The proof search algorithm is based on algorithm 7.40 from the book *Mathematical Logic for Computer Science* by Mordechai Ben-Ari [19], which is proven to be complete in the book. This algorithm is not designed for the SeCaV system, but for a tableaux system for first-order logic. SeCaV is of course not a tableaux system, so the first step in designing our algorithm will be to modify Ben-Ari's algorithm to fit the syntax and proof rules of SeCaV. Algorithm 1 is a direct translation of Ben-Ari's algorithm to the SeCaV syntax.

The abstract completeness framework used for the prover does not have a notion of open proof branches, so we will remove them from the algorithm. This does not affect the algorithm when attempting to prove valid formulas, since the final clause of the algorithm will then never be encountered. We can thus remove the final clause of the algorithm without affecting the completeness of the algorithm.

Additionally, we need to make the algorithm deterministic to actually implement it. To do this, we will always choose to work on the left-most untermiated branch in the proof tree. Since the sequents of SeCaV are ordered, and proof rules in the system must always be applied to the first formula in the sequent, the selection of formulas can be made deterministic by always choosing the first formula in the sequent. With this change, we will also move away from the set-based representation of sequents in Ben-Ari's algorithm, replacing it by the list-based representation used in the SeCaV system.

We will also simplify the node labels by computing the terms in each sequent on-the-fly as suggested in Ben-Ari's algorithm.

These modifications applied to algorithm 1 result in algorithm 2.

Algorithm 2 does not work, since it always applies rules to the first formula of the sequent, and thus never considers any other formulas. Additionally, algorithm 2 does not specify how the BASIC rule is to be applied, and how the  $\gamma$ -rules are used to generate the specified sequent.

All of these problems must be handled by specifying how the EXT rule should be used to move formulas around and duplicate formulas within the sequent. To ensure structured use of the EXT rule, we will encode its use by introducing two new pseudo-rules, ROTATE



---

---

**Algorithm 1** Ben-Ari's algorithm modified with the syntax of SeCaV.

---

---

*Input:* A SeCaV formula  $p$

*Output:* A proof tree for  $T$  for  $p$ : each branch may be infinite, finite and marked open, or finite and marked closed.

A proof tree is a tree  $T$  where each node is labeled by a pair  $W(n) = (U(n), C(n))$  where:

$$U(n) = \{A_{n_1}, \dots, A_{n_k}\}$$

is a set of formulas and:

$$C(n) = \{c_{n_1}, \dots, c_{n_m}\}$$

is a set of terms.  $C(n)$  contains the list of terms that appear in the formulas in  $U(n)$ . Of course, the sets  $C(n)$  could be created on-the-fly from  $U(n)$ , but the algorithm is easier to understand if they explicitly label the nodes. Initially,  $T$  consists of a single node  $n_0$ , the root, labeled with

$$(\{p\}, \{a_{0_1}, \dots, a_{0_k}\}),$$

where  $\{a_{0_1}, \dots, a_{0_k}\}$  is the set of terms that appear in  $p$ . If  $p$  has no terms, take the first term  $a_0$  in the set of terms and label the node with  $(\{p\}, \{a_0\})$ .

The proof tree is built inductively by repeatedly *choosing* an unmarked leaf  $l$  labeled with  $W(l) = (U(l), C(l))$ , and applying the *first applicable rule* in the following list:

- If  $U(l)$  contains a complementary pair of literals, use the BASIC rule to mark the leaf closed.
- If  $U(l)$  is not a set of literals, *choose* a formula  $A$  in  $U(l)$  that is an  $\alpha$ -,  $\beta$ - or  $\delta$ -formula.
  - If  $A$  is an  $\alpha$ -formula, create a new node  $l'$  as a child of  $l$ . Label  $l'$  with:

$$W(l') = ((U(l) - \{A\}) \cup \{\alpha_1, \alpha_2\}, C(l)).$$

(In the case that  $A$  is  $\neg\neg A_1$ , there is no  $\alpha_2$ .)

- If  $A$  is a  $\beta$ -formula, create two new nodes  $l'$  and  $l''$  as children of  $l$ . Label  $l'$  and  $l''$  with:

$$W(l') = ((U(l) \setminus \{A\}) \cup \{\beta_1\}, C(l)),$$

$$W(l'') = ((U(l) \setminus \{A\}) \cup \{\beta_2\}, C(l)).$$

- If  $A$  is a  $\delta$ -formula, create a new node  $l'$  as a child of  $l$  and label  $l'$  with:

$$W(l') = ((U(l) \setminus \{A\}) \cup \{\delta(a')\}, C(l) \cup \{a'\}),$$

where  $a'$  is some constant that *does not appear in*  $U(l)$ .

- Let  $\{\gamma_{l_1}, \dots, \gamma_{l_m}\} \subseteq U(l)$  be all the  $\gamma$ -formulas in  $U(l)$  and let  $C(l) = \{c_{l_1}, \dots, c_{l_k}\}$ . Create a new node  $l'$  as a child of  $l$  and label  $l'$  with

$$W(l') = \left( U(l) \cup \left\{ \bigcup_{i=1}^m \bigcup_{j=1}^k \gamma_{l_i}(c_{l_j}) \right\}, C(l) \right).$$

*However*, if  $U(l)$  consists only of literals and  $\gamma$ -formulas and if  $U(l')$  as constructed would be the same as  $U(l)$ , do not create node  $l'$ ; instead, mark the leaf  $l$  as *open*.

---

---

**Algorithm 2** Algorithm 1 modified to be deterministic and without open branches and term labels.

---

*Input:* A SeCaV formula  $p$

*Output:* A proof tree for  $T$  for  $p$ : each branch may either be infinite, or finite and terminated by an application of the BASIC rule.

A proof tree is a tree  $T$  where each node is labeled by a list of formulas  $U(n)$  where:

$$U(n) = A_{n_1}, \dots, A_{n_k}.$$

Initially,  $T$  consists of a single node  $n_0$ , the root, labeled with  $U(n_0) = [p]$ .

The proof tree is built inductively by repeatedly applying the *first applicable rule* in the following list to left-most unterminated branch  $l$  of  $T$ :

- If  $U(l)$  contains a complementary pair of literals, use the BASIC rule to terminate the branch.
- If  $U(l)$  is not a set of literals, consider the first formula  $A$  in  $U(l)$  that is an  $\alpha$ -,  $\beta$ - or  $\delta$ -formula.
  - If  $A$  is an  $\alpha$ -formula, create a new node  $l'$  as a child of  $l$ . Label  $l'$  with:

$$U(l') = \alpha_1, \alpha_2, (U(l) - \{A\}).$$

(In the case that  $A$  is  $\neg\neg A_1$ , there is no  $\alpha_2$ .)

- If  $A$  is a  $\beta$ -formula, create two new nodes  $l'$  and  $l''$  as children of  $l$ . Label  $l'$  and  $l''$  with:

$$U(l') = \beta_1, (U(l) \setminus \{A\}),$$

$$U(l'') = \beta_2, (U(l) \setminus \{A\}).$$

- If  $A$  is a  $\delta$ -formula, create a new node  $l'$  as a child of  $l$  and label  $l'$  with:

$$U(l') = \delta(a'), (U(l) \setminus \{A\}),$$

where  $a'$  is some constant that *does not appear in*  $U(l)$ .

- Let  $\{\gamma_{l_1}, \dots, \gamma_{l_m}\} \subseteq U(l)$  be all the  $\gamma$ -formulas in  $U(l)$  and let  $C(l) = \{c_{l_1}, \dots, c_{l_k}\}$  be the set of terms in  $U(l)$ . Create a new node  $l'$  as a child of  $l$  and label  $l'$  with

$$U(l') = U(l), \bigcup_{i=1}^m \bigcup_{j=1}^k \gamma_{l_i}(c_{l_j}).$$


---

and `DUPLICATE`, which replace the `EXT` rule:

$$\frac{\Vdash p, z}{\Vdash z, p} \text{ ROTATE} \qquad \frac{\Vdash p, z}{\Vdash p, z, p} \text{ DUPLICATE}$$

The `ROTATE` rule moves the head of the sequent to the end, while the `DUPLICATE` rule creates a copy of the first formula in the sequent at the end of the sequent.

Using these rules, we obtain algorithm 3.

In algorithm 3, rules are applied until certain conditions hold. If the sequent contains a complementary pair of literals, for example, `ROTATE` rules are applied until the positive literal is the first formula in the sequent, at which point a `BASIC` rule is applied. In addition to this type of application, algorithm 3 still contains the concept of applying the “first applicable rule” to a branch.

Unfortunately, the abstract completeness framework used for the prover requires that the stream of rules to attempt to apply must be a constant. For this reason, the prover is not able to directly examine the sequent to determine which rule it should apply next.

To solve this problem, we introduce the concept of *proof phases* and a new pseudo-rule, `NEXT`, which transitions the prover to the next proof phase. The phases form a cycle which the prover will move through as it attempts to apply proof rules. To encode the current phase, we modify the proof state so it contains not only the current sequent, but also the current phase, including any additional information needed for the phase. Each phase will correspond to one of the overall rules in algorithm 3.

The first phase is the “Basic” phase, in which a `BASIC` rule may be applied if the branch can be terminated by doing so. The next phase is the “Alpha–Beta–Delta” phase, in which  $\alpha$ -,  $\beta$ -, and  $\delta$ -rules may be applied if the sequent contains any  $\alpha$ -,  $\beta$ -, or  $\delta$ -formulas. The final phase is the “Gamma” phase, in which all  $\gamma$ -formulas are instantiated with all terms in the sequent using  $\gamma$ -rules. All of the phases also allow structured use of the `ROTATE` rule, and the Gamma phase uses the `DUPLICATE` rule.

Whenever a phase ends, the `NEXT` rule is applied to move on to the next phase in a cyclic fashion. The `BASIC` phase ends immediately if the branch cannot be terminated, and the prover then moves to an Alpha–Beta–Delta phase. The Alpha–Beta–Delta phase ends when there are no more  $\alpha$ -,  $\beta$ -, or  $\delta$ -formulas in the sequent, and the prover then moves to a Gamma phase. Whenever an  $\alpha$ -,  $\beta$ -, or  $\delta$ -rule is applied, the prover moves to a `BASIC` phase to determine whether the branch can be terminated. The Gamma phase ends when all  $\gamma$ -formulas have been instantiated, and the prover then moves to a `BASIC` phase.

The `BASIC` phase does not require any additional information, since the phase either applies `ROTATE` rules until a `BASIC` rule can be applied, at which point the branch is terminated, or no rules at all. The Alpha–Beta–Delta phase also requires no additional information, since it simply applies  $\alpha$ -,  $\beta$ -, and  $\delta$ -rules until no more can be applied.

The Gamma phase is subdivided into two subphases: a Preparation phase and an Instantiation phase. The Gamma phase cycles between the two subphases until all  $\gamma$ -formulas in the original sequent have been instantiated with all terms. The Preparation subphase applies `ROTATE` rules until a  $\gamma$ -formula is at the head of the sequent. It then applies a `DUPLICATE` rule to create one copy of the  $\gamma$ -formula for every term in the sequent, and simultaneously transitions to an Instantiation subphase. The Instantiation subphase applies an appropriate  $\gamma$ -rule, then applies a `ROTATE` rule to move the newly instantiated formula to the back of the sequent. This is done for every term in the sequent, and the prover then

---



---

**Algorithm 3** Algorithm 2 modified to use the pseudo-rules ROTATE and DUPLICATE.

---



---

*Input:* A SeCaV formula  $p$

*Output:* A proof tree for  $T$  for  $p$ : each branch may either be infinite, or finite and terminated by an application of the BASIC rule.

A proof tree is a tree  $T$  where each node is labeled by a list of formulas  $U(n)$  where:

$$U(n) = A_{n_1}, \dots, A_{n_k}.$$

Initially,  $T$  consists of a single node  $n_0$ , the root, labeled with  $U(n_0) = [p]$ .

The proof tree is built inductively by repeatedly applying the *first applicable rule* in the following list to left-most unterminated branch  $l$  of  $T$ :

- If  $U(l)$  contains a complementary pair of literals, apply ROTATE rules until the positive literal of the pair is the first formula in the sequent, then apply the BASIC rule to terminate the branch.
- If  $U(l)$  is not a set of literals and  $U(l)$  contains any  $\alpha$ -,  $\beta$ - or  $\delta$ -formulas, apply ROTATE rules until the first formula in  $U(l)$  is an  $\alpha$ -,  $\beta$ - or  $\delta$ -formula  $A$ .
  - If  $A$  is an  $\alpha$ -formula, create a new node  $l'$  as a child of  $l$ . Label  $l'$  with:

$$U(l') = \alpha_1, \alpha_2, (U(l) - \{A\}).$$

(In the case that  $A$  is  $\neg\neg A_1$ , there is no  $\alpha_2$ .)

- If  $A$  is a  $\beta$ -formula, create two new nodes  $l'$  and  $l''$  as children of  $l$ . Label  $l'$  and  $l''$  with:

$$\begin{aligned} U(l') &= \beta_1, (U(l) \setminus \{A\}), \\ U(l'') &= \beta_2, (U(l) \setminus \{A\}). \end{aligned}$$

- If  $A$  is a  $\delta$ -formula, create a new node  $l'$  as a child of  $l$  and label  $l'$  with:

$$U(l') = \delta(a'), (U(l) \setminus \{A\}),$$

where  $a'$  is some constant that *does not appear in*  $U(l)$ .

- Let  $C(l) = \{c_{l_1}, \dots, c_{l_k}\}$  be the set of terms in  $U(l)$ .  
For every formula in  $G(l)$ :
  - If the first formula in  $U(l)$  is not a  $\gamma$ -formula, apply the ROTATE rule
  - If the first formula in  $U(l)$  is a  $\gamma$ -formula, do the following for each term in  $C(l)$ :
    - \* Apply the DUPLICATE rule
    - \* Apply the appropriate  $\gamma$ -rule (either GAMMAEXI or GAMMAUNI, depending on the formula)
    - \* Apply the ROTATE rule

This will result in the creation of a sequence of new nodes starting with a child of  $l$ , with the final node  $l'$  labeled with

$$U(l') = U(l)_1, \gamma_{l_1}(c_{l_1}), \dots, \gamma_{l_1}(c_{l_k}), \dots, U(l)_n, \gamma_{l_n}(c_{l_1}), \dots, \gamma_{l_n}(c_{l_k})$$


---

transitions to another Preparation subphase. When all formulas in the original sequent have been instantiated or moved to the back of the sequent, the prover transitions from the Preparation subphase to a Basic phase.

The Preparation subphase needs to know how many formulas were in the sequent before the Gamma phase started so it knows when to transition to a BASIC phase. For efficiency, it also stores the list of terms to instantiate formulas with so they do not need to be recalculated for every Instantiation subphase. The Instantiation subphase needs to know how many formulas are still left to instantiate and which terms to instantiate formulas with. The subphase actually stores two lists of terms: one containing all terms to instantiate with, and one containing only the terms that have not yet been instantiated. Additionally, it must keep track of whether the current head of the sequent should be instantiated or moved to the back of the sequent. This is done by storing a Boolean flag which is set to true when the current head of the sequent should be moved to the back of the sequent and false when it should be instantiated. It is necessary to keep track of this because a formula may contain two quantifications right after one another, and the prover can thus not know whether a formula has just been instantiated solely by inspecting the sequent. The prover transitions from an Instantiation subphase to a Preparation subphase whenever the flag is false (indicating that the last instantiated formula has been rotated away) and the list of terms to instantiate is empty.

---

**Algorithm 4** Algorithm 3 modified to use proof phases.

---

*Input:* A SeCaV formula  $p$

*Output:* A proof tree for  $T$  for  $p$ : each branch may either be infinite, or finite and terminated by an application of the BASIC rule.

A proof tree is a tree  $T$  where each node is labeled by a state consisting of a list of formulas  $U(n)$  where:

$$U(n) = A_{n_1}, \dots, A_{n_k}$$

and a phase  $P(n)$ .

Initially,  $T$  consists of a single node  $n_0$ , the root, labeled with  $U(n_0) = [p]$  and  $P(n_0) = \text{BASIC}$ .

The proof tree is built inductively by repeatedly applying proof rules to create new nodes in the tree. Proof rules are applied based on the current phase of the proof. When  $P(l) = \text{BASIC}$ , proof rules are applied according to algorithm 4a. When  $P(l) = \text{Alpha-Beta-Delta}$ , proof rules are applied according to algorithm 4b. When  $P(l) = \text{Preparation}$ , proof rules are applied according to algorithm 4c. When  $P(l) = \text{Instantiation}$ , proof rules are applied according to algorithm 4d.

---



---

**Algorithm 4a** The BASIC phase.

---

- If  $U(l)$  contains a complementary pair of literals and the positive literal is at the head of the sequent, apply a BASIC rule to terminate the branch.
  - If  $U(l)$  contains a complementary pair of literals and the positive literal is not at the head of the sequent, apply a ROTATE rule to move the head of the sequent to the back of the sequent and stay in the BASIC phase.
  - Otherwise, apply a NEXT rule to transition to an Alpha-Beta-Delta phase.
- 

Rewriting algorithm 3 to use the concept of phases, we obtain algorithm 4. Algorithm 4 controls the application of rules such that exactly one rule is enabled at all times, and the algorithm is thus deterministic. When only one rule is enabled at any time, the rule

---

---

**Algorithm 4b** The Alpha–Beta–Delta phase.

---

---

- If the head  $A$  of the sequent is an  $\alpha$ -formula, create a new node  $l'$  as a child of  $l$ . Label  $l'$  with:

$$U(l') = \alpha_1, \alpha_2, (U(l) - \{A\}),$$
$$P(l') = \text{Basic.}$$

(In the case that  $A$  is  $\neg\neg A_1$ , there is no  $\alpha_2$ .)

- If the head  $A$  of the sequent is a  $\beta$ -formula, create two new nodes  $l'$  and  $l''$  as children of  $l$ . Label  $l'$  and  $l''$  with:

$$U(l') = \beta_1, (U(l) \setminus \{A\}),$$
$$P(l') = \text{Basic,}$$
$$U(l'') = \beta_2, (U(l) \setminus \{A\}),$$
$$P(l'') = \text{Basic.}$$

- If the head  $A$  of the sequent is a  $\delta$ -formula, create a new node  $l'$  as a child of  $l$  and label  $l'$  with:

$$U(l') = \delta(a'), (U(l) \setminus \{A\}),$$
$$P(l') = \text{Basic.}$$

where  $a'$  is some constant that *does not appear in*  $U(l)$ .

- If the head of the sequent is not an  $\alpha$ -,  $\beta$ -, or  $\delta$ -formula, but there are  $\alpha$ -,  $\beta$ - or  $\delta$ -formulas in the sequent, apply a ROTATE rule to move the head of the sequent to the back of the sequent and stay in the Alpha–Beta–Delta phase.
  - If there are no  $\alpha$ -,  $\beta$ -, or  $\delta$ -formulas in the sequent, apply a NEXT rule to transition to a Preparation subphase of the Gamma phase, initializing the phase with the number of formulas in the sequent and a list of all terms in the sequent.
- 

---

---

**Algorithm 4c** The Gamma Preparation subphase.

---

---

Let  $n(l)$  be the number of formulas in the sequent that have not yet been instantiated. This number is set when entering the phase such that it only counts formulas that were present in the sequent before this Gamma phase began. Let  $t(l)$  be the list of terms present in the sequent. This list is not needed in this subphase, but is precomputed for use in subsequent Gamma Instantiation subphases to avoid unnecessary recomputation of the list.

- If the head of the sequent is a  $\gamma$ -formula, and  $n(l) > 0$ , apply a DUPLICATE rule to create  $\text{length}(t(l))$  copies of the head of the sequent at the beginning of the sequent, and move the head of the sequent to the back of the sequent. Simultaneously, transition to a Gamma Instantiation subphase with the number of formulas left set to  $n(l)$ , both lists of terms set to  $t(l)$  and the Boolean flag set to false.
  - If the head of the sequent is not a  $\gamma$ -formula, and  $n(l) > 0$ , apply a ROTATE rule to move the head of the sequent to the back of the sequent.
  - If  $n(l) = 0$ , apply a NEXT rule to transition to a BASIC phase.
-

---

---

**Algorithm 4d** The Gamma Instantiation subphase.

---

---

Let  $n_i(l)$  be the number of formulas in the sequent that have not yet been instantiated. This number is not needed in this phase, but must be preserved for use in subsequent Gamma Preparation subphases. Let  $t_0(l)$  be the list of terms in the sequent. This list is not needed in this subphase, but is precomputed for use in subsequent Gamma Instantiation subphases to avoid unnecessary recomputation of the list, and must therefore be preserved. Let  $t_i(l)$  be the list of terms in the sequent that the current formula has not yet been instantiated with. Let  $b$  be a Boolean flag that determines whether the next proof step should be an instantiation or a rotation.

- If  $b = \text{true}$ , apply a `ROTATE` rule to move the head of the sequent to the back of the sequent, preserving the values of  $n_i(l)$ ,  $t_0(l)$  and  $t_i(l)$ , and setting  $b = \text{false}$ .
  - If  $b = \text{false}$ , and  $t_i(l)$  is not empty, apply an appropriate  $\gamma$ -rule to instantiate the  $\gamma$ -formula at the head of the sequent with the term at the head of  $t_i(l)$  (the Gamma Preparation phase ensures that the head of the sequent will always be a  $\gamma$ -formula when we reach this case). Preserve the values of  $n_i(l)$  and  $t_0(l)$ , remove the head of  $t_i(l)$  and set  $b = \text{true}$ .
  - If  $b = \text{false}$ , and  $t_i(l)$  is empty, apply a `NEXT` rule to transition to a Gamma Preparation phase, setting  $n(l) = n_i(l) - 1$  and  $t(l) = t_0(l)$ .
- 

stream required by the abstract completeness framework can be defined as any stream in which every proof (pseudo-)rule occurs infinitely often, as the prover will simply move past disabled rules until it finds the one rule that is enabled. For simplicity, we take the rule stream to be a cycle of all of the pseudo-rules in order.

## 3.2 Implementing the algorithm

The abstract completeness framework expects the rules to be specified in terms of their effect when applied to a node. This effect function and a number of auxiliary functions needed to specify it are contained in the module `Prover` in the Isabelle session. The `Prover` module also contains an infinite list of rules to attempt to apply, proofs that the effect function and the infinite list of rules satisfy the conditions required to use the abstract completeness framework, and a definition of a non-executable prover using the framework. We will see how this definition is made executable later on.

### 3.2.1 Datatypes

To implement this effect function in Isabelle, we will first need to define the datatypes needed for the algorithm. First, a sequent is a list of SeCaV formulas:

**type-synonym** `sequent` =  $\langle \text{fm list} \rangle$

We also need a datatype for the phases used by the prover:

**datatype** `phase` = `PBasic` | `PABD` | `PPrepGamma nat (tm list)` | `PInstGamma nat (tm list) (tm list) bool`

As previously mentioned, the Gamma subphases both need to store the number of formulas in the original sequent and the list of terms in the sequent, while the Gamma Instantiation subphase also needs to store a list of terms not yet instantiated and a flag for whether the next formula should be instantiated or rotated.

Next, we need a datatype for the proof state, which consists of a sequent and a phase:

**type-synonym** `state` =  $\langle \text{sequent} \times \text{phase} \rangle$

We also need a datatype for the pseudo-proof rules used by the prover:

```
datatype PseudoRule =
  Basic
  | AlphaDis | AlphaImp | AlphaCon
  | BetaCon | BetaImp | BetaDis
  | DeltaUni | DeltaExi
  | NegNeg
  | GammaExi | GammaUni
  | Rotate
  | Duplicate
  | Next
```

We note that the Gamma-pseudo-rules, in contrast to the “real” proof rules from the Sequent Calculus Verifier, do not specify which terms they are instantiated with, and that the EXT rule has been replaced by the ROTATE and DUPLICATE pseudo-rules.

### 3.2.2 Auxiliary functions

To make the definition of the effect function for the pseudo-proof rules shorter and more readable, we will define a number of auxiliary functions before proceeding to the actual effect function.

When applying a Delta-rule, we need to instantiate a quantifier with a fresh function, and we thus need a way to generate a fresh function name (which is a natural number index in SeCaV). This is accomplished with the help of two functions which compute the largest index used for functions in a term and a formula, respectively:

```
fun maxFunTm :: (tm ⇒ nat) where
  ⟨maxFunTm (Fun n ts) = max n (foldl max 0 (map maxFunTm ts))⟩
  | ⟨maxFunTm (Var n) = 0⟩
```

```
fun maxFun :: (fm ⇒ nat) where
  ⟨maxFun (Pre n ts) = foldl max 0 (map maxFunTm ts)⟩
  | ⟨maxFun (Imp f1 f2) = max (maxFun f1) (maxFun f2)⟩
  | ⟨maxFun (Dis f1 f2) = max (maxFun f1) (maxFun f2)⟩
  | ⟨maxFun (Con f1 f2) = max (maxFun f1) (maxFun f2)⟩
  | ⟨maxFun (Exi f) = maxFun f⟩
  | ⟨maxFun (Uni f) = maxFun f⟩
  | ⟨maxFun (Neg f) = maxFun f⟩
```

Using these functions, we can define a function, which generates a new function index:

```
fun generateNew :: (fm ⇒ fm list ⇒ nat) where
  ⟨generateNew p z = 1 + max (maxFun p) (foldl max 0 (map maxFun z))⟩
```

When applying a Gamma-rule, we need to instantiate a quantifier with all terms occurring in the sequent (except bound variables). To do so, we first write a primitive recursive function to flatten a list of lists into a single list:

```
primrec flatten :: ('a list list ⇒ 'a list) where
  ⟨flatten [] = []⟩
  | ⟨flatten (l # ls) = l @ flatten ls⟩
```

Using this function, we can define two functions which compute a list of all subterms in occurring in a term and a list of all subterms occurring in a formula, respectively:

```
fun subtermTm :: (nat ⇒ tm ⇒ tm list) where
  ⟨subtermTm q (Fun n ts) = (Fun n ts) # (remdups (flatten (map (subtermTm q) ts)))⟩
  | ⟨subtermTm q (Var n) = (if n ≥ q then [Var n] else [])⟩
```



```

fun subtermFm :: ⟨nat ⇒ fm ⇒ tm list⟩ where
  ⟨subtermFm q (Pre - ts) = remdups (flatten (map (subtermTm q) ts))⟩
| ⟨subtermFm q (Imp f1 f2) = remdups (subtermFm q f1 @ subtermFm q f2)⟩
| ⟨subtermFm q (Dis f1 f2) = remdups (subtermFm q f1 @ subtermFm q f2)⟩
| ⟨subtermFm q (Con f1 f2) = remdups (subtermFm q f1 @ subtermFm q f2)⟩
| ⟨subtermFm q (Exi f) = subtermFm (q + 1) f⟩
| ⟨subtermFm q (Uni f) = subtermFm (q + 1) f⟩
| ⟨subtermFm q (Neg f) = subtermFm q f⟩

```

Finally, we define a function which either returns a list of all subterms occurring in a sequent, or, if this list is empty, the first function as an arbitrary default value:

```

fun subterms :: ⟨sequent ⇒ tm list⟩ where
  ⟨subterms s = (case remdups (flatten (map (subtermFm 0) s)) of
    [] ⇒ [Fun 0 []]
  | ts ⇒ ts)⟩

```

It is necessary to have a default value to ensure that even quantifiers in formulas like  $\forall x.P(x) \implies \exists x.P(x)$ , which contain no terms except bound variables, can be instantiated.

When the prover is in the Alpha–Beta–Delta phase, it needs to know when to stop looking for new formulas to decompose, this being when there are no more formulas which can be decomposed using Alpha-, Beta-, or Delta-rules. This can be detected by inspecting each formula in the sequent, to see whether it contains a pattern that matches one of the rules with the following function:

```

fun abdDone :: ⟨sequent ⇒ bool⟩ where
  ⟨abdDone (Dis - - # -) = False⟩
| ⟨abdDone (Imp - - # -) = False⟩
| ⟨abdDone (Neg (Con - -) # -) = False⟩
| ⟨abdDone (Con - - # -) = False⟩
| ⟨abdDone (Neg (Imp - -) # -) = False⟩
| ⟨abdDone (Neg (Dis - -) # -) = False⟩
| ⟨abdDone (Neg (Neg -) # -) = False⟩
| ⟨abdDone (Uni - # -) = False⟩
| ⟨abdDone (Neg (Exi -) # -) = False⟩
| ⟨abdDone (- # z) = abdDone z⟩
| ⟨abdDone [] = True⟩

```

When the prover is in the Basic phase, it needs to know whether it is currently possible to close the branch by applying ROTATE and BASIC rules, or whether it should just go on to the next phase. This can be detected by inspecting each formula in the sequent to see whether the sequent contains its negation with the following function:

```

fun branchDone :: ⟨sequent ⇒ bool⟩ where
  ⟨branchDone [] = False⟩
| ⟨branchDone (Neg p # z) = (p ∈ set z ∨ Neg (Neg p) ∈ set z ∨ branchDone z)⟩
| ⟨branchDone (p # z) = (Neg p ∈ set z ∨ branchDone z)⟩

```

In the second case of the function, the check for  $\text{Neg } (\text{Neg } p) \in z$  is only necessary to prove the following lemma, which will be needed later:

```

lemma pinz-done: ⟨Neg p ∈ set z ⟹ branchDone (p # z)⟩
  by (cases p; simp)

```

### 3.2.3 The effect function

The effect of each pseudo-rule is specified by an effect function, which takes a pseudo-rule and a proof state and returns either a finite set of new proof states (branches) created by applying the pseudo-rule, or nothing, if the pseudo-rule cannot be applied to the current proof state.

The effect function works by pattern matching on the pseudo-rule, sequent, and prover phase to compute whether the pseudo-rule can be applied in the current state, and if so, which results it should have. The effect function first pattern matches on the pseudo-rule it should attempt to apply, and then on the proof state. The pattern matching in the function is split into two stages to aid performance of the termination checker of Isabelle, which needs to examine every case to prove that the function is total.

The first case is the `BASIC` rule, which is applicable only in the `BASIC` phase, and only if the sequent contains the negation of formula at the head of the sequent, in which case the rule will close the branch:

```
⟨effect Basic state = (case state of
  ((p # z), PBasic) ⇒ (if Neg p ∈ set z then Some {} else None)
  | (-, -) ⇒ None)⟩
```

The next case is the `ROTATE` pseudo-rule, which is applicable in all proof states:

```
| ⟨effect Rotate state = (case state of
  (p # z, PBasic) ⇒ (if branchDone (p # z) ∧ Neg p ∉ set z
    then Some { (z @ [p], PBasic) } else None)
  | (p # z, PABD) ⇒ (if abdDone (p # z) then None else
    (if abdDone [p]
      then Some { (z @ [p], PABD) } else None))
  | ((Exi p) # -, PPrepGamma --) ⇒ None
  | ((Neg (Uni p)) # -, PPrepGamma --) ⇒ None
  | (p # z, PPrepGamma n ts) ⇒
    (if n = 0 then None else Some { (z @ [p], PPrepGamma (n - 1) ts) })
  | (p # z, PInstGamma n ots ts True) ⇒
    Some { (z @ [p], PInstGamma n ots ts False) }
  | (-, PInstGamma --- False) ⇒ None
  | ([], -) ⇒ None)⟩
```

If the sequent is empty, the rule can never be applied. In the `Basic` phase, the rule is applicable if the branch contains a formula and its negation and if this formula is not the first formula in the sequent, in which case the rule will move the first formula in the sequent to the end of the sequent. In the `Alpha-Beta-Delta` phase, the rule is applicable if the branch contains a formula that can be decomposed by an `Alpha-`, `Beta-`, or `Delta-` rule, and this formula is not the first formula in the sequent. If this is the case, the rule will move the first formula in the sequent to the end of the sequent. In the `Gamma Preparation` phase, the rule is only applicable if the first formula in the sequent is not existentially quantified or the negation of a universally quantified formula. Additionally, the number of formulas remaining must be greater than zero, in which case the rule will move the first formula in the sequent to the end of the sequent and subtract one from the number of remaining formulas. In the `Gamma Instantiation` phase, the rule is applicable if the flag is set to `True`, in which case the rule will move the first formula in the sequent to the end of the sequent and set the flag to `false`.

The next case is the NEXT pseudo-rule, which is also applicable in all proof states:

```

| effect Next state = (case state of
  (s, PBasic) ⇒ (if branchDone s then None else Some {| (s, PABD) |})
  | (s, PABD) ⇒ (if abdDone s
    then Some {| (s, PPrepGamma (length s) (subterms s)) |}
    else None)
  | ([], PPrepGamma n -) ⇒ Some {| ([], PBasic) |}
  | (s, PPrepGamma n -) ⇒ (if n = 0 then Some {| (s, PBasic) |} else None)
  | (s, PInstGamma n ots [] False) ⇒ Some {| (s, PPrepGamma (n - 1) ots) |}
  | (Pre - - # z, PInstGamma n ots (- # -) False) ⇒
    Some {| ([], PPrepGamma (n - 1) ots) |}
  | (Imp - - # z, PInstGamma n ots (- # -) False) ⇒
    Some {| ([], PPrepGamma (n - 1) ots) |}
  | (Dis - - # z, PInstGamma n ots (- # -) False) ⇒
    Some {| ([], PPrepGamma (n - 1) ots) |}
  | (Con - - # z, PInstGamma n ots (- # -) False) ⇒
    Some {| ([], PPrepGamma (n - 1) ots) |}
  | (Uni - # z, PInstGamma n ots (- # -) False) ⇒
    Some {| ([], PPrepGamma (n - 1) ots) |}
  | (Neg (Pre - -) # z, PInstGamma n ots (- # -) False) ⇒
    Some {| ([], PPrepGamma (n - 1) ots) |}
  | (Neg (Imp - -) # z, PInstGamma n ots (- # -) False) ⇒
    Some {| ([], PPrepGamma (n - 1) ots) |}
  | (Neg (Dis - -) # z, PInstGamma n ots (- # -) False) ⇒
    Some {| ([], PPrepGamma (n - 1) ots) |}
  | (Neg (Con - -) # z, PInstGamma n ots (- # -) False) ⇒
    Some {| ([], PPrepGamma (n - 1) ots) |}
  | (Neg (Exi -) # z, PInstGamma n ots (- # -) False) ⇒
    Some {| ([], PPrepGamma (n - 1) ots) |}
  | (Neg (Neg -) # z, PInstGamma n ots (- # -) False) ⇒
    Some {| ([], PPrepGamma (n - 1) ots) |}
  | ([], PInstGamma n ots -) ⇒ Some {| ([], PPrepGamma (n - 1) ots) |}
  | (-, PInstGamma - - -) ⇒ None)

```

In the BASIC phase, the rule is applicable if the branch cannot be closed by applying the BASIC rule, in which case the rule makes the prover transition to an Alpha–Beta–Delta phase. In the Alpha–Beta–Delta phase, the rule is applicable if there are no more formulas in the sequent which can be decomposed by Alpha-, Beta, or Delta-rules. If this is the case, the rule makes the prover transition to a Gamma Preparation subphase, setting the number of remaining formulas to the length of the sequent and computing the list of subterms to instantiate with. In the Gamma Preparation subphase, the rule is applicable if the sequent is empty or if the number of remaining formulas to instantiate is zero. In both cases, the rule will make the prover transition to a BASIC phase. In the Gamma Instantiation subphase, the rule is applicable if the sequent is empty, or there are no more terms to instantiate with and the flag is set to False. The rule is also applicable if the first formula in the sequent is not a Gamma-formula and the flag is set to false, but this is only to simplify the proofs, since this will never actually happen, which is why the sequent is just set to the empty sequent in this case. In all of these cases, the rule makes the prover transition to a Gamma Preparation subphase, subtracting one from the number of remaining formulas.

The next cases are the Alpha-, Beta-, and Delta-rules, which are only applicable in the Alpha–Beta–Delta phase:

```

| ⟨effect AlphaDis state = (case state of
  ((Dis p q # z), PABD) ⇒ Some {| (p # q # z, PBasic) |}
  | (-, -) ⇒ None)⟩
| ⟨effect AlphaImp state = (case state of
  ((Imp p q # z), PABD) ⇒ Some {| (Neg p # q # z, PBasic) |}
  | (-, -) ⇒ None)⟩
| ⟨effect AlphaCon state = (case state of
  ((Neg (Con p q) # z), PABD) ⇒ Some {| (Neg p # Neg q # z, PBasic) |}
  | (-, -) ⇒ None)⟩
| ⟨effect BetaCon state = (case state of
  ((Con p q # z), PABD) ⇒ Some {| (p # z, PBasic) , (q # z, PBasic) |}
  | (-, -) ⇒ None)⟩
| ⟨effect BetaImp state = (case state of
  ((Neg (Imp p q) # z), PABD) ⇒
    Some {| (p # z, PBasic) , (Neg q # z, PBasic) |}
  | (-, -) ⇒ None)⟩
| ⟨effect BetaDis state = (case state of
  ((Neg (Dis p q) # z), PABD) ⇒
    Some {| (Neg p # z, PBasic), (Neg q # z, PBasic) |}
  | (-, -) ⇒ None)⟩
| ⟨effect DeltaUni state = (case state of
  ((Uni p # z), PABD) ⇒
    Some {| (sub 0 (Fun (generateNew p z) []) p # z, PBasic) |}
  | (-, -) ⇒ None)⟩
| ⟨effect DeltaExi state = (case state of
  ((Neg (Exi p) # z), PABD) ⇒
    Some {| (Neg (sub 0 (Fun (generateNew p z) []) p) # z, PBasic) |}
  | (-, -) ⇒ None)⟩
| ⟨effect NegNeg state = (case state of
  ((Neg (Neg p) # z), PABD) ⇒ Some {| (p # z, PBasic) |}
  | (-, -) ⇒ None)⟩

```

Each rule is applicable if the first formula in the sequent matches the pattern of the proof rule. Alpha-rules and the `NEGNEG`-rule decompose the first formula into two formulas placed at the head of the current branch. Beta-rules decompose the first formula, creating two branches with one formula added to the head of each branch. Delta-rules instantiate the quantifier with a fresh function.

The next case is the `DUPLICATE` pseudo-rule, which is only applicable in the Gamma Preparation subphase:

```

| ⟨effect Duplicate state = (case state of
  (Exi p # z, PPrepGamma n ts) ⇒ (if n = 0 then None else
    Some {| (replicate (length ts) (Exi p) @ z @ [Exi p],
      PInstGamma n ts ts False) |})
  | ((Neg (Uni p)) # z, PPrepGamma n ts) ⇒ (if n = 0 then None else
    Some {| (replicate (length ts) (Neg (Uni p)) @ z @ [Neg (Uni p)],
      PInstGamma n ts ts False) |})
  | - ⇒ None)⟩

```

The rule is applicable if the first formula in the sequent is a Gamma-formula and the number of remaining formulas is greater than zero, in which case the rule will replicate the first formula once for each term to instantiate with at the beginning of the sequent, and the first formula once at the end of the sequent and transition to a Gamma Instantiation

subphase with the flag set to False.

The final cases are the Gamma-rules, which are only applicable in the Gamma Instantiation subphase:

$$\begin{aligned} & | \langle \text{effect GammaExi state} = (\text{case state of} \\ & \quad (\text{Exi } p \# z, \text{PInstGamma } n \text{ ots } (t \# ts) \text{ False}) \Rightarrow \\ & \quad \text{Some } \{ | (\text{sub } 0 \ t \ p \ \# \ z, \text{PInstGamma } n \ \text{ots } \ ts \ \text{True}) \} \} \\ & \quad | (-, -) \Rightarrow \text{None} \rangle \\ & | \langle \text{effect GammaUni state} = (\text{case state of} \\ & \quad (\text{Neg } (\text{Uni } p) \# z, \text{PInstGamma } n \ \text{ots } (t \# ts) \text{ False}) \Rightarrow \\ & \quad \text{Some } \{ | (\text{Neg } (\text{sub } 0 \ t \ p) \# z, \text{PInstGamma } n \ \text{ots } \ ts \ \text{True}) \} \} \\ & \quad | (-, -) \Rightarrow \text{None} \rangle \end{aligned}$$

Each rule is applicable if the first formula in the sequent matches the pattern of the proof rule, the list of terms to instantiate with is not empty, and the flag is set to False. The rules instantiate the quantifier in the first formula in the sequent with the next term from the list of terms to instantiate with, remove the term from the list of terms to instantiate with, and set the flag to True.

The effect function implements the proof rule restrictions in algorithm 4.

### 3.2.4 The rule stream

Since the rules are restricted by the effect function such that they are all mutually exclusive, the exact order in which the prover attempts to apply the rules does not matter much. Hence the infinite list of rules to attempt to apply can simply be the list of all rules in some arbitrary order, repeated forever:

**definition** *rulesList* **where**

$$\langle \text{rulesList} = [ \text{Basic}, \\ \quad \text{AlphaDis}, \text{AlphaImp}, \text{AlphaCon}, \text{BetaCon}, \text{BetaImp}, \text{BetaDis}, \text{DeltaUni}, \text{DeltaExi}, \text{NegNeg}, \\ \quad \text{GammaExi}, \text{GammaUni}, \\ \quad \text{Rotate}, \text{Duplicate}, \text{Next} ] \rangle$$

**definition** *rules* **where**

$$\langle \text{rules} = \text{cycle rulesList} \rangle$$

### 3.2.5 The prover function

Once it has been proven that the effect function and the infinite list of rules to apply satisfy the criteria set by the abstract completeness framework, the actual prover function can be created using the facilities provided by the framework. The framework provides a function to make a proof tree and a fair enumeration of rules to attempt to apply, and these can be combined to create the prover:

**definition**  $\langle \text{secaVProver} \equiv \lambda x . \text{mkTree fenum } (x, \text{PBasic}) \rangle$

The prover always starts in the Basic phase, following the conditions in algorithm 4. This prover is not actually executable, but represents the prover in proofs. We will see how an executable version of this prover is obtained in the next section.

## 3.3 The Haskell program

The prover itself is specified in an Isabelle program corresponding to algorithm 4 as described above, but this program is not very useful by itself. First of all, the formula to prove must be given in the slightly inconvenient full SeCaV syntax, and we would like

to allow users to input formulas in SeCaV Unshortener syntax. Much more importantly, the result of a successful application of the prover results in a proof tree, while an actual SeCaV proof is linear with branches carried along at each step. To obtain a SeCaV proof we thus need to linearize the proof tree. The nodes of the proof tree are labelled by sequents, pseudo-rules, and phases, but actual SeCaV proofs involve sequents and SeCaV rules. Some post-processing is thus also needed to replace pseudo-rules by actual SeCaV rules and remove the now superfluous phase information.

These tasks are performed by a Haskell library, which is integrated with the prover and a small command-line interface to obtain the fully functional prover application.

### 3.3.1 Extracting the prover

The first step is to extract an executable program library from the Isabelle implementation of the prover, which is not executable. Isabelle allows for extraction to a number of functional programming languages including Haskell. We choose Haskell due to its good support for coinduction and its advanced type system, which particularly allows for programming with side effects through the State monad and elegant parsing through monadic parser combinators.

The coinductive stream type of the abstract completeness framework in Isabelle is translated into a lazy list type in Haskell to enable computation with coinductive streams:

**code-lazy-type** *stream*

We also need to define how to actually compute if something is a member of an infinite list:

**declare** *Stream.smember-code* [code del]

**lemma** [code]: *Stream.smember*  $x$  ( $y \## s$ ) = ( $x = y \vee \text{Stream.smember } x s$ )

**unfolding** *Stream.smember-def* **by** *auto*

The option type in Isabelle is translated into the Maybe monad in Haskell, and this needs to be imported in the relevant modules in Haskell:

**code-printing**

**constant** *the*  $\rightarrow$  (*Haskell*) *MaybeExt.fromJust*

| **constant** *Option.is-none*  $\rightarrow$  (*Haskell*) *MaybeExt.isNothing*

**code-printing code-module** *MaybeExt*  $\rightarrow$  (*Haskell*)

*(module MaybeExt(fromJust, isNothing) where*

*import Data.Maybe(fromJust, isNothing);)*

Since several modules depend on each other, and to simplify the Haskell code working with the prover, we collect most of the Isabelle modules into a single Haskell module:

**code-identifier**

**code-module** *Stream*  $\rightarrow$  (*Haskell*) *Prover*

| **code-module** *Prover*  $\rightarrow$  (*Haskell*) *Prover*

| **code-module** *Export*  $\rightarrow$  (*Haskell*) *Prover*

| **code-module** *MaybeExt*  $\rightarrow$  (*Haskell*) *Prover*

| **code-module** *Abstract-Completeness*  $\rightarrow$  (*Haskell*) *Prover*

Finally, we need to use the executable version of the abstract completeness framework facilities to define the prover:

**definition** *(rhoCode*  $\equiv$  *i.fenum rules)*

**definition** *(secaVTreeCode*  $\equiv$  *i.mkTree effect rhoCode)*

**definition** ( $secavProverCode \equiv \lambda x . secavTreeCode (x, PBasic)$ )

Once these translations have been set up in Isabelle, the prover program can be extracted into a Haskell library using the export functionality of the Isabelle build system:

**export-code open** *secavProverCode* **in** *Haskell*

### 3.3.2 The parsing and extraction library

The prover is supported by a Haskell library for parsing and extracting formulas and proofs. The library consists of a number of modules that parse SeCaV formulas and proofs, extract a SeCaV proof from the proof tree generated by the prover, and unshorten a proof into a full derivation in the SeCaV Isabelle syntax.

**The ShortParser module** contains a parser for the SeCaV Unshortener syntax. The parser is implemented using the Parsec library for monadic parser combinators. The ShortParser module exposes a parser for single formulas for parsing input to the prover and a parser for entire proofs for parsing proofs if the user wishes to generate a full SeCaV proof to verify in Isabelle.

**The SeCaVTranslator module** contains functions that translate SeCaV Unshortener formulas with named functions and predicates into SeCaV formulas with functions and predicates denoted by de Bruijn indices. This prepares the parsed formula for the prover, which can then be run on the formula.

**The ProofExtractor module** contains functions that turn a proof tree from the prover into a linear SeCaV proof. The first step is to perform some tree surgery to remove NEXT rules, which are completely irrelevant to the actual proof. Next, the pseudo-rules are translated into real SeCaV rules. The BASIC rule and the  $\alpha$ -,  $\beta$ -, and  $\delta$ -rules can be translated directly, since they are completely identical to the SeCaV rules. ROTATE and DUPLICATE rules can both be translated into EXT rules.

The difficult part is  $\gamma$ -rules, which must be annotated with the term that the formula is instantiated with by applying the rule. This term can be extracted from the phase as the first element in the list of terms that the formula has not yet been instantiated with, since this the rule uses the first element in the list to instantiate with when applied. The final step before linearization is to collapse subsequent uses of the EXT rule, which will be very common due to the use of the ROTATE pseudo-rule, which moves only a single formula at a time. This is not necessary, but shortens proofs considerably.

After the tree surgery step, a linear proof is extracted from the proof tree by inductively traversing the proof tree and “folding” in the branches in each proof step. The extraction function returns a textual representation of the proof in SeCaV Unshortener syntax.

The Haskell functions performing the removal, substitution and collapsing of rules, and the extraction function, do not return meaningful information for all possible inputs. This is because Haskell does not allow us to prove that the proof rules never generate more than two branches, which causes issues due to the encoding of the proof tree using finite sets in the abstract completeness framework. It is very clear from the definition of the effect function, however, that the missing cases will never occur, since all proof rules generate a finite set containing no more than two branches when applied.

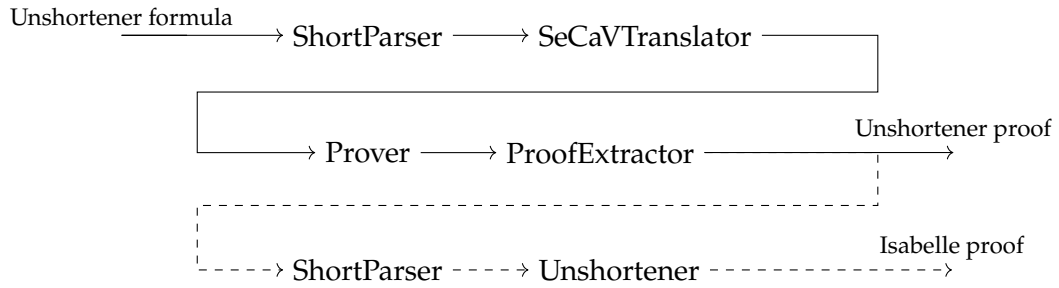


Figure 3.1: The structure of the prover application. The solid lines indicate steps that are always done, while the dashed lines indicate steps that are optionally enabled.

**The Unshortener module** contains functions that turn a proof in SeCaV Unshortener syntax into a proof in SeCaV Isabelle syntax. This module is essentially an embedded version of the SeCaV Unshortener online tool. The functions recursively move through the abstract syntax tree of the proof and turn each proof step into a textual representation in SeCaV Isabelle syntax.

### 3.3.3 The main application

The main application is a very simple command line interface implemented using an applicative combinator library for parsing command line options. It consists of a single module, `Main`, which is the entry point of the application. This module essentially just composes the modules in the parsing and extraction library to obtain a “pipeline” for feeding the prover and consuming its results. Figure 3.1 contains an overview of the pipeline.

The application takes a formula to attempt to prove (in SeCaV Unshortener syntax) as the only mandatory argument. If no other arguments are provided, the application prints a proof in SeCaV Unshortener syntax (looping forever if a proof is not found).

The application takes a single optional argument, `-i FILENAME`, which instructs the application to generate a proof in SeCaV Isabelle syntax and save it to `FILENAME` instead. This proof can then be verified by opening it in Isabelle in a session that has access to SeCaV. Using this option makes it more difficult to read the proofs since the SeCaV Isabelle syntax uses de Bruijn indices instead of names. On the other hand, the proofs can actually be verified when using this option. It is of course also always possible to generate proofs in SeCaV Unshortener syntax and translate them into SeCaV Isabelle later on with the SeCaV Unshortener online tool or by hand.



## 4 Soundness

The proof system of the Sequent Calculus Verifier has been proven sound in [1]. The soundness proof for the system shows that the system is sound for sequents, but our prover uses not only a sequent, but also a proof phase as its proof state. Of course the proof phase has no impact on the final proof, since it is exclusively used to guide the prover during the proof process.

The abstract completeness framework, that we will use for the proof of completeness, has a corresponding abstract soundness framework which can be used to prove soundness of the proofs generated by the prover resulting from the abstract completeness framework.

Due to time constraints, this abstract soundness framework was not used to attempt to prove soundness of the prover. Instead, we describe here an informal sketch of how the framework could be used, with a corresponding Isabelle sketch of the formalization. The keyword “sorry” instructs Isabelle to admit a result without proof, and will be used throughout the proof sketch.

The abstract soundness framework expects a semantics function for sequents, but the Sequent Calculus Verifier defines semantics only for a single formula. The semantics function for a sequent is simple to define inductively on the definition of a sequent, which is just a list of formulas:

```
fun ssemantics :: (nat  $\Rightarrow$  'a)  $\times$  (nat  $\Rightarrow$  'a list  $\Rightarrow$  'a)  $\times$  (nat  $\Rightarrow$  'a list  $\Rightarrow$  bool)  $\Rightarrow$  state  $\Rightarrow$  bool
where
  (ssemantics (e,f,g) ([],-) = False)
  | (ssemantics (e,f,g) (p # z),phase) = (semantics e f g p  $\vee$  ssemantics (e,f,g) (z,phase))
```

An empty sequent is false. A non-empty sequent is true if the semantics of the first formula in the sequent are true or if the the semantics of the rest of the sequent are true.

To use the abstract soundness framework, an interpretation must be given for the effect function, the infinite list of rules to apply, and the semantics of sequents:

```
interpretation Soundness eff rules UNIV ssemantics
unfolding Soundness-def
```

To do so we need to prove local soundness of the proof rules. Local soundness of the proof system means that, if the sequent in each branch generated by applying a proof rule are valid, then so is the sequent that the proof rule was applied to:

```
proof (safe)
  fix r sequent phase sl f g and e :: (nat  $\Rightarrow$  'a)
  assume r-rule: (r  $\in$  R)
  assume r-enabled: (eff r (sequent, phase) sl)
  assume next-sound: ( $\forall$  s'. s'  $\in$  | sl  $\longrightarrow$  ( $\forall$  S  $\in$  UNIV. ssemantics S s'))
  show (ssemantics (e, f, g) (sequent, phase))
  proof (cases sequent)
```

For empty sequents, the proof is by contradiction on the assumption that the sequents in the branches generated by application of a proof rule to the empty sequent are valid:

```

case Nil
assume empty: ⟨sequent = []⟩
show ⟨ssemantics (e, f, g) (sequent, phase)⟩
proof (rule ccontr)
  note ⟨∀ s'. s' |∈| sl ⟶ (∀ S ∈ UNIV. ssemantics S s')⟩
  have ⟨eff r ([], phase) sl ⟹ (∀ s'. s' |∈| sl ⟶ fst s' = [])⟩
  sorry
  moreover have ⟨∀ s'. fst s' = [] ⟶ (¬ (∀ S ∈ UNIV. ssemantics S s'))⟩
  sorry
  ultimately show (False) using next-sound r-enabled r-rule
  sorry
qed

```

The idea is to prove that application of a proof rule to an empty sequent will always result in a branch with an empty sequent, which thus cannot be valid, contradicting the assumption.

For non-empty sequents local soundness should be quite easy to prove, since the formulas in the branches returned by the effect function for each proof rule is essentially the same as the semantics for the related connective or quantifier:

```

next
case (Cons p z)
assume ⟨sequent = p # z⟩
show ⟨ssemantics (e, f, g) (sequent, phase)⟩
sorry
qed
qed

```

The abstract soundness framework can be used to prove that locally sound proof systems give rise to proof trees which have valid root sequents if they are finite and well-formed:

```

theorem prover-soundness:
  fixes t
  assumes f: ⟨tfinite t⟩ and w: ⟨wf t⟩
  shows ⟨∀ i. ssemantics i (fst (root t))⟩
  sorry

```

**end**

Combined with the proof of completeness, which shows that the prover will always return a finite and well-formed proof tree if the initial sequent is valid, this shows that the proofs generated by the prover will have valid root sequents.

To obtain some confidence that the prover is sound, a test suite containing a number of invalid formulas has been run on the prover. This test suite and its results will be detailed in section 6.3.

## 5 Completeness

The major problem is proving that the prover is complete in the sense that it can prove any valid SeCaV formula. We will use the abstract completeness framework for the proof. To do so we first need to prove that our prover satisfies the requirements set by the framework.

The first step is simply a rule system definition, which fixes the effect of the rules `eff` and a rule stream rules to apply. The abstract completeness framework expects a Boolean effect relation, while our effect function returns an optional result. We adapt the effect function as follows to accommodate this:

**definition** *eff where*

*(eff ≡ λr s ss. effect r s = Some ss)*

The next step is a rule system, which additionally fixes a set of possible states  $S$ . For the rule system, we need to prove that  $S$  is closed under `eff`, and the property of enabledness: that for all states in  $S$ , there is some rule in `rules` which is enabled.

The final step is a persistent rule system, for which we must additionally prove that all rules are persistent. Persistency means that once a rule is enabled, it will stay enabled even if another rule is applied first.

### 5.1 Enabledness

For our prover to have a rule system in the sense of the abstract completeness framework, it needs to have the property of enabledness. Algorithm 4 is designed such that exactly one rule is enabled at all times, so our system does have the required property. We instantiate the definition of a rule system from abstract completeness framework by proving that the effect function has the property of enabledness:

**interpretation** *RuleSystem eff rules UNIV*

**unfolding** *rules-def RuleSystem-def*

**proof** (*simp*)

**show**  $(\forall \text{sequent phase}. \exists r \in i.R (\text{cycle rulesList}). \exists sl. \text{eff } r (\text{sequent}, \text{phase}) sl)$

**proof** (*intro allI*)

**fix** *sequent phase*

**show**  $(\exists r \in i.R (\text{cycle rulesList}). \exists sl. \text{eff } r (\text{sequent}, \text{phase}) sl)$

The goal is now to prove that, for some fixed `sequent` and `phase`, there is always some proof rule in the list of rules to attempt to apply which can actually be applied. We prove this by induction over the `phase`:

**proof** (*induct phase*)

The first case is the Basic phase, for which we next perform induction on the `sequent`:

**case** *PBasic*

**then show** *?case unfolding eff-def*

**proof** (*induct sequent*)

If the `sequent` is empty, the proof is now trivial:

**case** *Nil*

**then show** *?case unfolding rulesList-def by simp*

If the sequent is not empty, it consists of a formula at the head of the list, and a list containing the rest of the formulas in the sequent:

```

next
  case (Cons p z)
  then show ?case

```

We now need to determine which proof rule is enabled, which depends on two conditions: whether the branch can be closed by applying a combination of ROTATE and BASIC rules and, if so, whether the current proof state is not one in which a BASIC rule can be applied:

```

proof (cases (branchDone (p # z)))
  case bd: True
  show ?thesis
  proof (cases (Neg p  $\notin$  set z))

```

If it is possible to close the branch by applying ROTATE rules followed by a BASIC rule, and the BASIC rule is not enabled in the current proof state, the prover should apply a ROTATE rule, and we proceed by proving that this is possible by case analysis on the first formula in the sequent:

```

  case neg: True
  then show ?thesis
  proof –
    have (Rotate  $\in$  i.R (cycle rulesList))
    unfolding rulesList-def by simp
    moreover have (effect Rotate (p # z, PBasic) = Some { | (z @ [p], PBasic) | })
    using bd neg
    proof (cases p)
    case (Neg q)
    with neg bd show ?thesis by (cases q) simp-all
    qed simp-all
    ultimately show ?thesis unfolding rulesList-def by simp
  qed

```

If the BASIC rule is enabled, the proof is trivial:

```

  next
  case False
  then show ?thesis unfolding rulesList-def by simp
  qed

```

If the branch cannot be closed by BASIC and ROTATE rules in the current proof state, the NEXT rule can always be applied, and the proof is again trivial:

```

  next
  case False
  then show ?thesis unfolding rulesList-def
  by (simp split: fm.splits)
  qed
  qed

```

This concludes the case of the Basic phase.

The next case in the proof is the Alpha–Beta–Delta phase. Since the rules enabled in this case depend solely on the shape of the first formula in the sequent, the proof is by induction on the shape of the sequent, and then by trivial case analysis of the first formula in it:

```

next
  case PABD
  then show ?case unfolding rulesList-def eff-def
  proof (induct sequent)
    case Nil
    show ?case by simp
  next
  case (Cons p z)
  show ?case
  proof (cases p)
    case (Neg q)
    then show ?thesis
    by (cases q) simp-all
  qed simp-all
qed

```

The next case is the Gamma Preparation phase. In this case, the rule the prover chooses to apply depends on the number of remaining formulas, so the proof is by induction on this number:

```

next
  case (PPrepGamma n ts)
  then show ?case unfolding eff-def
  proof (induct n)

```

If the number is zero, the NEXT rule is chosen, and the rest of the proof is by cases on the shape of the sequent, since the NEXT rule is enabled for empty and non-empty sequents in two separate cases in the effect function:

```

  case 0
  then show ?case unfolding rulesList-def
  by (cases sequent; simp split: fm.splits)

```

If the number is not zero, the enabled rule actually depends on the shape of the sequent, so the proof proceeds by case analysis on the sequent:

```

  next
  case n': (Suc n')
  then show ?case
  proof (cases sequent)

```

For an empty sequent, the NEXT rule is again enabled, and the rest of the proof is trivial:

```

  case p: Nil
  then show ?thesis unfolding p rulesList-def by simp

```

If the sequent is not empty, the prover will either apply the DUPLICATE rule or the ROTATE rule depending on the first formula in the sequent.

The proof thus proceeds by case analysis on this formula:

```

next
  case (Cons p z)
  then show ?thesis
  proof simp
  show  $(\exists r \in i.R \text{ (cycle rulesList)}).$ 
     $\exists sl. \text{effect } r \text{ (p \# z, PPrepGamma (Suc n') ts) = Some sl}$ 
  proof (cases p)
  case p: (Neg q)
  then show ?thesis unfolding rulesList-def
  by (cases q) simp-all
  qed (unfold rulesList-def, simp-all)
qed
qed
qed

```

This concludes the case of the Gamma Preparation phase.

The final case is the Gamma Instantiation phase. In this phase, the enabled rule depends on the list of remaining terms to instantiate, so the proof proceeds by induction on the shape of this list:

```

next
  case (PInstGamma n ots ts b)
  then show ?case unfolding eff-def
  proof (induct ts)

```

If the list is empty, the rule depends on the value of the Boolean flag and the shape of the sequent, but the proof is by simple case analysis on the value and the shape:

```

  case Nil
  then show ?case unfolding rulesList-def
  by (cases b; cases sequent; simp split: list.splits fm.splits bool.splits)

```

If the list is non-empty, the rule also depends on the value of the Boolean flag and the shape of the sequent, but the proof is now slightly more complicated, since the rule also depends on the shape of the first formula in the sequent:

```

next
  case (Cons t ts')
  then show ?case
  proof (cases b)
  case bt: True
  then show ?thesis unfolding rulesList-def
  by (cases sequent; simp add: bt split: fm.splits)
  next
  case bf: False
  then show ?thesis
  proof (cases sequent)
  case Nil
  then show ?thesis unfolding rulesList-def
  by (simp add: bf)
  next
  case ss: (Cons p z)
  then show ?thesis
  proof (cases p)
  case pneg: (Neg q)
  then show ?thesis unfolding rulesList-def

```

```

    by (cases q; simp add: bf ss pneg)
  qed (unfold rulesList-def, simp-all add: bf ss)
qed
qed
qed
qed
qed
qed

```

This concludes the proof of the enabledness property, and allows our prover to use the parts of the abstract completeness framework depending on this property.

## 5.2 Persistency

To actually use the abstract completeness framework, we also need to prove that the effect function and the list of proof rules to attempt to apply have the property of persistency. The property is defined as follows:

$$\text{per } r \equiv \forall s r_1 sl' s'. s \in S \wedge \text{enabled } r s \wedge r_1 \in R - \{r\} \wedge \text{eff } r_1 s sl' \wedge s' \in sl' \implies \text{enabled } r s'$$

If we can prove that no more than one rule is ever enabled at any time, this vacuously implies persistency, since there will never be another rule to apply “instead” of the enabled rule.

We prove the property in the following lemma, which turned out to be significantly more complicated than the enabledness property:

**lemma** *enabled-unique*:

$$\langle \forall r \text{ sequent phase. enabled } r (\text{sequent}, \text{phase}) \longrightarrow (\forall r' \in R - \{r\}. \neg \text{enabled } r' (\text{sequent}, \text{phase})) \rangle$$

We first need to move the quantifiers and assumption into the meta-logic:

```

proof (intro allI impI)
  fix r sequent phase
  assume  $\langle \text{enabled } r (\text{sequent}, \text{phase}) \rangle$ 
  then show  $\langle \forall r' \in R - \{r\}. \neg \text{enabled } r' (\text{sequent}, \text{phase}) \rangle$ 

```

As before, the proof then proceeds by induction on the proof phase:

```

proof (induct phase)

```

The first case is the Basic phase, for which the proof immediately proceeds by case analysis on the contents of the sequent:

```

  case pb: PBasic
  then show ?case
  proof (cases sequent)

```

If the sequent is empty, we again need to move quantifiers and assumptions into the meta-logic:

```

    case sn: Nil
    assume r-enabled:  $\langle \text{enabled } r (\text{sequent}, \text{PBasic}) \rangle$ 
    assume  $\langle \text{sequent} = [] \rangle$ 
    then show ?thesis
    proof (intro ballI)
      fix r'
      assume empty:  $\langle \text{sequent} = [] \rangle$ 
      assume not-eq:  $\langle r' \in R - \{r\} \rangle$ 
      show  $\langle \neg \text{enabled } r' (\text{sequent}, \text{PBasic}) \rangle$ 

```

We now need to prove that, given that  $r$  is some enabled rule and  $r' \neq r$ , then  $r'$  cannot be enabled. We will prove this by contradiction, first assuming that  $r'$  is enabled:

```
proof (rule ccontr, simp)
  assume  $r'$ -enabled: ⟨enabled  $r'$  (sequent, PBasic)⟩
```

Next, we prove that if  $r'$  is enabled, it *must* be the NEXT rule by induction on  $r'$ :

```
have ⟨enabled  $r'$  ([], PBasic) ⟹  $r' = \text{Next}$ ⟩
  unfolding enabled-def eff-def RuleSystem-Defs.enabled-def
  by (induct  $r'$ ) simp-all
```

Since we have assumed that  $r'$  is enabled, we can then show that  $r'$  is the NEXT rule:

```
then have ( $r' = \text{Next}$ ) using  $r'$ -enabled empty by simp
```

We can of course prove exactly the same for  $r$ , since we have also assumed that  $r$  is enabled, and this leads to a contradiction of the assumption that  $r' \neq r$ :

```
moreover have ⟨enabled  $r$  ([], PBasic) ⟹  $r = \text{Next}$ ⟩
  unfolding enabled-def eff-def RuleSystem-Defs.enabled-def
  by (induct  $r$ ) simp-all
ultimately show False using not-eq  $r$ -enabled empty by simp
qed
qed
```

If the sequent is non-empty, we again need to move quantifiers and assumptions into the meta-logic:

```
next
case (Cons  $p z$ )
fix  $p z$ 
assume  $r$ -enabled: ⟨enabled  $r$  (sequent, PBasic)⟩
assume content: ⟨sequent =  $p \# z$ ⟩
then show ?thesis
proof (intro ball)
  fix  $r'$ 
  assume not-eq: ( $r' \in R - \{r\}$ )
  show ⟨ $\neg$  enabled  $r'$  (sequent, PBasic)⟩
```

To determine which rule the prover should apply in this case, we need to know whether the branch can be closed by application of ROTATE rules and a BASIC rule, so we proceed by case analysis on the result of the branchDone function and case analysis on whether the negation of the first formula in the sequent is contained in the sequent:

```
proof (cases ⟨branchDone ( $p \# z$ )⟩)
  case bd: True
  then show ?thesis
  proof (cases ⟨Neg  $p \notin \text{set } z$ ⟩)
```

If the branchDone function indicates that it is possible to close the branch, but the sequent does not contain the negation of the current first formula in the sequent, the prover should apply the ROTATE rule.



The proof that no other rule application is possible is again by contradiction, and follows exactly the same procedure as above except that we now also need case analysis on the first formula in the sequent and to split case structures on formulas:

```

case neg: True
then show ?thesis
proof –
  show  $\langle \neg \text{enabled } r' \text{ (sequent, PBasic)} \rangle$ 
  proof (rule ccontr, simp)
    assume r'-enabled:  $\langle \text{enabled } r' \text{ (sequent, PBasic)} \rangle$ 
    have  $\langle \text{enabled } r' \text{ (p \# z, PBasic)} \implies r' = \text{Rotate} \rangle$ 
      unfolding enabled-def eff-def RuleSystem-Defs.enabled-def
      using bd neg by (induct r'; cases p; simp split: fm.splits)
    then have  $\langle r' = \text{Rotate} \rangle$  using r'-enabled content by simp
    moreover have  $\langle \text{enabled } r \text{ (p \# z, PBasic)} \implies r = \text{Rotate} \rangle$ 
      unfolding enabled-def eff-def RuleSystem-Defs.enabled-def
      using neg bd by (induct r; cases p; simp split: fm.splits)
    ultimately show False using not-eq r-enabled content by simp
  qed
qed

```

If the `branchDone` function indicates that it is possible to close the branch, and the sequent contains the negation of the first formula in the sequent, the prover should apply the `BASIC` rule to close the branch. The proof that no other rule application is possible is almost identical to the one above:

```

next
case neg: False
then show ?thesis
proof –
  show  $\langle \neg \text{enabled } r' \text{ (sequent, PBasic)} \rangle$ 
  proof (rule ccontr, simp)
    assume r'-enabled:  $\langle \text{enabled } r' \text{ (sequent, PBasic)} \rangle$ 
    have  $\langle \text{enabled } r' \text{ (p \# z, PBasic)} \implies r' = \text{Basic} \rangle$ 
      unfolding enabled-def eff-def RuleSystem-Defs.enabled-def
      using bd neg
      by (induct r'; simp split: fm.splits)
    then have  $\langle r' = \text{Basic} \rangle$  using r'-enabled content by simp
    moreover have  $\langle \text{enabled } r \text{ (p \# z, PBasic)} \implies r = \text{Basic} \rangle$ 
      unfolding enabled-def eff-def RuleSystem-Defs.enabled-def
      using neg bd by (induct r; cases p; simp split: fm.splits)
    ultimately show False using not-eq r-enabled content by simp
  qed
qed
qed

```

If the `branchDone` function indicates that it is not possible to close the branch, and the sequent does not contain the negation of the first formula in the sequent, the prover should apply the `NEXT` rule to advance to the next phase.

The proof that no other rule is enabled is again almost identical:

```

next
case bd: False
then show ?thesis
proof (cases (Neg p  $\notin$  set z))
case neg: True
then show ?thesis
proof -
show  $\langle \neg \text{enabled } r' \text{ (sequent, PBasic)} \rangle$ 
proof (rule ccontr, simp)
assume r'-enabled:  $\langle \text{enabled } r' \text{ (sequent, PBasic)} \rangle$ 
have  $\langle \text{enabled } r' \text{ (p \# z, PBasic)} \rangle \implies r' = \text{Next}$ 
  unfolding enabled-def eff-def RuleSystem-Defs.enabled-def
  using bd neg by (induct r'; simp split: fm.splits)
then have  $\langle r' = \text{Next} \rangle$  using r'-enabled content by simp
moreover have  $\langle \text{enabled } r \text{ (p \# z, PBasic)} \rangle \implies r = \text{Next}$ 
  unfolding enabled-def eff-def RuleSystem-Defs.enabled-def
  using neg bd by (induct r; cases p; simp split: fm.splits)
ultimately show False using not-eq r-enabled content by simp
qed
qed

```

Finally, if the branchDone function indicates that it is not possible to close the branch, but the sequent does contain the negation of the first formula in the sequent, we have a contradiction, since this situation is not possible due to the definition of the branchDone function. This is where we need our earlier lemma about the branchDone function:

```

next
case neg: False
then show ?thesis
proof -
show  $\langle \neg \text{enabled } r' \text{ (sequent, PBasic)} \rangle$ 
proof (rule ccontr)
show False using pinz-done bd neg by simp
qed
qed
qed
qed
qed

```

This concludes the case of the Basic phase.

The next case is the Alpha–Beta–Delta phase. Again we first proceed by case analysis on the contents of the sequent:

```

next
case pabd: PABD
then show ?case
proof (cases sequent)

```

If the sequent is empty, the prover should again apply the NEXT rule, and the proof that this is the only option is very similar to the previous one:

```

case sn: Nil
assume r-enabled:  $\langle \text{enabled } r \text{ (sequent, PABD)} \rangle$ 
assume  $\langle \text{sequent} = [] \rangle$ 
then show ?thesis

```

```

proof (intro ball)
  fix r'
  assume empty: ⟨sequent = []⟩
  assume not-eq: ⟨r' ∈ R - {r}⟩
  show ⟨¬ enabled r' (sequent, PABD)⟩
  proof (rule ccontr, simp)
    assume r'-enabled: ⟨enabled r' (sequent, PABD)⟩
    have ⟨enabled r' ([], PABD) ⟹ r' = Next⟩
      unfolding enabled-def eff-def RuleSystem-Defs.enabled-def
      by (induct r') simp-all
    then have ⟨r' = Next⟩ using r'-enabled empty by simp
    moreover have ⟨enabled r ([], PABD) ⟹ r = Next⟩
      unfolding enabled-def eff-def RuleSystem-Defs.enabled-def
      by (induct r) simp-all
    ultimately show False using not-eq r-enabled empty by simp
  qed
qed

```

If the sequent is non-empty, we need to examine the first formula in the sequent to determine whether the formula can be decomposed by applying an Alpha-, Beta-, or Delta-rule:

```

next
  case (Cons p z)
  fix p z
  assume r-enabled: ⟨enabled r (sequent, PABD)⟩
  assume content: ⟨sequent = p # z⟩
  then show ?thesis
  proof (intro ball)
    fix r'
    assume not-eq: ⟨r' ∈ R - {r}⟩
    show ⟨¬ enabled r' (sequent, PABD)⟩
    proof (cases p)

```

If the first formula is a predicate, it cannot be decomposed further. To determine which rule the prover should apply in this case, we need to know whether there are any decomposable formulas left elsewhere in the sequent:

```

  case pre: (Pre n ts)
  then show ?thesis
  proof (cases ⟨abdDone (p # z)⟩)

```

If there are not, the prover should apply the next rule, and the proof that this is the only option is familiar at this point:

```

  case d: True
  show ?thesis
  proof (rule ccontr, simp)
    assume r'-enabled: ⟨enabled r' (sequent, PABD)⟩
    have ⟨enabled r' (p # z, PABD) ⟹ r' = Next⟩
      unfolding enabled-def eff-def RuleSystem-Defs.enabled-def
      using d by (induct r'; simp split: fm.splits)
    then have ⟨r' = Next⟩ using r'-enabled content by simp
    moreover have ⟨enabled r (p # z, PABD) ⟹ r = Next⟩
      unfolding enabled-def eff-def RuleSystem-Defs.enabled-def
      using d by (induct r; simp split: fm.splits)
    ultimately show False using not-eq r-enabled content by simp
  qed

```

If there are decomposable formulas left in the sequent, the prover should apply the ROTATE

rule, but for this to be enabled, the first formula in the sequent must not be decomposable, and so we proceed by case analysis on this condition before the by now familiar proof:

```

next
  case  $d$ : False
  then show ?thesis
  proof (cases ⟨abdDone [p]⟩)
    case  $pd$ : True
    show ?thesis
    proof (rule ccontr, simp)
      assume  $r'$ -enabled: ⟨enabled  $r'$  (sequent, PABD)⟩
      have ⟨enabled  $r'$  ( $p \# z$ , PABD)  $\implies r' = \text{Rotate}$ ⟩
      unfolding enabled-def eff-def RuleSystem-Defs.enabled-def
      using  $d$   $pd$  by (induct  $r'$ ; simp split: fm.splits)
      hence ⟨ $r' = \text{Rotate}$ ⟩ using  $r'$ -enabled content by simp
      moreover have ⟨enabled  $r$  ( $p \# z$ , PABD)  $\implies r = \text{Rotate}$ ⟩
      unfolding enabled-def eff-def RuleSystem-Defs.enabled-def
      using  $d$   $pd$  by (induct  $r$ ; simp split: fm.splits)
      ultimately show False using not-eq  $r$ -enabled content by simp
    qed

```

We now need to handle the other case, where the first formula in the sequent is composable. We do this by contradiction on the assumption that this is the case, since the first formula is a predicate and thus cannot be decomposed further:

```

next
  case  $pd$ : False
  show ?thesis
  proof (rule ccontr, simp)
    assume  $r'$ -enabled: ⟨enabled  $r'$  (sequent, PABD)⟩
    have notdone: ⟨ $\neg$  abdDone [p]⟩ using  $pd$  by simp
    have ⟨abdDone [Pre n ts]⟩ by simp
    have ⟨ $p = \text{Pre n ts}$ ⟩ using pre content by simp
    then have ⟨abdDone [p]⟩ by simp
    then show False using notdone by simp
  qed
qed
qed

```

The next three cases are implication, disjunction and conjunction, and these cases are all proven using the structure of the by now familiar proof by contradiction to show that the only applicable rules are ALPHAIMP, ALPHADIS, and BETA CON, respectively. The proofs are therefore omitted here:

```

next
  case  $p$ : (Imp  $a$   $b$ )
  (12 lines omitted)
next
  case  $p$ : (Dis  $a$   $b$ )
  (12 lines omitted)
next
  case  $p$ : (Con  $a$   $b$ )
  (12 lines omitted)

```

The next case is the existential quantifier. Just like predicates, formulas that are existentially quantified cannot be decomposed in this phase, so the prover should either apply a ROTATE rule or a NEXT rule depending on whether the sequent contains any decomposable formulas elsewhere. The proof is almost identical to the proof for the case of a predicate,

so we omit the proof here:

```
next  
  case  $p$ : ( $Exi\ q$ )  
(45 lines omitted)
```

The next case is the universal quantifier, for which we show that the only applicable rule is DELTAUNI, again using the same structure for the proof by contradiction as earlier. We omit the proof here:

```
next  
  case  $p$ : ( $Uni\ q$ )  
(12 lines omitted)
```

The final case is negation, for which we need to perform further case analysis on the negated formula to determine which rule to select:

```
next  
  case  $p$ : ( $Neg\ q$ )  
  then show ?thesis  
  proof (cases  $q$ )  
(168 lines omitted)
```

The cases in this case analysis all correspond exactly to the cases in the original case analysis of the first formula in the sequent. This is also true of the proofs, which are therefore omitted here. The only differences are that Alpha- and Beta-rules are switched due to the negation, that the negated universal quantifier can be decomposed while the negated existential quantifier cannot, and that the NEGNEG rule is used to decompose the double negation. This concludes the case of the Alpha–Beta–Delta-phase.

The next phase is the Gamma Preparation phase, for which we will once again proceed by case analysis on the contents of the sequent:

```
next  
  case  $pprep$ : ( $PPrepGamma\ n\ ts$ )  
  then show ?case  
  proof (cases sequent)
```

If the sequent is empty, the NEXT rule should be applied to advance to the next proof phase. The proof that this is the only option is by now familiar, and we omit the proof here:

```
case  $sn$ : Nil  
(20 lines omitted)
```

If the sequent is non-empty, the rule to apply depends on the remaining number of formulas in the sequent that have not yet been instantiated, and we thus proceed by moving quantifiers into the meta-logic, then performing case analysis on this number:

```
next  
  case ( $Cons\ p\ z$ )  
  assume  $r$ -enabled:  $\langle enabled\ r\ (sequent,\ PPrepGamma\ n\ ts) \rangle$   
  assume  $\langle sequent = p \# z \rangle$   
  then show ?thesis  
  proof (intro ballI)  
    fix  $r'$   
    assume content:  $\langle sequent = p \# z \rangle$   
    assume not-eq:  $\langle r' \in R - \{r\} \rangle$   
    show  $\langle \neg enabled\ r' (sequent,\ PPrepGamma\ n\ ts) \rangle$   
    proof (cases  $n$ )
```

If there are no more formulas left to attempt to instantiate, the `NEXT` rule should be applied to advance to a Basic phase. The proof that this is the only option follows the familiar recipe, except we now also need to split case structures on the list of terms in addition to splitting case structures on formulas:

```

case 0
assume n0: ⟨n = 0⟩
show ?thesis
proof (rule ccontr, simp)
  assume r'-enabled: ⟨enabled r' (sequent, PPrepGamma n ts)⟩
  have ⟨enabled r' (sequent, PPrepGamma 0 ts) ⟹ r' = Next⟩
    unfolding enabled-def eff-def RuleSystem-Defs.enabled-def
    by (induct r'; simp split: fm.splits list.splits)
  moreover have ⟨enabled r' (sequent, PPrepGamma 0 ts)⟩ using r'-enabled n0 by simp
  ultimately have ⟨r' = Next⟩ using content r'-enabled n0 by simp
  moreover have ⟨enabled r (sequent, PPrepGamma 0 ts) ⟹ r = Next⟩
    unfolding enabled-def eff-def RuleSystem-Defs.enabled-def
    by (induct r; simp split: fm.splits list.splits)
  ultimately show False using not-eq r-enabled content n0 by simp
qed

```

If there are still formulas left that the prover should try to instantiate, the rule to apply depends on the first formula in the sequent. If the first formula is quantified by an existential quantifier or a negated universal quantifier, the `DUPLICATE` rule should be applied, and the prover should advance to a Gamma Instantiation phase. In all other cases, the prover should apply the `ROTATE` rule and subtract one from the amount of formulas to be instantiated to move to the next formula in the sequent. We proceed by case analysis on the first formula in the sequent:

```

next
case n': (Suc n')
then show ?thesis
proof (cases p)

```

The proofs that these rules are the only options are a little more involved than the previous proofs by contradiction, but not much. The first case is a formula that is just a predicate:

```

case p: (Pre pn pts)
show ?thesis

```

The proof is by contradiction, so we first assume that  $r'$  is also enabled:

```

proof (rule ccontr, simp)
  assume r'-enabled: ⟨enabled r' (sequent, PPrepGamma n ts)⟩

```

We then show that, if  $r'$  is enabled for a predicate in this phase,  $r'$  must be the `ROTATE` rule by induction on  $r'$ :

```

  have ⟨enabled r' (Pre pn pts # z, PPrepGamma (Suc n') ts) ⟹ r' = Rotate⟩
    unfolding enabled-def eff-def RuleSystem-Defs.enabled-def
    by (induct r') simp-all

```

We then use that we are in the case where the sequent is non-empty and the first formula is a predicate, to show that  $r'$  is enabled:

```

  moreover have ⟨enabled r' (p # z, PPrepGamma (Suc n') ts)⟩
    using r'-enabled n' content by simp
  then have ⟨enabled r' (Pre pn pts # z, PPrepGamma (Suc n') ts)⟩
    using p by simp

```

This leads us to conclude that  $r'$  must be the ROTATE rule:

**ultimately have**  $\langle r' = Rotate \rangle$  **using** *content*  $r'$ -enabled  $n'$  **by** *simp*

The same argument can also be applied to  $r$ :

**moreover have**  $\langle enabled\ r\ (Pre\ pn\ pts\ \#z,\ PPrepGamma\ (Suc\ n')\ ts) \implies r = Rotate \rangle$   
**unfolding** *enabled-def eff-def RuleSystem-Defs.enabled-def*  
**by** *(induct r) simp-all*  
**moreover have**  $\langle enabled\ r\ (p\ \#z,\ PPrepGamma\ (Suc\ n')\ ts) \rangle$   
**using** *r-enabled n' content by simp*  
**then have**  $\langle enabled\ r\ (Pre\ pn\ pts\ \#z,\ PPrepGamma\ (Suc\ n')\ ts) \rangle$   
**using** *p by simp*

This leads to a contradiction of the fact that  $r' \neq r$ , since we have now proven that they must both be the ROTATE rule:

**ultimately show** *False* **using** *not-eq r-enabled content n' by simp*  
**qed**

The proofs of the rest of the cases are essentially identical, except the case of an existentially quantified formula, where the  $r'$  and  $r$  are the DUPLICATE rule instead of the ROTATE rule. We omit the proofs here:

**next**  
**case**  $p$ : *(Imp f1 f2)*  
*(20 lines omitted)*  
**next**  
**case**  $p$ : *(Dis f1 f2)*  
*(20 lines omitted)*  
**next**  
**case**  $p$ : *(Con f1 f2)*  
*(20 lines omitted)*  
**next**  
**case**  $p$ : *(Exi q)*  
*(20 lines omitted)*  
**next**  
**case**  $p$ : *(Uni q)*  
*(20 lines omitted)*

The only special case is the negated formula, for which we need to perform further case analysis on the negated subformula:

**next**  
**case**  $p$ : *(Neg q)*  
**then show** *?thesis*  
**proof** *(cases q)*

The proof that the only enabled rule for a negated predicate is the ROTATE rule is identical to the proof for a predicate except that we now also need to use that we know what the negated subformula is:

```

case  $q$ : (Pre pn pts)
show ?thesis
proof (rule ccontr, simp)
  assume  $r'$ -enabled: (enabled  $r'$  (sequent, PPrepGamma n ts))
  have (enabled  $r'$  (Neg (Pre pn pts) # z, PPrepGamma (Suc n') ts)  $\implies$   $r' = \text{Rotate}$ )
    unfolding enabled-def eff-def RuleSystem-Defs.enabled-def
    by (induct  $r'$ ) simp-all
  moreover have (enabled  $r'$  (p # z, PPrepGamma (Suc n') ts))
    using  $r'$ -enabled  $n'$  content by simp
  then have (enabled  $r'$  (Neg (Pre pn pts) # z, PPrepGamma (Suc n') ts))
    using p q by simp
  ultimately have ( $r' = \text{Rotate}$ ) using content  $r'$ -enabled  $n'$  by simp
  moreover have (enabled  $r$  (Neg (Pre pn pts) # z, PPrepGamma (Suc n') ts)  $\implies$   $r = \text{Rotate}$ )
    unfolding enabled-def eff-def RuleSystem-Defs.enabled-def
    by (induct  $r$ ) simp-all
  moreover have (enabled  $r$  (p # z, PPrepGamma (Suc n') ts))
    using  $r$ -enabled  $n'$  content by simp
  then have (enabled  $r$  (Neg (Pre pn pts) # z, PPrepGamma (Suc n') ts))
    using p q by simp
  ultimately show False using not-eq  $r$ -enabled content  $n'$  by simp
qed

```

The proofs of the rest of the cases are almost identical, and we omit them here:

```

  next
    case  $q$ : (Imp a b)
    (20 lines omitted)
  next
    case  $q$ : (Dis a b)
    (20 lines omitted)
  next
    case  $q$ : (Con a b)
    (20 lines omitted)
  next
    case  $q$ : (Exi a)
    (20 lines omitted)
  next
    case  $q$ : (Uni a)
    (20 lines omitted)
  next
    case  $q$ : (Neg a)
    (20 lines omitted)
  qed
qed
qed
qed
qed

```

This concludes the case of the Gamma Preparation phase.



The final case of the proof of this lemma is the Gamma Instantiation phase. As in the other cases, we proceed by case analysis on the contents of the sequent:

```

next
  case (PInstGamma n ots ts b)
  then show ?case
  proof (cases sequent)

```

If the sequent is empty, the prover should apply a NEXT rule, but we need to distinguish a few cases to prove that this is what happens. We first proceed by case analysis on the value of the Boolean flag:

```

  case Nil
  assume r-enabled: ⟨enabled r (sequent, PInstGamma n ots ts b)⟩
  assume empty: ⟨sequent = []⟩
  show ?thesis
  proof (cases b)

```

If the value of the flag is True, we proceed by the familiar proof by contradiction, which we omit here:

```

    case b: True
    show ?thesis
    (13 lines omitted)

```

If the value of the flag is False, we additionally need to perform case analysis on the contents of the list of terms to instantiate formulas with:

```

  next
  case b: False
  show ?thesis
  proof (cases ts)

```

In both cases, we simply proceed with the familiar proof by contradiction to show that the NEXT rule is the only enabled option. We omit the proofs here:

```

    case Nil
    (17 lines omitted)
  next
  case (Cons t' ts')
  (18 lines omitted)
  qed
  qed

```

If the sequent is non-empty, the enabled rule is actually determined by the Boolean flag, so we again need to proceed by case analysis on the value of the flag:

```

  next
  case (Cons p z)
  fix p z
  assume r-enabled: ⟨enabled r (sequent, PInstGamma n ots ts b)⟩
  assume content: ⟨sequent = p # z⟩
  then show ?thesis
  proof (cases b)

```

If the value of the flag is True, the prover will apply the ROTATE rule to move the already instantiated formula out of the way.

The proof that this is the only option is the familiar proof by contradiction, which we omit here:

```
case b: True
(14 lines omitted)
```

If the value of the flag is `False`, the rule to apply depends on the contents of the list of terms to instantiate formulas with:

```
next
case b: False
show ?thesis
proof (cases ts)
```

If the list of terms is empty, no more formulas should be instantiated, and the prover should apply the `NEXT` rule. Unfortunately we need to show that this is what will happen for each possible first formula in the sequent separately, so we proceed by induction on the shape of the first formula in the sequent:

```
case Nil
assume ts: (ts = [])
show ?thesis
proof (induct p)
```

The proof of almost every case is simply the familiar proof by contradiction, so we omit the individual proofs:

```
case (Pre f1 f2)
(14 lines omitted)
next
case (Imp f1 f2)
(14 lines omitted)
next
case (Dis f1 f2)
(14 lines omitted)
next
case (Con f1 f2)
(14 lines omitted)
next
case (Exi f)
(14 lines omitted)
next
case (Uni f)
(14 lines omitted)
```

The final case is a negated formula, for which we perform a further induction on the negated subformula and use the induction hypothesis from the outer induction to prove the cases:

```
next
case (Neg f)
then show ?case by (induct f) simp-all
qed
```

If the list of terms to instantiate formulas with is not empty, the prover will apply a `GAMMAEXI` rule or a `GAMMAUNI` rule depending on the first formula in the sequent. Due to the construction of the algorithm, formulas that do not fit these proof rules will never actually occur in this phase, but to simplify the proof, the effect function has been designed to

simply apply the NEXT rule in these cases. We thus proceed by case analysis on the first formula in the sequent:

```

next
  case (Cons t' ts')
  assume ts: (ts = t' # ts')
  then show ?thesis
  proof (cases p)

```

The proof of most cases is the usual proof by contradiction, so we omit the proofs here:

```

  case p: (Pre pn pts)
  (14 lines omitted)
  next
  case p: (Imp f1 f2)
  (14 lines omitted)
  next
  case p: (Dis f1 f2)
  (14 lines omitted)
  next
  case p: (Con f1 f2)
  (14 lines omitted)
  next
  case p: (Exi f1)
  (14 lines omitted)
  next
  case p: (Uni f1)
  (14 lines omitted)

```

In the case of a negated formula we perform further case analysis on the negated subformula:

```

next
  case p: (Neg q)
  then show ?thesis
  proof (cases q)

```

The proofs of each case are now the usual proof by contradiction, which we again omit:

```

  case q: (Pre pn pts)
  (15 lines omitted)
  next
  case q: (Imp f1 f2)
  (15 lines omitted)
  next
  case q: (Dis f1 f2)
  (15 lines omitted)
  next
  case q: (Con f1 f2)
  (15 lines omitted)
  next
  case q: (Exi f1)
  (15 lines omitted)
  next
  case q: (Uni f1)
  (15 lines omitted)

```

```

      next
      case q: (Neg f1)
(14 lines omitted)
    qed
  qed
  qed
  qed
  qed
  qed
  qed

```

This concludes the Gamma Instantiation case and thus the proof of the lemma.

Using the lemma, we can now prove the persistency property:

```

interpretation PersistentRuleSystem eff rules UNIV
unfolding PersistentRuleSystem-def RuleSystem-def PersistentRuleSystem-axioms-def

```

We first move the quantifiers and assumptions into the meta-logic:

```

proof (safe)
  fix sequent phase

```

We first need to show some rule is always enabled, which is trivial because we have already proven the property of enabledness:

```

  show  $\langle \exists r \in R. \exists sl. \text{eff } r \text{ (sequent, phase) } sl \rangle$ 
  using enabled-R rules-def by fastforce

```

Next, we need to actually prove the property of persistency. Again, we first move the quantifiers and assumptions into the meta-logic:

```

  fix r
  assume r-rule:  $\langle r \in R \rangle$ 
  show  $\langle \text{per } r \rangle$  unfolding per-def
  proof (safe)
    fix sequent phase r' sl' sequent' phase'
    assume st:  $\langle (sequent, phase) \in (UNIV :: \text{state set}) \rangle$ 
    assume r-enabled:  $\langle \text{enabled } r \text{ (sequent, phase)} \rangle$ 
    assume r':  $\langle r' \in R \rangle$ 
    assume r-not-enabled:  $\langle \neg \text{enabled } r \text{ (sequent', phase')} \rangle$ 
    assume r'-real:  $\langle r' \notin \{\} \rangle$ 
    assume r'-enabled:  $\langle \text{eff } r' \text{ (sequent, phase) } sl' \rangle$ 
    assume st'-follows:  $\langle (sequent', phase') \mid \in \mid sl' \rangle$ 
    show  $\langle r' = r \rangle$ 

```

The proof is by contradiction, so we first assume that there are two different rules  $r'$  and  $r$ , which are both enabled:

```

  proof (rule ccontr)
    assume noteq:  $\langle r' \neq r \rangle$ 
    from r'-enabled have  $\langle \text{enabled } r' \text{ (sequent, phase)} \rangle$ 
    unfolding enabled-def by fastforce

```

Next, we use our lemma to show that any rule other than  $r$  cannot be enabled:

```

  moreover have  $\langle \forall r' \in R - \{r\}. \neg \text{enabled } r' \text{ (sequent, phase)} \rangle$ 
  using r-enabled enabled-unique by simp

```

This leads to a contradiction:

```

ultimately show False using noteq r'
  by simp
qed
qed

```

Finally, we need to show that the set of possible proof states is closed under the effect relation. This is trivial because our set of possible proof states is the entire universe:

```

fix sl sequent' phase'
assume  $\langle (sequent, phase) \in (UNIV :: state\ set) \rangle$ 
assume  $\langle eff\ r\ (sequent, phase)\ sl \rangle$ 
assume  $\langle (sequent', phase') \in | sl \rangle$ 
show  $\langle (sequent', phase') \in UNIV \rangle$ 
  by simp
qed

```

This concludes the proof of the property of persistency. We have now proven that the prover can always apply some rule, and that rules will stay enabled until they are applied once they become enabled. This allows us to use the abstract completeness framework to prove our main theorem of completeness.

### 5.3 Putting it all together

The abstract completeness framework will help us prove that there is either a finite well-formed proof tree or a proof tree with an escape path for every SeCaV formula.

But the abstract completeness framework does not actually tell us anything about how to obtain this finite tree if it exists. Additionally, there is still no connection to the actual SeCaV proof system as defined in the SeCaV theory.

What we really want to prove is that if a formula is provable in the SeCaV system, then running the prover on this formula will result in a finite, well-formed tree, the root of which is labelled by the formula. Formally, for the tree  $t \equiv \text{secavProver } [p]$ , we want to prove the following as our main completeness theorem:

$$\Vdash [p] \implies \text{fst}(\text{fst}(\text{root } t)) = [p] \wedge \text{wf } t \wedge \text{tfinite } t.$$

The proof of this theorem will be based on the completeness result of the abstract completeness framework, which says that there is always either a finite and well-formed proof tree with  $[p]$  as its root, or a saturated proof tree containing an escape path containing  $[p]$ . We will use this result with a soundness result from the SeCaV theory to obtain the main result.

First we need to specialize the main theorem of the abstract completeness framework to our prover, which is easily done using the same proof idea as in the framework:

```

theorem epath-prover-completeness:
  assumes  $p \in (UNIV :: fm\ set)$ 
  defines  $t \equiv secavProver\ [p]$ 
  shows
     $(fst\ (fst\ (root\ t)) = [p] \wedge wf\ t \wedge tfinite\ t) \vee$ 
     $(\exists\ steps.\ fst\ (fst\ (shd\ steps)) = [p] \wedge epath\ steps \wedge Saturated\ steps)$  (is ?A  $\vee$  ?B)
proof –
  { assume  $\neg\ ?A$ 
    with assms have  $\neg\ tfinite\ (mkTree\ fenum\ ([p],\ PBasic))$ 
    unfolding secavProver-def using wf-mkTree fair-fenum by simp
    then obtain steps where  $ipath\ (mkTree\ fenum\ ([p],\ PBasic))\ steps$  using Konig by blast
    with assms have  $fst\ (fst\ (shd\ steps)) = [p] \wedge epath\ steps \wedge Saturated\ steps$ 
    by  $(metis\ UNIV-I\ fair-fenum\ ipath.cases\ ipath-mkTree-Saturated\ mkTree.simps(1)\ prod.sel(1)\ wf-ipath-epath\ wf-mkTree)$ 
    hence  $?B$  by blast
  }
  thus  $?thesis$  by blast
qed

```

Next, we would like to show that there is a “Herbrand-function”, which extracts a counter-model from a saturated escape path. Using such a function, we can obtain the following lemma:

```

lemma epath-countermodel:
  assumes  $(\exists\ steps.\ fst\ (fst\ (shd\ steps)) = [p] \wedge epath\ steps \wedge Saturated\ steps)$ 
  shows  $(\exists\ e\ f\ g.\ \neg\ (semantics\ e\ f\ g\ p))$ 
  sorry

```

Unfortunately, proving this lemma is quite difficult, and due to time constraints this will not be done in this project. The keyword “sorry” instructs Isabelle to admit the lemma without proof. To actually prove completeness, we would need to prove the lemma, but admitting it without proof allows us to progress for now.

Next, we need to use the law of excluded middle to show that the prover will construct a finite proof tree if it does not construct a proof tree containing a saturated escape path. This is a simple consequence of the main theorem of the abstract completeness framework:

```

lemma epath-lem:
  assumes  $p \in (UNIV :: fm\ set)$   $(\nexists\ steps.\ fst\ (fst\ (shd\ steps)) = [p] \wedge epath\ steps \wedge Saturated\ steps)$ 
  defines  $t \equiv secavProver\ [p]$ 
  shows  $fst\ (fst\ (root\ t)) = [p] \wedge wf\ t \wedge tfinite\ t$ 
  using assms(2) epath-prover-completeness t-def by blast

```

Finally, we need to show that the prover will not construct an escape path if the formula it is attempting to prove is provable in the proof system of the Sequent Calculus Verifier.

This is a simple consequence of the fact that the proof system of the Sequent Calculus Verifier is sound and the existence of a Herbrand-function:

```

lemma epath-contr:
  assumes  $\langle \Vdash [p] \rangle$ 
  shows  $\langle \nexists \text{ steps. } \text{fst} (\text{shd steps}) = [p] \wedge \text{epath steps} \wedge \text{Saturated steps} \rangle$ 
proof (rule ccontr, simp)
  show  $\langle \exists \text{ steps. } \text{epath steps} \wedge \text{fst} (\text{fst} (\text{shd steps})) = [p] \wedge \text{Saturated steps} \implies \text{False} \rangle$ 
  proof -
    assume ep:  $\langle \exists \text{ steps. } \text{epath steps} \wedge \text{fst} (\text{fst} (\text{shd steps})) = [p] \wedge \text{Saturated steps} \rangle$ 
    have  $\langle \exists e f g . \neg (\text{semantics } e f g p) \rangle$ 
    using ep epath-countermodel by blast
    with assms show False using sound by fastforce
  qed
qed

```

Using these lemmas, we can now easily prove our main theorem, which states that the prover will find a proof of any formula provable in the proof system of the Sequent Calculus Verifier:

```

theorem completeness:
  assumes  $\langle \Vdash [p] \rangle$ 
  defines  $\langle t \equiv \text{secavProver } [p] \rangle$ 
  shows  $\langle \text{fst} (\text{fst} (\text{root } t)) = [p] \wedge \text{wf } t \wedge \text{tfinite } t \rangle$ 
  by (simp add: assms epath-contr epath-lem epath-prover-completeness)

```

**end**

We note again that the theorem has not actually been fully proven due to the missing proof of the lemma that a countermodel can be extracted from a saturated escape path. More discussion on this can be found in section 6.5.

## 6 Results and discussion

The project has resulted in a fully functioning 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. This will be demonstrated below.

The prover does have some limitations, mostly related to performance and length of the generated proofs. These limitations will be explained below.

The project has also resulted in an almost finished proof of completeness of the prover in the sense that the prover is able to prove any formula for which a proof is possible. There are some holes in the proof which will be explained in more detail below.

### 6.1 Example proofs

To demonstrate that the automated theorem prover works, we will first examine the proofs generated by the prover for some simple formulas.

We will start with perhaps the simplest possible example, proving that  $p \implies p$ . The proof generated by the prover is as follows in SeCaV Unshortener format:

```
Imp p p
AlphaImp
  Neg p
  p
Ext
  p
  Neg p
Basic
```

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.

Next, we will let the prover show that  $p \wedge q \implies q \wedge p$ . The proof generated by the prover is as follows:

```
Imp (Con p q) (Con q p)
AlphaImp
  Neg (Con p q)
  Con q p
AlphaCon
  Neg p
  Neg q
  Con q p
Ext
  Con q p
  Neg p
  Neg q
```



```

BetaCon
  q
  Neg p
  Neg q
+
  p
  Neg p
  Neg q
Basic
  p
  Neg p
  Neg q
Basic

```

The only place to improve this proof is the final step. Since proof rules can actually be applied to multiple branches at once in the SeCaV system, the proof could have been finished with only a single application of the Basic rule.

The next example is  $p(a) \implies \exists x.p(x)$ . The proof generated by the prover is as follows:

```

Imp (p [a]) (Exi (p [0]))

```

```

AlphaImp
  Neg (p [a])
  Exi (p [0])
Ext
  Exi (p [0])
  Neg (p [a])
  Exi (p [0])
GammaExi [a]
  p [a]
  Neg (p [a])
  Exi (p [0])
Ext
  p [a]
  Neg (p [a])
  Exi (p [0])
Basic

```

This proof can be improved quite a bit since the quantified formula in step 3 only needs to be instantiated once. Additionally, the prover will always rotate formulas after an instantiation, but this is unnecessary here. Both applications of the EXT rule are thus unnecessary. The prover does not consider ahead of time whether it is enough to instantiate a formula only once, and thus always duplicates a quantified formula before instantiating it.

We see similar issues in the prover-generated proof of the formula

$$\forall x.\forall y.p(x,y) \implies \forall x.p(x,x),$$

The prover generates the following proof of this formula:

Imp (Uni (Uni (p [1, 0]))) (Uni (p [0, 0]))

AlphaImp

Neg (Uni (Uni (p [1, 0])))  
Uni (p [0, 0])

Ext

Uni (p [0, 0])  
Neg (Uni (Uni (p [1, 0])))

DeltaUni

p [a, a]  
Neg (Uni (Uni (p [1, 0])))

Ext

Neg (Uni (Uni (p [1, 0])))  
p [a, a]  
Neg (Uni (Uni (p [1, 0])))

GammaUni[a]

Neg (Uni (p [a, 0]))  
p [a, a]  
Neg (Uni (Uni (p [1, 0])))

Ext

Neg (Uni (Uni (p [1, 0])))  
Neg (Uni (p [a, 0]))  
p [a, a]  
Neg (Uni (Uni (p [1, 0])))

GammaUni[a]

Neg (Uni (p [a, 0]))  
Neg (Uni (p [a, 0]))  
p [a, a]  
Neg (Uni (Uni (p [1, 0])))

Ext

Neg (Uni (p [a, 0]))  
p [a, a]  
Neg (Uni (Uni (p [1, 0])))  
Neg (Uni (p [a, 0]))  
Neg (Uni (p [a, 0]))

GammaUni[a]

Neg (p [a, a])  
p [a, a]  
Neg (Uni (Uni (p [1, 0])))  
Neg (Uni (p [a, 0]))  
Neg (Uni (p [a, 0]))

Ext

p [a, a]  
Neg (Uni (Uni (p [1, 0])))  
Neg (Uni (p [a, 0]))  
Neg (Uni (p [a, 0]))  
Neg (p [a, a])

Basic

The prover still has the issue of unnecessarily duplicating quantified formulas, but we now also notice that the prover instantiates formulas that need not be instantiated at all. The first three steps of the proof are the only possible way to progress, and cannot be improved. But then the prover instantiates the same formula twice, which may look weird. The reason this happens is that the prover needs to enter two separate Gamma phases to instantiate the two universal quantifiers. In the first Gamma phase, the original formula is duplicated, and it is thus instantiated again in the second Gamma phase. A human is able to determine that this is unnecessary, but this type of reasoning is not included in the algorithm of the prover.

An extreme case of this issue can be found by letting the prover prove the Grandfather problem [41]: *If every person that is not rich has a rich father, then some rich person must have a rich grandfather.* Using the predicate  $r$  to denote being rich and the function  $f$  to denote the relationship of fatherhood, the problem can be formalized as follows:

$$\forall x.(\neg r(x) \implies r(f(x))) \implies \exists x.(r(x) \wedge r(f(f(x))))$$

The prover generates a proof of this formula consisting of 234 steps, while a slightly experienced student has proven the formula in only 19 steps. Most of the proof steps generated by the prover are instantiation and rotation of instantiated formulas which are not actually necessary for the proof.

## 6.2 Performance

For all of the proofs in the test suite mentioned above, the prover generates a proof near-instantly on a laptop machine with an Intel Core i7-7500U CPU clocked at 2.70 GHz and 16 GB RAM. Since the formulas in the test suite were selected from exercises, we conclude that most formulas needed by students can easily be proven by the prover. Even the Grandfather Paradox, which students rarely manage to prove, is proven near-instantly by the prover.

Another interesting aspect of performance is the length of the generated proofs. Since the prover works by decomposing and instantiating all terms, it will very often spend large amounts of the proof decomposing formulas that are not involved in the current branch at all, while an experienced human will be able to recognize that some formulas are not needed and thus save steps in their proofs. As we saw, the prover generally produces longer proofs than a human, and in some cases significantly longer. This is especially the case with nested quantifiers and formulas containing many functions, which can induce many unnecessary proof steps due to unnecessary instantiations.

## 6.3 Tests

The prover has been tested for both soundness and completeness through two separate test suites. The test suites are implemented using the test suite system of the cabal build system. Both test suites consist of a list of formulas to attempt to prove and a mechanism for determining whether the prover has generated a correct proof or not.

The test suite for completeness works by testing that the prover produces correct proofs for every formula in the test suite. This is done by using the prover to generate an Isabelle file containing a proof of the formula, then using the Isabelle proof assistant to verify that the proofs generated by the prover are correct. If the prover fails to generate a proof, or if the proof cannot be verified to be correct by Isabelle, the test fails.

The test suite for soundness works by testing that the prover does not produce a proof within 10 seconds. Since the prover produces proofs almost instantly for small formulas such as those in the test suite, 10 seconds was determined to be enough to conclude that the prover will never produce a proof. Choosing to let the prover attempt to produce a proof for a longer period would have to be balanced by the performance requirements of allowing the prover to construct very large proof trees, which eventually require large amounts of memory.

## 6.4 Limitations in the prover

There are a number of limitations and possibilities for optimization in the prover algorithm itself. Most importantly, the focus of the algorithm is only on completeness, not optimization. This means that the prover essentially works by “brute force”.

The most obvious opportunity for optimization is controlling the order of proof rules. In systems with unordered sequents, it is generally better to apply Alpha-rules before Beta-rules to avoid duplication of work in the proof, but the prover simply applies rules based on the order of the formulas in the sequent. Since our system uses ordered sequents, re-ordering the sequent requires an application of the EXT rule. This means that changing the rule order to apply Alpha-rules before Beta-rules may not always be worth it, since this may introduce additional applications of the EXT rule. The introduction of this optimization will thus need some heuristics to determine whether the optimization is worth it in the specific case.

Another opportunity for optimization is to only instantiate Gamma-formulas with some terms instead of every term occurring in the sequent. Unfortunately, this optimization does interfere with the proof of completeness, since it is generally not obvious how to determine which terms are actually needed to prove a formula. It might be the case, however, that some specific cases where instantiations are not needed could be considered in the algorithm. This could potentially save quite a lot of steps in proofs, especially if the algorithm could skip entire Gamma-phases.

The final obvious opportunity for optimization is to decompose only those formulas that are actually needed to prove a formula. In the formula  $((p \vee p) \wedge (q \vee q)) \implies p$ , for example, there is no reason to decompose the formula  $q \vee q$ , since the predicates in the formula do not appear anywhere else in the formula. Experienced humans are typically reasonably good at determining when a subformula is unnecessary, but the prover does not do so. In some cases this can save many proof steps, since entire parts of a sequent can be ignored. It might be possible to determine some specific cases for which an algorithm can determine that a formula is not needed without losing completeness.

All of these optimizations will of course improve performance, since they mean that fewer proof steps have to be considered (assuming that the algorithms that determine if a step is unnecessary are not too complex). Unfortunately, they will most likely require much more work within the completeness proof to implement, and may also complicate the algorithm, making it harder to understand. If we do not care about performance, but only the length and readability of the generated proofs, a simpler approach could be to post-process the proofs found by the current algorithm to apply the optimizations. With an already existing proof in hand, it should be quite simple to determine which instantiations and formulas turned out to be necessary for the proof, and then work backward to remove the superfluous elements of the proof.

## 6.5 Missing proofs

While large parts of a completeness proof for the prover have been completed, there is still a hole in the proof. Additionally, a proof of soundness for proof trees has not been formalized. Finally, the proof trees generated by the prover require some unverified post-processing to obtain actual proofs in the SeCaV system. All of these issues can most likely be resolved given more time. A brief sketch of a soundness proof has already been given in chapter 4. In this section we will discuss how to prove the missing lemma in the completeness theorem and verify the post-processing functions.

### 6.5.1 The missing completeness lemma

The completeness proof is missing a key lemma which says that it is possible to extract an actual countermodel from a saturated escape path in a proof tree.

Following the approach in [5], an approach to proving this lemma is to define a “Herbrand” function which takes a saturated escape path starting in state  $(s, p)$  and turns it into a countermodel for  $s$ . We then need to prove that the structure returned by the function is actually a countermodel.

The approach in [5] is to prove that the union of all sequents in the escape path forms a Hintikka set, which means that it is “well-behaved” with respect to connectives and quantifiers in that the set is downwards saturated. This follows from the saturation of the escape path with respect to the proof rules.

To show that the structure returned by the Herbrand function is a countermodel, we need to show that there is some evaluation  $e$  which does not satisfy the initial sequent  $s$ . This can then be done by exploiting the satisfiability property of Hintikka sets and the construction of escape paths, though it is not clear exactly how this is done.

### 6.5.2 Verified post-processing

Since the prover produces a proof tree and not a linear SeCaV proof, the Haskell part of the implementation post-processes the generated proof trees to obtain a linear textual representation of the proof in SeCaV Unshortener syntax. We would of course like to prove that this post-processing does not invalidate the proofs generated by the prover.

Since the post-processing occurs outside of the part of the prover implemented in the Isabelle proof assistant, we cannot prove theorems about it. The first step in performing such a proof would thus be to move the post-processing into the Isabelle part of the implementation. Unfortunately this complicates the implementation since the finite set type used to contain generated branches is not an inductive type in Isabelle, but is in Haskell. This means that functions cannot be defined inductively on this type in Isabelle, which makes the implementation of the post-processing functions much more involved.

The first step would be to convert the coinductive proof tree into an inductive proof tree on which inductive functions can then be defined. Each of the post-processing functions (removal of NEXT rules, substitution of Gamma rules, and collapsing of EXT rules) can then be defined almost as in Haskell, except that pattern matching cannot be used for finite sets.

Once the functions have been implemented, we are interested in proving that they preserve soundness and completeness. If this can be proven, the existing theorems can then be converted into theorems about post-processed proofs.

### 6.5.3 Trustworthiness of the prover

While the soundness and completeness of the prover have not been verified, the prover is still quite trustworthy. The properties of enabledness and persistency have been proven, meaning that the prover will never get stuck while attempting to prove a formula. Additionally, the prover has been tested for soundness and completeness for a number of formulas. Finally, the algorithm underlying the prover is based on an algorithm that is known to be sound and complete [19].

## 6.6 Sources of complexity

The project was more difficult than originally anticipated, and much time was spent attempting different ways to implement an algorithm before arriving at the algorithm described above. Much time was initially spent on an approach based on generating all possible terms, formulas, sequents, and finally rule applications from an isomorphism with the natural numbers, and then constructing a completeness proof from there. This was inspired by an existing application of the abstract completeness framework to a system of propositional logic, but was not viable for the system of the Sequent Calculus Verifier because it was not clear how to ensure that the prover did not apply endless amounts of instantiations while still guaranteeing that all relevant terms were used in instantiations. This problem was largely due to the fact that the abstract completeness framework expects a fixed stream of proof rules to attempt to apply, which meant that the proof rules for each phase of the algorithm had to be interleaved in a fashion that guaranteed that an appropriate amount of other rules were applied before each application of a Gamma-rule.

Later, the decision was made to introduce the NEXT rule and the proof phases as part of the proof state to explicitly control the transitions between phases. This introduced a proof rule that does nothing in the actual proof, but made the task of creating a fair stream of proof rules much easier, since the rules could now simply occur in any order as long as every proof rule would eventually be attempted. Unfortunately this decision came quite late, which decelerated the project significantly.

If the abstract completeness framework could be modified to allow the stream of proof rules to attempt to apply to depend on the proof state in a more direct manner, the effect function could be simplified significantly by removing the need for explicit control of the proof phases. This would likely also simplify the proofs due to the simplification of the effect function.

Another source of complexity is that post-processing is needed to obtain an actual SeCaV proof from the proof tree generated by the prover. One of the more complicated parts of this post-processing is the substitution of Gamma-rules with implicit terms with Gamma-rules that explicitly specify what terms they are instantiating the quantifier with. If the Sequent Calculus Verifier could be modified such that explicitly specifying the instantiating term was never necessary, this post-processing would not be necessary. Unfortunately it is not currently clear whether this is possible.

## 7 Conclusion

The main goal of the project was to design an automated theorem prover for the proof system of the Sequent Calculus Verifier. The design of the prover and the generated proofs were to be “natural” in the sense that they are easy to understand for human readers, including students. We argue that both the algorithm and the implementation are quite easy to understand since the phases mirror the way a human would approach proving a formula. As discussed, the proofs generated by the prover do sometimes include some “unnatural” steps since the prover cannot detect useless formulas and instantiations like an experienced human would. The algorithm could be enriched to do so and thus produce more natural proofs, but this might come at the expense of an overly complicated and hard to understand algorithm.

Another goal was to formally verify the soundness and completeness of the automated theorem prover. This goal was not met, but significant progress towards a proof of completeness was made, and testing indicates that the implementation of the prover is both sound and complete. We are confident that the missing proofs could be produced following the suggested proof sketches if given more time.

The automated theorem prover is fully functional and can be used to find proofs of complicated formulas in its current state. The proofs generated by the prover can be studied to obtain ideas on how to prove a formula, and further optimization of the proofs by eliminating useless decompositions and instantiations could be a good exercise for students. Additionally, students can easily inspect and experiment with the algorithm to learn how to formalize their approach to proving formulas in the SeCaV system. More advanced students could also attempt to modify the algorithm and rework the proofs made in this project to fit their alterations.

In conclusion, the goal of designing an easily understandable automated theorem prover was met, while the goal of formalizing the prover was only partly completed. While the work presented here is incomplete, both the prover and the associated proofs are already useful as a learning tool for students to experiment with, and the prover is of course also able to prove actual formulas in the SeCaV system.

# Bibliography

- [1] Asta Halkjær From et al. “Teaching a Formalized Logical Calculus”. In: *Electronic Proceedings in Theoretical Computer Science* 313 (2020). Ed. by Pedro Quaresma, Walther Neuper, and Joao Marcos, pp. 73–92. ISSN: 20752180. DOI: 10.4204/EPTCS.313.5. URL: <https://doi.org/10.4204/EPTCS.313.5>.
- [2] Asta Halkjær From, Jørgen Villadsen, and Patrick Blackburn. “Isabelle/HOL as a Meta-Language for Teaching Logic”. In: *Electronic Proceedings in Theoretical Computer Science* 328 (Oct. 2020), pp. 18–34. ISSN: 2075-2180. DOI: 10.4204/eptcs.328.2. URL: <http://dx.doi.org/10.4204/EPTCS.328.2>.
- [3] Asta Halkjær From, Frederik Krogsdal Jacobsen, and Jørgen Villadsen. “SeCaV: A Sequent Calculus Verifier in Isabelle/HOL”. In: *International Workshop on Logical and Semantic Frameworks with Applications* (2021). (Accepted).
- [4] Jasmin Christian Blanchette, Andrei Popescu, and Dmitriy Traytel. “Unified Classical Logic Completeness”. In: *Automated Reasoning*. Ed. by Stéphane Demri, Deepak Kapur, and Christoph Weidenbach. Cham: Springer International Publishing, 2014, pp. 46–60. ISBN: 978-3-319-08587-6.
- [5] Jasmin Christian Blanchette, Andrei Popescu, and Dmitriy Traytel. “Soundness and Completeness Proofs by Coinductive Methods”. In: *Journal of Automated Reasoning* 58 (2017), pp. 149–179. DOI: 10.1007/s10817-016-9391-3. URL: <https://doi.org/10.1007/s10817-016-9391-3>.
- [6] Jasmin Christian Blanchette and Andrei Popescu. “Mechanizing the Metatheory of Sledgehammer”. In: *Frontiers of Combining Systems*. Ed. by Pascal Fontaine, Christophe Ringeissen, and Renate A. Schmidt. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 245–260. ISBN: 978-3-642-40885-4.
- [7] Jørgen Villadsen, Andreas Halkjær From, and Anders Schlichtkrull. “Natural Deduction Assistant (NaDeA)”. In: *Electronic Proceedings in Theoretical Computer Science* 290.290 (2019), pp. 14–29. ISSN: 20752180. DOI: 10.4204/EPTCS.290.2.
- [8] David Cerna et al. “Aiding an Introduction to Formal Reasoning Within a First-Year Logic Course for CS Majors Using a Mobile Self-Study App”. In: *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*. ITiCSE ’20. Trondheim, Norway: Association for Computing Machinery, 2020, pp. 61–67. ISBN: 9781450368742. DOI: 10.1145/3341525.3387409. URL: <https://doi-org.proxy.findit.dtu.dk/10.1145/3341525.3387409>.
- [9] J. Breitner. “Visual Theorem Proving with the Incredible Proof Machine”. In: *Interactive Theorem Proving*. Ed. by J. Blanchette and S. Merz. Vol. ITP 2016. Springer, Cham, 2016. DOI: 10.1007/978-3-319-43144-4\_8.
- [10] Arno Ehle, Norbert Hundeshagen, and Martin Lange. “The Sequent Calculus Trainer with Automated Reasoning - Helping Students to Find Proofs”. In: *Electronic Proceedings in Theoretical Computer Science* 267 (Mar. 2018), pp. 19–37. ISSN: 2075-2180. DOI: 10.4204/eptcs.267.2. URL: <http://dx.doi.org/10.4204/EPTCS.267.2>.
- [11] Arno Ehle. “Proof Search in the Sequent Calculus for First-Order Logic with Equality”. MA thesis. Universität Kassel, Feb. 2017.
- [12] Francis Jeffrey Pelletier. “Automated Natural Deduction in THINKER”. In: *Studia Logica* 60.1 (1998), pp. 3–43. ISSN: 15728730, 00393215. DOI: 10.1023/A:1005035316026. URL: <https://doi.org/10.1023/A:1005035316026>.



- [13] Dominique Pastre. “MUSCADET 2.3: A knowledge-based theorem prover based on natural deduction”. In: *Lecture Notes in Computer Science (including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 2083 (2001). Ed. by R. Gore, A. Leitsch, and T. Nipkow, pp. 685–689. issn: 16113349, 03029743. doi: 10.1007/3-540-45744-5\_56.
- [14] G. Sutcliffe. “The TPTP Problem Library and Associated Infrastructure. From CNF to TH0, TPTP v6.4.0”. In: *Journal of Automated Reasoning* 59.4 (2017), pp. 483–502.
- [15] Anders Schlichtkrull, Jasmin Christian Blanchette, and Dmitriy Traytel. “A Verified Prover Based on Ordered Resolution”. In: *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2019. Cascais, Portugal: Association for Computing Machinery, 2019, pp. 152–165. isbn: 9781450362221. doi: 10.1145/3293880.3294100. url: <https://doi.org/10.1145/3293880.3294100>.
- [16] Alexander Birch Jensen et al. “Programming and Verifying a Declarative First-Order Prover in Isabelle/HOL”. In: *Ai Communications* 31.3 (2018), pp. 281–299. issn: 18758452, 09217126. doi: 10.3233/AIC-180764.
- [17] Jørgen Villadsen, Anders Schlichtkrull, and Andreas Halkjær From. “A Verified Simple Prover for First-Order Logic”. In: *CEUR Workshop Proceedings* 2162 (2018). Ed. by Boris Konev, Josef Urban, and Philipp Rümmer, pp. 88–104. issn: 16130073.
- [18] Tom Ridge and James Margetson. “A mechanically verified, sound and complete theorem prover for first order logic”. In: *Lecture Notes in Computer Science* 3603 (2005). Ed. by J. Hurd et al., pp. 294–309. issn: 03029743, 16113349. doi: 10.1007/11541868\_19.
- [19] Mordechai Ben-Ari. *Mathematical Logic for Computer Science*. Springer London, 2012, pp. 149–150. isbn: 978-1-4471-4128-0.
- [20] Gerhard Gentzen. “Untersuchungen über das logische Schließen, I and II”. (German: Studies on logical inference). In: *Mathematische Zeitschrift* 39.1 (1935), pp. 176–210. issn: 14321823, 00255874. doi: 10.1007/BF01201353.
- [21] Evert Willem Beth. *Semantic Entailment and Formal Derivability*. Noord-Hollandsche, 1955.
- [22] J. A. Robinson. “A Machine-Oriented Logic Based on the Resolution Principle”. In: *J. ACM* 12.1 (Jan. 1965), pp. 23–41. issn: 0004-5411. doi: 10.1145/321250.321253. url: <https://doi-org.proxy.findit.dtu.dk/10.1145/321250.321253>.
- [23] S.C. Kleene. *Introduction to metamathematics*. Vol. 1. North-Holland, 1952.
- [24] Kaarlo Jaakko Hintikka. “A new approach to sentential logic”. In: *Commentationes Physico-Mathematicae* 17 (1953), pp. 1–14.
- [25] K. Gödel. “Die Vollständigkeit der Axiome des logischen Funktionenkalküls”. (German: The completeness of the axioms of the logical function calculus). In: *Monatsh. Math. Phys.* 37 (1930), pp. 349–360. issn: 0026-9255; 1436-5081/e.
- [26] John O’Leary. “Formal Verification in Intel CPU Design”. In: *Proceedings of the Second ACM/IEEE International Conference on Formal Methods and Models for Co-Design*. MEMOCODE ’04. USA: IEEE Computer Society, 2004, p. 152. isbn: 0780385098. doi: 10.1109/MEMCOD.2004.1459841. url: <https://doi-org.proxy.findit.dtu.dk/10.1109/MEMCOD.2004.1459841>.
- [27] Laura Kovács and Andrei Voronkov. “First-Order Theorem Proving and Vampire”. In: *Lecture Notes in Computer Science*. CAV 2013 8044 (2013). doi: 10.1007/978-3-642-39799-8\_1.
- [28] Stephan Schulz, Simon Cruanes, and Petar Vukmirović. “Faster, Higher, Stronger: E 2.3”. In: *Proc. of the 27th CADE, Natal, Brasil*. Ed. by Pascal Fontaine. LNAI 11716. Springer, 2019, pp. 495–507.

- [29] Konstantin Korovin. “iProver - An Instantiation-Based Theorem Prover for First-Order Logic (System Description)”. In: *Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia, August 12-15, 2008, Proceedings*. Ed. by Alessandro Armando, Peter Baumgartner, and Gilles Dowek. Vol. 5195. Lecture Notes in Computer Science. Springer, 2008, pp. 292–298. DOI: 10.1007/978-3-540-71070-7\\_24. URL: [https://doi.org/10.1007/978-3-540-71070-7%5C\\_24](https://doi.org/10.1007/978-3-540-71070-7%5C_24).
- [30] W. McCune. *Prover9 and Mace4*. 2010. URL: <http://www.cs.unm.edu/~mccune/prover9/>.
- [31] B. Jacobs and J. Rutten. “A tutorial on (co)algebras and (co)induction”. In: *Bulletin of the European Association for Theoretical Computer Science* 62 (1997), pp. 222–59. ISSN: 02529742.
- [32] Simon Marlow (editor). *Haskell 2010 Language Report*. 2010. URL: <https://www.haskell.org/definition/haskell2010.pdf>.
- [33] Larry Paulson and Tobias Nipkow. *Isabelle/HOL*. 2021. URL: <https://isabelle.in.tum.de/>.
- [34] Vladimir Voevodsky. “The Origins and Motivations of Univalent Foundations”. In: *The Institute Letter Summer 2014* (2014).
- [35] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq’Art: the calculus of inductive constructions*. Springer, 2004. ISBN: 9783540208549.
- [36] Leonardo de Moura et al. “The Lean Theorem Prover (system description)”. In: *Lecture Notes in Computer Science (including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 9195 (2015). Ed. by Amy P. Felty and Aart Middeldorp, pp. 378–388. ISSN: 16113349, 03029743. DOI: 10.1007/978-3-319-21401-6\_26.
- [37] Grzegorz Bancerek et al. “Mizar: State-of-the-art and Beyond”. In: *Intelligent Computer Mathematics, CICM 2015*. Ed. by M. Kerber et al. Vol. 9150. Lecture Notes in Computer Science. Springer, 2015, pp. 261–279. DOI: 10.1007/978-3-319-20615-8\_17. URL: [https://doi.org/10.1007/978-3-319-20615-8\\_17](https://doi.org/10.1007/978-3-319-20615-8_17).
- [38] SRI International. *Prototype Verification System (PVS)*. 333 Ravenswood Avenue, Menlo Park, CA 94025-3493, USA, 2021. URL: <https://pvs.csl.sri.com/>.
- [39] William A. Howard. “The formulae-as-types notion of construction”. In: *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*. Ed. by Jonathan P. Seldin and J. Roger Hindley. Academic Press, Sept. 1980, pp. 479–490. ISBN: 978-0-12-349050-6.
- [40] Raymond M. Smullyan. *First-order logic*. Dover Publications, 1995.
- [41] Marcello D’Agostino et al. *Handbook of tableau methods*. Ed. by Marcello D’Agostino. Kluwer, 1999, p. 128. ISBN: 9780792356271.



We describe the design and implementation of an automated theorem prover for the proof system of the Sequent Calculus Verifier. The prover is designed to generate “natural” proofs in a one-sided sequent calculus, and the algorithm underlying the prover is a formalization of an intuitive approach to proving formulas. The automated theorem prover is implemented in Isabelle/HOL and Haskell, and we additionally present an unfinished proof of completeness of the prover and a sketch of a proof of soundness in Isabelle/HOL. Both the prover and the proofs about it are useful as a learning tool for students to experiment with and see how properties about programs can be proven. While formalized proofs of soundness and completeness are unfinished, confidence that the prover works correctly has been obtained through automated tests.

Technical  
University of  
Denmark

Richard Petersens Plads  
Building 324  
2800 Kgs. Lyngby  
Tel. 4525 3031

[www.compute.dtu.dk](http://www.compute.dtu.dk)