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

Asta Halkjær From
DTU Compute, Technical University of Denmark, Kongens Lyngby, Denmark

Frederik Krogsdal Jacobsen
DTU Compute, Technical University of Denmark, Kongens Lyngby, Denmark

Abstract

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

2012 ACM Subject Classification Theory of computation → Automated reasoning; Theory of computation → Proof theory; Theory of computation → Program verification

Keywords and phrases Isabelle/HOL, SeCaV, First-Order Logic, Prover, Soundness, Completeness

Digital Object Identifier 10.4230/LIPIcs.ITP.2022.13

Supplementary Material *Software (Isabelle/HOL formalization in the Archive of Formal Proofs)*: `https://www.isa-afp.org/entries/FOL_Seq_Calc2.html` [18]

Acknowledgements We would like to thank Agnes Moesgård Eschen, Alexander Birch Jensen, Anders Schlichtkrull, Simon Tobias Lund and Jørgen Villadsen for comments on drafts. We are very grateful to the anonymous reviewers for their thoughtful comments.

1 Introduction

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

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

In this paper, we present an automated theorem prover for first-order logic with functions based on sequent calculus. We formalize its soundness and completeness in Isabelle/HOL. We reuse the syntax and semantics of first-order logic from the Sequent Calculus Verifier (SeCaV) system [16] (Section 2.1). We state the soundness and completeness of the prover with respect to the SeCaV proof system, its semantics and a bounded semantics that we introduce here. The prover can generate human-readable and machine-verifiable SeCaV proofs for valid formulas.

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

Our prover is deterministic, fair and works on finite sequents. To handle the quantifiers we must thus build our countermodel in a Herbrand universe that contains only the subset of terms that actually appear in the failed proof. This idea is inspired by Ben-Ari's textbook proof [2], where terms are either variables or constants, and by Ridge's Isabelle proof [38], where only variables are considered. We are not aware of any previous formalization of this construction that handles functions. We consider all terms in our Herbrand universe, including those with free variables, yielding completeness for both open and closed formulas.

The prover is free software and the source code is available as supplementary material. This consists of around 3000 lines of Isabelle/HOL and 1300 lines of supplementary Haskell.

We summarize our main contributions:

- A formally verified sound and complete automated theorem prover for full first-order logic with functions.
- An analytic proof of completeness for both open and closed formulas for a deterministic prover via a bounded semantics.
- A method of translating the prover-generated certificates of validity into human-readable and machine-verifiable proofs in SeCaV.
- A concrete application of the abstract completeness framework, and a demonstration of how to obtain soundness and completeness of an actual, executable prover using the framework as a starting point.

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

1.1 Related work

The present paper is a much improved version of the work started in the second author's master's thesis [21]. The Sequent Calculus Verifier (SeCaV) is a well-established proof system, and both soundness and completeness have been proven for the system [19]. The system has been used to teach students in several courses at the Technical University of Denmark [20,47]. An online tool called the SeCaV Unshortener has been developed to allow input of proofs in a simple format which is then translated to an Isabelle proof [16].

Our prover is based on the abstract completeness framework by Blanchette et al. [10,11]. The framework contains a simple example prover for propositional logic, and the original application of the framework was in the formalization of the metatheory of the Sledgehammer tool for automated theorem proving within Isabelle/HOL [8]. Blanchette et al. [11] have used the framework to formalize soundness and completeness of a calculus for first-order logic with equality and in negation normal form. Their search is nondeterministic and they do not generate an executable prover like we do. As such, we improve on their work by using the framework to prove soundness and completeness of an executable prover.

A number of other systems have formally verified metatheories. NaDeA (Natural Deduction Assistant) by Villadsen et al. [49] is a web application that allows users to prove formulas with natural deduction. The metatheory of a model of the system is formalized in Isabelle/HOL, and the application allows export of proofs for verification in Isabelle. The Incredible Proof Machine by Breitner [12] is a web application that allows users to create proofs using a specialized graphical interface. The proof system is as strong as natural deduction, and a model of the system is formalized in Isabelle using the abstract framework by Blanchette et al. [11]. Neither system includes automated theorem provers; they are essentially simple proof assistants designed to aid students in understanding logical systems.

THINKER by Pelletier [35] is a proof system and an attached automated theorem prover. THINKER is a natural deduction system designed to allow for what the author calls "direct proofs", as opposed to proofs based on reduction to a resolution system. THINKER was perhaps the first automated theorem prover designed specifically with "naturality" in mind, as a reaction to the indirectness of resolution-based proof systems. MUSCADET by Pastre [34] is also an automated theorem prover based on natural deduction. The system distinguishes itself by also supporting usage of prior knowledge such as previously proven theorems through a Prolog knowledge base.

While there are many very advanced automated theorem provers such as Vampire [24], Zipperposition [3] and Z3 [13], their metatheory and implementations are rarely formalized. As a first step towards formally verifying modern provers, Schlichtkrull et al. [40] have formalized an ordered resolution prover for *clausal* first-order logic in Isabelle/HOL. Jensen et al. [22] formalized the soundness, but not the completeness, of a prover for first-order logic with equality in Isabelle/HOL. Villadsen et al. [50] verified a simple prover for first-order logic in Isabelle/HOL with the aim of allowing students to understand both the prover and the formalization. That work is based on an earlier formalization by Ridge and Margetson [38], but simplifies both the prover and the proofs to enable easier understanding by students. Neither of these two provers provide support for functions or generation of proof certificates.

Blanchette [5] gives an overview of a number of verification efforts including the metatheory of SAT solvers [6, 14, 29, 30, 43] and certificate checkers [26, 27], SMT solvers [28, 31, 45], the superposition calculus [37], resolution [36, 39, 41], a number of non-classical logics [15, 17, 42, 46, 48], and a wide range of proof systems for classical propositional logic [32, 33]. Some of these efforts are part of the IsaFoL project (Isabelle Formalization of Logic). Part of the goal is to develop "a methodology for formalizing modern research in automated reasoning". Our work points in this direction too, by formally verifying a non-saturation-based prover.

2 Background

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

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

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

Figure 1 The syntax of the Sequent Calculus Verifier (parentheses added for clarity).

```
definition shift e v x ≡ λn. if n < v then e n else if n = v then x else e (n − 1)

primrec semantics-term and semantics-list where
  semantics-term e f (Var n) = e n
| semantics-term e f (Fun i l) = f i (semantics-list e f l)
| semantics-list e f [] = []
| semantics-list e f (t # l) = semantics-term e f t # semantics-list e f l

primrec semantics where
  semantics e f g (Pre i l) = g i (semantics-list e f l)
| semantics e f g (Imp p q) = (semantics e f g p ⟶ semantics e f g q)
| semantics e f g (Dis p q) = (semantics e f g p ∨ semantics e f g q)
| semantics e f g (Con p q) = (semantics e f g p ∧ semantics e f g q)
| semantics e f g (Exi p) = (∃x. semantics (shift e 0 x) f g p)
| semantics e f g (Uni p) = (∀x. semantics (shift e 0 x) f g p)
| semantics e f g (Neg p) = (¬ semantics e f g p)
```

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

2.1 The Sequent Calculus Verifier

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

The semantics of a formula is due to Berghofer [4], who models the universe as a type variable like we do for now. The interpretation consists of an environment e for variables, a function denotation f and a predicate denotation g. The semantics of the system is standard and defined using the three recursive functions in Figure 2. The semantics of the logical connectives is defined using the connectives from the meta-logic in Isabelle/HOL. The *shift* function handles shifting de Bruijn-indices when interpreting quantifiers. We say that a sequent is valid when, under all interpretations, some formula in the sequent is satisfied.

The system has a number of proof rules, some of which are displayed in Figure 3 (abusing set notation for the membership and inclusion relations on lists – see the formalization for details and the remaining rules). The rules should be read from the bottom up, since we generally work backwards from a sequent we wish to prove. The rules are classified according to Smullyan's uniform notation [44].

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

$$\frac{\text{Neg } p \in z}{\Vdash p, z}\ \textsc{Basic} \qquad \frac{\Vdash z \qquad z \subseteq y}{\Vdash y}\ \textsc{Ext} \qquad \frac{\Vdash p, z}{\Vdash \text{Neg (Neg } p), z}\ \textsc{NegNeg}$$

$$\frac{\Vdash p, q, z}{\Vdash \text{Dis } p\ q, z}\ \textsc{AlphaDis} \qquad \frac{\Vdash \text{Neg } p, z \qquad \Vdash \text{Neg } q, z}{\Vdash \text{Neg (Dis } p\ q), z}\ \textsc{BetaDis}$$

$$\frac{\Vdash p\,[\text{Var } 0/t], z}{\Vdash \text{Exi } p, z}\ \textsc{GammaExi} \qquad \frac{\Vdash \text{Neg } (p\,[\text{Var } 0/\text{Fun } i\ []]), z \qquad i \text{ fresh}}{\Vdash \text{Neg (Exi } p), z}\ \textsc{DeltaExi}$$

Figure 3 Sample proof rules for the Sequent Calculus Verifier (rules omitted here are similar).

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

The NegNeg rule removes a double negation from the first formula in a sequent. It can be considered an α-rule, but we keep it separate from the others because it does not generate two formulas. The AlphaDis rule decomposes disjunctions (and similar for the AlphaImp and AlphaCon rules omitted here). The BetaDis rule decomposes negated disjunctions and requires that two sequents are proven separately, creating branches in the proof tree (and similar for the BetaCon, BetaImp rules omitted here). This essentially moves the connective into the proof tree itself, since both branches now need to be proven separately. The GammaExi rule instantiates an existential quantifier with any term t by substituting t for variable 0 in the quantified formula. The GammaUni rule omitted here is similar. The DeltaExi rule instantiates a negated existential quantifier in the first formula in a sequent with a fresh constant function, with fresh here meaning that the function identifier does not already occur anywhere in the sequent. The fresh constant cannot have any relationship to other terms in the sequent: it is *arbitrary*. Thus we could have used any other term without affecting the validity of the formula, which is exactly what is needed to prove a universally quantified ("there does not exist") formula. The DeltaUni rule omitted here is similar.

The proof system in Figure 3 has been formally verified to be sound and complete with regards to the semantics in Figure 2 by From et al. [19]. We use these results to relate our prover to SeCaV.

2.2 Abstract frameworks for soundness and completeness

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

Locales [1, 23] allow the abstraction of definitions and proofs over given parameters. As an example, consider groups in algebra defined by a carrier set, a binary operation and the group axioms. With a locale, these can be specified abstractly and a number of operations

Table 1 The *RuleSystem* locale with premises above the line and important conclusions below.

eff	Effect relation between a rule, a state and a finite set of resulting states.
rules	Stream of rules. The set of these is called R.
S	Set of well formed states.
eff-S	Proof that for any rule in R and proof state in S the *eff*-related states are in S.
enabled-R	Proof that for any state in S, some rule in R is *enabled*, i.e. applies to that state.
mkTree	A function from a stream of rules and a starting state to a tree of states and rules.
wf-mkTree	Proof that the tree generated by *mkTree* is well formed wrt. *eff*.

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

per	Proof that if a rule r in R is enabled in a well formed state s and s' is *eff*-related to s by a rule r' in R distinct from r, then r is enabled in s'.
epath-completeness-Saturated	Proof that for any well formed state s, there exists either a well formed finite tree with s as root or a saturated escape path with s as root.

and results can then be given in the abstract. Later, we can instantiate the locale with a concrete group by providing the carrier set and binary operation, and proving that the group axioms are fulfilled. We then obtain instantiations of the results for our concrete group.

In this section we give an overview of the locales provided by the abstract framework: what they require and what they provide. We have condensed the Isabelle code into four tables for brevity, since the specific details of the framework are not our main focus. The exact definitions can be found in the Archive of Formal Proofs entry by Blanchette et al. [9].

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

Tables 1 and 2 cover the two locales *RuleSystem* and *PersistentRuleSystem* which are central for proving completeness. The locale premises are given above each vertical line and the (important) conclusions are given below. The locales require us to prove a number of things about three definitions. First, the *eff* relation specifies the effect of applying a rule to a state in our proof search. By (proof) state we mean a sequent, potentially coupled with additional information. The nodes of our proof tree will be proof states in this sense. Second, *rules* is a stream of rules for the prover to attempt to apply. Third, S is a set of well formed states (in our case simply the set of all states).

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

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

Table 3 The *Soundness* locale.

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

Table 4 The *RuleSystem-Code* locale.

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

Table 3 covers the *Soundness* locale used to prove the soundness of resulting proof trees. Here, besides *eff* and *rules*, we must state a set of models, *structure*, and a satisfaction predicate, *sat*, on sequents and models. The locale then turns a local soundness proof, *local-soundness*, that validity of a sequent follows from validity of its children, into a global result, *soundness*, that any finite, well formed tree has a valid root.

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

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

3 Prover

In this section we explain the design of the proof search procedure driving our prover. The procedure does not use the proof system of SeCaV directly, but introduces a new set of similar proof rules that apply to entire sequents at once. This obviates the need for the structural Ext rule, which is therefore not present. Additionally, we remove the Basic rule and let the prover close proof branches implicitly.

Before we can define what the rules do, we need a few auxiliary definitions. The function *generateNew* generates a function name that is fresh to a given list of terms. The function *subtermFms* computes the list of terms occurring in a list of functions. We define *subterms* as the list of all terms in a sequent, except that the list contains exactly *Fun 0 []* when it would otherwise be empty. This ensures that we always have some term to instantiate γ-formulas with. The function *sub* implements substitution in a standard way using de Bruijn indices. See the formalization [18] or the original SeCaV work [19] for details. The function *branchDone* computes whether a sequent is an axiom, i.e. whether the sequent contains both a formula and its negation. The prover uses this to determine when a branch of the proof tree is proven and can be closed.

We first define which "parts" of a single formula must be proven for a rule to apply:

```
definition parts :: tm list ⇒ rule ⇒ fm ⇒ fm list list where
  parts A r f ≡ (case (r, f) of
      (NegNeg, Neg (Neg p)) ⇒ [[p]]
    | (AlphaDis, Dis p q) ⇒ [[p, q]]
    | (BetaDis, Neg (Dis p q)) ⇒ [[Neg p], [Neg q]]
    | (DeltaExi, Neg (Exi p)) ⇒ [[Neg (sub 0 (Fun (generateNew A) []) p)]]
    | (GammaExi, Exi p) ⇒ [Exi p # map (λt. sub 0 t p) A]
    ⋮
    | - ⇒ [[f]])
```

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

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

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

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

```
primrec list-prod :: ′a list list ⇒ ′a list list ⇒ ′a list list where
  list-prod - [] = []
| list-prod hs (t # ts) = map (λh. h @ t) hs @ list-prod hs ts
```

The type variable $'a$ in the type signature means that the function works on lists of lists containing any type of elements.

It behaves in the following way (similar to the Cartesian product):

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

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

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

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

```
primrec effect :: rule ⇒ state ⇒ state fset where
  effect r (A, z) =
  (if branchDone z then {||} else
    fimage (λz'. (remdups (A @ subterms z @ subterms z'), z'))
    (fset-of-list (children (remdups (A @ subtermFms z)) r z)))
```

To fit the types of the framework, the function returns a finite set (*fset*) instead of a list. If the sequent is an axiom, the branch is proven, and the function returns an empty set of child nodes, closing the branch. Otherwise, the function converts the result of the *children* function to a finite set, and adds any new terms to the list of terms in each child node.

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

3.1 Applying the framework

We are now ready to apply the abstract completeness framework to obtain the actual proof search procedure (cf. Section 2.2). First, we define a relational version of the *effect* of a rule, called *eff*. To use the framework, we need to prove three properties: that the set of well formed proof states is closed under *eff* (*eff-S*), that it is always possible to apply some rule (*enabled-R*), and that the rules that can be applied are still possible to apply after applying other rules (*per*). We do not need to restrict the set of well formed proof states, so the first property is trivial. Since all of our rules can always be applied (they simply do nothing if they do not match the sequent), the other two properties are also trivial. We can thus instantiate the framework with our effect relation and stream of rules. This allows us to define the prover using the *mkTree* function from the framework:

```
definition secavProver ≡ mkTree rules
```

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

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

```
abbreviation rootSequent t ≡ snd (fst (root t))
```

3.2 Making the prover executable

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

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

4 Soundness

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

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

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

Proof. By induction on z for arbitrary *pre* and A.

For the empty sequent, the thesis holds immediately as we get by assumption and the definition of *children* that we can derive $\Vdash$ *pre*.

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

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

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

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

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

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

The remaining α- and β-cases are similar. In the δ-cases we prove that the constant used by the prover is new to the sequent, as required by the SeCaV δ-rules.

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

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

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

▶ **Corollary 2** (Proof tree to SeCaV). *We derive a sequent from derivations of its children:*

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

We obtain the following soundness theorem from the abstract soundness framework.

▶ **Theorem 3** (Prover soundness wrt. SeCaV). *The root sequent of any finite, well formed proof tree has a derivation in SeCaV:*

assumes $\mathit{tfinite}\ t$ **and** $\mathit{wf}\ t$
shows $\Vdash \mathit{rootSequent}\ t$

5 Completeness

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

▶ **Lemma 4** (Prover cases). *The proof tree* generated by the prover *is either finite and well formed or there exists a saturated escape path with our initial state as root:*

defines $t \equiv \mathit{secavProver}\ (A,\ z)$
shows $(\mathit{fst}\ (\mathit{root}\ t) = (A,\ z) \land \mathit{wf}\ t \land \mathit{tfinite}\ t) \lor$
  $(\exists\ \mathit{steps}.\ \mathit{fst}\ (\mathit{shd}\ \mathit{steps}) = (A,\ z) \land \mathit{epath}\ \mathit{steps} \land \mathit{Saturated}\ \mathit{steps})$

In the first case, the sequent has a proof (cf. Section 4). In the second case, we need to build a countermodel from the saturated escape path to contradict validity of the sequent. The rest of this section does exactly that. Inspired by Ben-Ari [2] and Ridge [38], we start off by giving a definition of Hintikka sets over a restricted set of terms (Section 5.1). We show that the set of formulas on saturated escape paths fulfill all Hintikka requirements when we take the set of terms to be the terms on the path (Section 5.2). We then define a countermodel for any formula in such a set using a new semantics that bounds quantifiers by an explicit set rather than by types alone (Section 5.3). Finally we tie these results together to show that the prover terminates for all sequents that are valid under our new semantics (Section 5.4). In Section 6.1 we use existing results to prove completeness of the prover wrt. the SeCaV semantics.

locale *Hintikka* =
  fixes H :: *fm set*
  assumes
    Basic: $\textit{Pre } n \textit{ ts} \in H \Longrightarrow \textit{Neg } (\textit{Pre } n \textit{ ts}) \notin H$ **and**
    AlphaDis: $\textit{Dis } p\ q \in H \Longrightarrow p \in H \wedge q \in H$ **and**
    BetaDis: $\textit{Neg } (\textit{Dis } p\ q) \in H \Longrightarrow \textit{Neg } p \in H \vee \textit{Neg } q \in H$ **and**
    GammaExi: $\textit{Exi } p \in H \Longrightarrow \forall t \in \textit{terms } H.\ \textit{sub } 0\ t\ p \in H$ **and**
    DeltaExi: $\textit{Neg } (\textit{Exi } p) \in H \Longrightarrow \exists t \in \textit{terms } H.\ \textit{Neg } (\textit{sub } 0\ t\ p) \in H$ **and**
    ⋮
    Neg: $\textit{Neg } (\textit{Neg } p) \in H \Longrightarrow p \in H$

Figure 4 Abridged list of requirements for a set of formulas H to be a Hintikka set.

5.1 Hintikka

First, by the *terms* of a set of formulas H we mean the following:

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

This set contains an arbitrary (but fixed) constant, *Fun 0 []*, when H itself contains no terms. Otherwise it contains all subterms of all formulas in H.

Figure 4 contains an abridged definition of a Hintikka set H. Here, we use a locale slightly differently to the previous ones, in that we have specify no conclusions, only premises: the formula set H and the requirements *Basic*, *AlphaDis*, etc. The omitted requirements are similar to the ones shown. This use simply allows us to assume *Hintikka* H in a theorem and know that the set H then fulfills the stated requirements. Similarly, we can prove that a set H is *Hintikka* by proving that it fulfills the requirements. It is important to note that in the γ- and δ-cases, the quantifiers only range over the *terms* of H.

5.2 Saturated escape paths are Hintikka

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

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

The function *sset* returns the set of steps and *pseq* extracts the sequent from each.

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

We will need a number of intermediate results.

5.2.1 Unaffected formulas

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

We then prove the following key preservation lemma:

▶ **Lemma 5** (*effect* preserves unaffected formulas). *Assume formula p occurs in sequent z and the rule r does not affect p. Then p also occurs in all children of z as given by effect ($|{\in}|$ denotes membership of a finite set):*

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

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

We lift this to escape paths:

▶ **Lemma 6** (Escape paths preserve unaffected formulas). *Assume formula p occurs in some sequent at the head of an escape path which consists of a prefix pre, where none of the rules affect p, and a suffix suf. Then p occurs at the head of suf:*

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

Next, notice the following property of streams:

▶ **Lemma 7** (Eventual prefix). *When a property P eventually holds of a stream, then the stream is comprised of a prefix of n (possibly zero) elements for which P does not hold and then a suffix that starts with an element for which P does hold:*

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

Saturation states that a rule is eventually applied and Lemmas 6 and 7 combine to state that any affected formulas are preserved until then.

5.2.2 Affected formulas

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

▶ **Lemma 8** (Parts in effect). *For any formula p in a sequent z, the effect of rule r on z includes some part of r's effect on p:*

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

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

▶ **Corollary 9.** *Example effect of the NegNeg rule on a double-negated formula p:*

> **corollary** $\mathit{Neg}\ (\mathit{Neg}\ p) \in \mathit{set}\ z \Longrightarrow (B,\ z')\ |{\in}|\ \mathit{effect}\ \mathit{NegNeg}\ (A,\ z) \Longrightarrow p \in \mathit{set}\ z'$

5.2.3 Hintikka requirements

We then need to prove the following:

▶ **Theorem 10** (Hintikka escape paths). *Saturated escape paths fulfill all Hintikka requirements:*

> **assumes** $\mathit{epath}\ \mathit{steps}$ **and** $\mathit{Saturated}\ \mathit{steps}$
> **shows** $\mathit{Hintikka}\ (\mathit{tree\text{-}fms}\ \mathit{steps})$

Proof. This boils down to proving each requirement of Figure 4 (and those omitted there). We give a couple of examples and refer to the formalization for the full details.

For *Basic*, assume towards a contradiction that both a predicate and its negation appear on the branch. By preservation of formulas (Lemma 6), both appear in the same sequent at some point. But then *branchDone* holds for that sequent, so it has no children and the branch would terminate. This contradicts that escape paths are infinite, so *Basic* must hold.

For *AlphaDis*, assume that *Dis p q* appears on the branch. Then it appears at some step *n*. By saturation of the escape path, *AlphaDis* is eventually applied at some (earliest) step $n + k$. By Lemma 6, *Dis p q* is preserved until then. So by the effect of rule *AlphaDis*, both *p* and *q* appear at step $n + k + 1$. The cases for the β- and δ-requirements are very similar.

For *GammaExi* assume that *Exi p* occurs at step *n*. We need to show that it is instantiated with all terms that (eventually) appear on the branch. Fix an arbitrary such term *t*. There must be some point *m* where *t* appears in a sequent. Thus at every point greater than *m*, term *t* appears in the term list which is part of the proof state. By saturation, at some step greater than *n + m + 1*, rule *GammaExi* is applied. The formula *Exi p* is preserved until this stage (Lemma 6) and the term list only grows, so *t* is too. Thus, at the next step, *sub 0 t p* occurs on the branch as desired. ◀

5.3 Countermodel

We need to build a countermodel for any formula in a Hintikka set to contradict the validity of any formula on a saturated escape path. We do this in the usual term model with a (bounded) Herbrand interpretation. Unfortunately, we cannot build a countermodel in the original semantics where the universe is specified as a type, since we cannot form the type of terms in a given Hintikka set (the *typedef* command does not support free variables). Instead, we introduce a custom bounded semantics.

5.3.1 Bounded semantics

The bounded semantics is exactly like the usual semantics (cf. Figure 2) except for an extra argument *u*, standing for the universe, which bounds the range of the quantifiers in the following cases:

| *usemantics u e f g* (*Exi p*) = ($\exists$ *x* $\in$ *u*. *usemantics u* (*SeCaV.shift e 0 x*) *f g p*)
| *usemantics u e f g* (*Uni p*) = ($\forall$ *x* $\in$ *u*. *usemantics u* (*SeCaV.shift e 0 x*) *f g p*)

This leads to the following natural requirements on environments *e* and function denotations *f*, namely that they must stay inside *u*:

definition *is-env u e* $\equiv$ $\forall$ *n*. *e n* $\in$ *u*
definition *is-fdenot u f* $\equiv$ $\forall$ *i l*. *list-all* (λ*x*. *x* $\in$ *u*) *l* $\longrightarrow$ *f i l* $\in$ *u*

In general, we only consider environments and function denotations that satisfy these requirements and call them (and any model based on them) *well formed.* When *u* = *UNIV*, we do not actually bound the quantifiers and the two semantics coincide.

The SeCaV proof system (cf. Figure 3) is sound for the bounded semantics too.

▶ **Theorem 11** (SeCaV is sound for the bounded semantics). *Given a SeCaV derivation of sequent z and a well formed model, some formula p in z is satisfied in that model:*

assumes $\Vdash$ *z* **and** *is-env u e* **and** *is-fdenot u f*
shows $\exists$ *p* $\in$ *set z*. *usemantics u e f g p*

Proof. The proof closely resembles the original soundness proof (cf. [19]). ◀

We abbreviate validity of a sequent in the bounded semantics as *uvalid*:

abbreviation $uvalid\ z \equiv \forall\, u\ (e :: nat \Rightarrow tm)\ f\ g.\ is\text{-}env\ u\ e \longrightarrow is\text{-}fdenot\ u\ f \longrightarrow$
$(\exists\, p \in set\ z.\ usemantics\ u\ e\ f\ g\ p)$

Namely, for all universes and well formed models, some formula in the sequent is satisfied in the bounded semantics at that universe by that model.

5.3.2 Model construction

Our countermodel is given by a bounded Herbrand interpretation where terms are interpreted as themselves when they appear in the universe *terms H* and as an arbitrary term otherwise.

▶ **Definition 12** (Countermodel induced by Hintikka set S). *We abbreviate the model as $M\ S$:*

abbreviation $E\ S\ n \equiv if\ Var\ n \in terms\ S\ then\ Var\ n\ else\ SOME\ t.\ t \in terms\ S$
abbreviation $F\ S\ i\ l \equiv if\ Fun\ i\ l \in terms\ S\ then\ Fun\ i\ l\ else\ SOME\ t.\ t \in terms\ S$
abbreviation $G\ S\ n\ ts \equiv Neg\ (Pre\ n\ ts) \in S$
abbreviation $M\ S \equiv usemantics\ (terms\ S)\ (E\ S)\ (F\ S)\ (G\ S)$

The definition of G is what makes this a countermodel rather than a model: a predicate is satisfied exactly when its negation is present in the Hintikka set.

Importantly, these definitions are *well formed*:

▶ **Lemma 13** (Well formed countermodel). *Definition 12 is well formed:*

shows $is\text{-}env\ (terms\ S)\ (E\ S)$
shows $is\text{-}fdenot\ (terms\ S)\ (F\ S)$

Proof. By the construction of E and F and the nonemptiness of *terms S*. ◀

▶ **Theorem 14** (Model existence). *The given model falsifies any formula p in Hintikka set S:*

assumes *Hintikka S*
shows $(p \in S \longrightarrow \neg\ M\ S\ p) \wedge (Neg\ p \in S \longrightarrow M\ S\ p)$

Proof. By induction on the size of the formula p (substitution instances are smaller than the quantified formulas they arise from). The second part of the thesis is needed when the Hintikka requirements concern negated formulas. We show a few cases here and refer to the formalization for the full details. The cases omitted here are similar to those shown.

Assume $p = Pre\ n\ ts$ occurs in S. We need to show that the given model falsifies p. Since *terms S* is downwards closed by construction, *ts* is interpreted as itself by the bounded Herbrand interpretation. Moreover, by the *Basic* requirement, we know that *Neg p* is not in S and is therefore satisfied. Thus, p is falsified.

Assume $p = Dis\ q\ r$ occurs negated in S. Then by the *BetaDis* requirement, either *Neg q* or *Neg r* occurs in S. The induction hypothesis applies to these, so p is satisfied as desired.

Assume $p = Uni\ q$ occurs in S. By the *DeltaUni* requirement, so does some instance *sub 0 t q* for a term t in *terms S*. By the induction hypothesis, this is falsified by $M\ S$, and by its origin, t is interpreted as itself. Thus, we have a counterexample that falsifies p.

Assume $p = Exi\ q$ occurs in S. By the *GammaExi* requirement, so do all instances using terms from S. Thus, these are all falsified by the model. These terms from S are interpreted as themselves by definition so we have no witness for p in *terms S* and $M\ S$ falsifies it. ◀

We note that the above proof works for open and closed formulas alike because we consider both bound and free variables to be subterms of a formula.

5.4 Result

We start off by proving completeness for *uvalid* sequents. We need to relate these to saturated escape paths.

▶ **Lemma 15** (Saturated escape paths contradict uvalidity). *A sequent z with a saturated escape path, steps, cannot be uvalid:*

> **assumes** $\mathit{fst}\ (\mathit{shd}\ \mathit{steps}) = (A,\ z)$ **and** $\mathit{epath}\ \mathit{steps}$ **and** $\mathit{Saturated}\ \mathit{steps}$
> **shows** $\neg\ \mathit{uvalid}\ z$

Proof. Assume towards a contradiction that z is *uvalid*. By Theorem 10 the formulas on *steps* form a Hintikka set S. Every formula p in z also occurs in S, so by Theorem 14, the well formed model $M\ S$ (Lemma 13) falsifies all of them. This contradicts the uvalidity of z. ◀

This leads to completeness for *uvalid* sequents:

▶ **Theorem 16** (Completeness wrt. *uvalid*). *The prover terminates for uvalid sequents:*

> **assumes** $\mathit{uvalid}\ z$
> **defines** $t \equiv \mathit{secavProver}\ (A,\ z)$
> **shows** $\mathit{fst}\ (\mathit{root}\ t) = (A,\ z) \wedge \mathit{wf}\ t \wedge \mathit{tfinite}\ t$

Proof. From the abstract framework (Lemma 4), either the thesis holds or a saturated escape path exists for our sequent, but assumed uvalidity and Lemma 15 contradict the latter. ◀

▶ **Corollary 17** (Completeness wrt. SeCaV). *Termination for sequents derivable in SeCaV:*

> **assumes** $\Vdash z$
> **defines** $t \equiv \mathit{secavProver}\ (A,\ z)$
> **shows** $\mathit{fst}\ (\mathit{root}\ t) = (A,\ z) \wedge \mathit{wf}\ t \wedge \mathit{tfinite}\ t$

Proof. By the soundness of SeCaV (Theorem 11) and Theorem 16 for *uvalid* sequents. ◀

6 Results and discussion

We have presented an automated theorem prover for the Sequent Calculus Verifier system. The prover is capable of proving a number of selected exercise formulas very quickly, including formulas which are quite difficult for humans to prove. The prover does have some limitations, mostly related to performance and length of the generated proofs, since our proof search procedure is not very optimized for either of these metrics. In particular, our prover always instantiates quantified formulas with all terms in the sequent and breaks down all formulas as much as possible, even when some formulas are "obviously" irrelevant to the proof.

6.1 Summary of theorems

We have proven soundness and completeness of the proof search procedure with regards to the proof system of SeCaV (see Figure 3). For soundness, this was done directly (in Theorem 3), while we took a detour through our notion of a bounded semantics to prove completeness (in Theorems 11 and 16, which lead to Corollary 17). To justify the introduction of our bounded semantics, we can use the existing soundness and completeness theorems of the SeCaV proof system [19] and our results to prove that validity in the two semantics coincide. Additionally, a number of easy corollaries further linking the prover, the proof system and the two semantics follow from our results, and have been collected in Figure 5. In the figure, the interpretations are implicitly universally quantified and for the bounded semantics we only consider well formed interpretations.

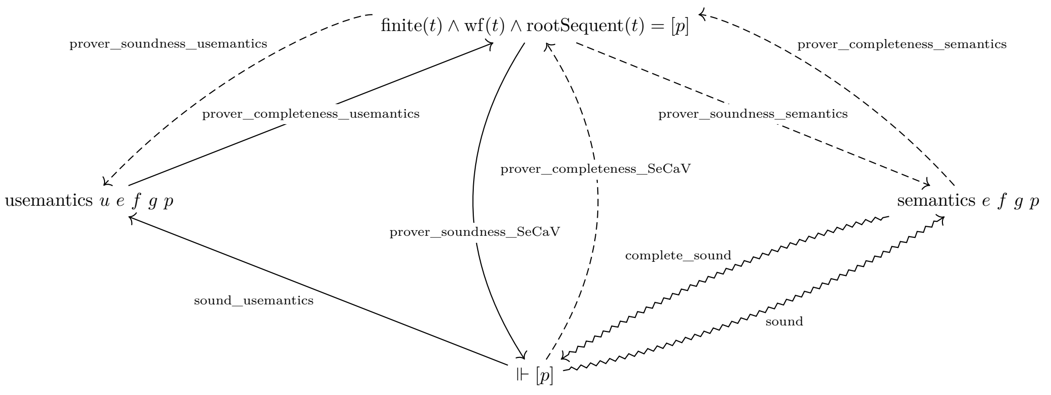
finite(t) ∧ wf(t) ∧ rootSequent(t) = [p]
prover_soundness_usemantics
prover_completeness_semantics
prover_completeness_usemantics
prover_soundness_semantics
prover_completeness_SeCaV
usemantics u e f g p
semantics e f g p
prover_soundness_SeCaV
complete_sound
sound_usemantics
sound
⊩ [p]

Figure 5 Overview of our results. Solid arrows represent our main contributions, squiggly arrows represent theorems of the existing SeCaV system, and dashed arrows represent easy corollaries.

6.2 Example proofs

Famously, we must beware of a program that has only been proven correct, but not tested. To demonstrate that the automated theorem prover works, we examine some simple generated proofs. The prover generates proofs in the SeCaV Unshortener format: first comes the formula to be proven, then the names of proof rules to apply and the resulting sequent after each application, with each formula in a sequent on its own line. Arguments to predicates and functions are given in square brackets and parentheses are used to disambiguate formulas.

We start with perhaps the simplest possible classical example, that $\neg p \vee p$. Figure 6a shows the proof generated by the prover. This is the shortest possible proof of the formula in the SeCaV system, and the prover is thus on par with a human in this very simple case.

The next example is $\neg p(a) \vee \exists x. p(x)$. Figure 6b contains the generated proof. It can be shortened since the quantified formula only needs to be instantiated once, by a. However, the prover always duplicates a γ-formula before instantiating it with *all* terms on the branch.

6.3 Verification challenges

While verifying the prover, we discovered that our initial version was unsound due to a missing update of the term list when applying (multiple) δ-rules to a sequent. The attempted soundness proof failed in exactly this case, pointing us directly to the issue. Thus, the formal verification caught a critical flaw that we had missed in our testing and helped us fix it.

We have designed the prover to be easily verified and it mostly was. Especially the abstract framework worked well for our novel case with a deterministic prover for first-order logic. One obstacle, however, was in using a type to represent the domain in the SeCaV semantics (cf. Figure 2). To build the countermodel, we need the domain to contain only the terms on the saturated escape path, but we cannot form this type, which depends on a local variable, in Isabelle/HOL. Here we would benefit from Isabelle integration of the work by Kunčar and Popescu [25] which adds exactly this capability to higher-order logic. Instead we introduced the bounded semantics ("the set-based relativization" in their terminology [25]) and proved a new soundness result for it (cf. Section 5.3.1). Otherwise the largest issue was dealing with substitutions using de Bruijn indices. We are excited to see how recent work by Blanchette et al. [7] for reasoning about syntax with bindings improves matters in this area.

```
Dis (Neg p) p

AlphaDis
  Neg p
  p
Ext
  p
  Neg p
Basic
```

(a) $\neg p \vee p$.

```
Dis (Neg (p [a]))
    (Exi (p [0]))

AlphaDis
  Neg (p [a])
  Exi (p [0])
Ext
  Exi (p [0])
  Exi (p [0])
  Neg (p [a])
GammaExi [a]
  p [a]
  Exi (p [0])
  Neg (p [a])
```

(b) $\neg p(a) \vee \exists x.p(x)$.

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

Figure 6b continued.

Figure 6 Proofs generated by the prover in SeCaV Unshortener format.

6.4 Limitations and future work

There are a number of limitations and possibilities for optimization in the proof search itself. Most importantly, the focus of the procedure is on completeness, not performance. Our prover is much slower than state-of-the-art provers such as Vampire [24], but our goal was not to compete on speed, but simply to show that formal verification of provers with advanced features such as generation of proof certificates and support for functions is possible. The prover also cannot output counterexamples, even though these can be detected in some cases: our prover simply never terminates on invalid formulas.

We believe that the approach used for our prover is extendable to more sophisticated and optimized proof search procedures, albeit with considerably more work needed to formally verify them. The most obvious opportunity for optimization is controlling the order of proof rules. In systems with unordered sequents, it is generally better to apply as many α-rules as possible before applying β-rules to avoid duplicating work, but the prover simply applies rules in a fixed order. As mentioned in Section 3, this optimization can be done by working with "meta-rules" corresponding to groups of SeCaV rules such that a meta-rule e.g. applies as many α-rules as possible before continuing to the next "phase" of the proof. We have attempted to implement this, but found that it complicates the proofs considerably since this idea makes it much harder to determine when a proof rule is actually applied. In the proof of fairness and the proof that the formulas on saturated escape paths form Hintikka sets, we need to know that certain formulas are preserved until proof rules are eventually applied to them. By introducing phases in the proof, proving this becomes much more difficult, since we then need to prove that each phase actually ends (requiring some measure which depends on the specific sequents in question), and to locate each rule within the meta-rule it is part of. We thus leave optimizations in this vein as future work. We note that, since the SeCaV system requires application of the Ext rule to permute sequents, and proof rules only apply to the first formula in a sequent, the optimization described above may not always reduce the number of SeCaV proof steps needed to prove a formula, and some heuristics would probably be needed to produce reasonably short proofs in all cases.

Another optimization could be to only support closed formulas and thus reduce the number of subterms of a given formula. For our current Herbrand interpretation, we need variables to be subterms, but if we only considered closed terms, we could do away with this.

The length of proofs could also be optimized by performing more post-processing of the found proofs, for example by removing unnecessary instantiations or rule applications that do not contribute to proving a branch. This would not improve the performance in the sense that the prover would still spend the same amount of time finding the proof, but it could reduce the length of some proofs significantly. The proof trees generated by the prover already require some (unverified) post-processing to obtain proofs in the SeCaV system. It would be interesting to move these steps from Haskell into Isabelle/HOL and extend the proofs to cover them.

7 Conclusion

We have designed, implemented and verified an automated theorem prover for first-order logic with functions in Isabelle/HOL. We have used an existing framework in a novel way to get us part of the way towards completeness and extended existing techniques on countermodels over restricted domains to reach our destination. We build on the existing SeCaV system and contribute an automatic way of finding derivations to the project. Thus, we have demonstrated the utility of Isabelle/HOL for implementing and verifying executable software and the strength of its libraries in doing so. Our prover handles the full syntax of first-order logic with functions and constructs human-readable proof certificates in a sequent calculus. We hope our work inspires others to verify more sophisticated provers in the same vein.

References

1 Clemens Ballarin. Locales: A module system for mathematical theories. *Journal of Automated Reasoning*, 52(2):123–153, 2014. `doi:10.1007/s10817-013-9284-7`.

2 Mordechai Ben-Ari. *Mathematical Logic for Computer Science*. Springer London, 2012. `doi:10.1007/978-1-4471-4129-7`.

3 A. Bentkamp, J. Blanchette, S. Tourret, and P. Vukmirović. Superposition for Full Higher-order Logic. In A. Platzer and G. Sutcliffe, editors, *Proceedings of the 28th International Conference on Automated Deduction*, number 12699 in Lecture Notes in Computer Science, pages 396–412. Springer-Verlag, 2021. `doi:10.1007/978-3-030-79876-5_23`.

4 Stefan Berghofer. First-order logic according to Fitting. *Archive of Formal Proofs*, August 2007. Formal proof development. URL: `https://isa-afp.org/entries/FOL-Fitting.html`.

5 Jasmin Christian Blanchette. Formalizing the metatheory of logical calculi and automatic provers in Isabelle/HOL (invited talk). In Assia Mahboubi and Magnus O. Myreen, editors, *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019*, pages 1–13. ACM, 2019. `doi:10.1145/3293880.3294087`.

6 Jasmin Christian Blanchette, Mathias Fleury, Peter Lammich, and Christoph Weidenbach. A verified SAT solver framework with learn, forget, restart, and incrementality. *Journal of Automated Reasoning*, 61(1-4):333–365, 2018. `doi:10.1007/s10817-018-9455-7`.

7 Jasmin Christian Blanchette, Lorenzo Gheri, Andrei Popescu, and Dmitriy Traytel. Bindings as bounded natural functors. *Proc. ACM Program. Lang.*, 3(POPL):22:1–22:34, 2019. `doi:10.1145/3290335`.

8 Jasmin Christian Blanchette and Andrei Popescu. Mechanizing the metatheory of Sledgehammer. In Pascal Fontaine, Christophe Ringeissen, and Renate A. Schmidt, editors, *Frontiers of Combining Systems*, pages 245–260, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. `doi:10.1007/978-3-642-40885-4_17`.

9 Jasmin Christian Blanchette, Andrei Popescu, and Dmitriy Traytel. Abstract completeness. *Archive of Formal Proofs*, April 2014. Formal proof development. URL: `https://isa-afp.org/entries/Abstract_Completeness.html`.

10 Jasmin Christian Blanchette, Andrei Popescu, and Dmitriy Traytel. Unified classical logic completeness. In Stéphane Demri, Deepak Kapur, and Christoph Weidenbach, editors, *Automated Reasoning*, pages 46–60, Cham, 2014. Springer International Publishing. `doi:10.1007/978-3-319-08587-6_4`.

11 Jasmin Christian Blanchette, Andrei Popescu, and Dmitriy Traytel. Soundness and completeness proofs by coinductive methods. *Journal of Automated Reasoning*, 58:149–179, 2017. `doi:10.1007/s10817-016-9391-3`.

12 Joachim Breitner. Visual theorem proving with the Incredible Proof Machine. In J. Blanchette and S. Merz, editors, *Interactive Theorem Proving*, volume ITP 2016. Springer, Cham, 2016. `doi:10.1007/978-3-319-43144-4_8`.

13 Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. `doi:10.1007/978-3-540-78800-3_24`.

14 Mathias Fleury. Optimizing a verified SAT solver. In Julia M. Badger and Kristin Yvonne Rozier, editors, *NASA Formal Methods – 11th International Symposium, NFM 2019, Houston, TX, USA, May 7-9, 2019, Proceedings*, volume 11460 of *Lecture Notes in Computer Science*, pages 148–165. Springer, 2019. `doi:10.1007/978-3-030-20652-9_10`.

15 Asta Halkjær From. Formalized soundness and completeness of epistemic logic. In Alexandra Silva, Renata Wassermann, and Ruy J. G. B. de Queiroz, editors, *Logic, Language, Information, and Computation – 27th International Workshop, WoLLIC 2021, Virtual Event, October 5-8, 2021, Proceedings*, volume 13038 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2021. `doi:10.1007/978-3-030-88853-4_1`.

16 Asta Halkjær From, Frederik Krogsdal Jacobsen, and Jørgen Villadsen. SeCaV: A sequent calculus verifier in Isabelle/HOL. In Mauricio Ayala-Rincon and Eduardo Bonelli, editors, Proceedings 16th *Logical and Semantic Frameworks with Applications,* Buenos Aires, Argentina (Online), 23rd – 24th July, 2021, volume 357 of *Electronic Proceedings in Theoretical Computer Science*, pages 38–55. Open Publishing Association, 2022. `doi:10.4204/EPTCS.357.4`.

17 Asta Halkjær From. Epistemic logic: Completeness of modal logics. *Archive of Formal Proofs*, October 2018. Formal proof development. URL: `https://isa-afp.org/entries/Epistemic_Logic.html`.

18 Asta Halkjær From and Frederik Krogsdal Jacobsen. A sequent calculus prover for first-order logic with functions. *Archive of Formal Proofs*, January 2022. Formal proof development. URL: `https://isa-afp.org/entries/FOL_Seq_Calc2.html`.

19 Asta Halkjær From, Alexander Birch Jensen, Anders Schlichtkrull, and Jørgen Villadsen. Teaching a formalized logical calculus. *Electronic Proceedings in Theoretical Computer Science*, 313:73–92, 2020. `doi:10.4204/EPTCS.313.5`.

20 Asta Halkjær From, Jørgen Villadsen, and Patrick Blackburn. Isabelle/HOL as a meta-language for teaching logic. *Electronic Proceedings in Theoretical Computer Science*, 328:18–34, October 2020. `doi:10.4204/eptcs.328.2`.

21 Frederik Krogsdal Jacobsen. Formalization of logical systems in Isabelle: An automated theorem prover for the Sequent Calculus Verifier. Master's thesis, Technical University of Denmark, June 2021. URL: `https://findit.dtu.dk/en/catalog/2691928304`.

22 Alexander Birch Jensen, John Bruntse Larsen, Anders Schlichtkrull, and Jørgen Villadsen. Programming and verifying a declarative first-order prover in Isabelle/HOL. *AI Communications*, 31(3):281–299, 2018. `doi:10.3233/AIC-180764`.

23 Florian Kammüller, Markus Wenzel, and Lawrence C. Paulson. Locales – A sectioning concept for Isabelle. In Yves Bertot, Gilles Dowek, André Hirschowitz, Christine Paulin-Mohring, and Laurent Théry, editors, *Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs'99, Nice, France, September, 1999, Proceedings*, volume 1690 of *Lecture Notes in Computer Science*, pages 149–166. Springer, 1999. `doi:10.1007/3-540-48256-3_11`.

24 Laura Kovács and Andrei Voronkov. First-order theorem proving and Vampire. *Lecture Notes in Computer Science*, 8044, 2013. `doi:0.1007/978-3-642-39799-8_1`.

25 Ondrej Kunčar and Andrei Popescu. From types to sets by local type definition in higher-order logic. *Journal of Automated Reasoning*, 62(2):237–260, 2019. `doi:10.1007/s10817-018-9464-6`.

26 Peter Lammich. The GRAT tool chain. In Serge Gaspers and Toby Walsh, editors, *Theory and Applications of Satisfiability Testing – SAT 2017*, pages 457–463, Cham, 2017. Springer International Publishing. `doi:10.1007/978-3-319-66263-3_29`.

27 Peter Lammich. Efficient verified (UN)SAT certificate checking. *Journal of Automated Reasoning*, 64(3):513–532, 2020. `doi:10.1007/s10817-019-09525-z`.

28 Stephane Lescuyer. *Formalizing and Implementing a Reflexive Tactic for Automated Deduction in Coq*. Phd thesis, Université Paris Sud – Paris XI, January 2011. URL: `https://tel.archives-ouvertes.fr/tel-00713668`.

29 Filip Marić. Formal verification of modern SAT solvers. *Archive of Formal Proofs*, July 2008. Formal proof development. URL: `https://isa-afp.org/entries/SATSolverVerification.html`.

30 Filip Marić. Formal verification of a modern SAT solver by shallow embedding into Isabelle/HOL. *Theoretical Computer Science*, 411(50):4333–4356, 2010. `doi:10.1016/j.tcs.2010.09.014`.

31 Filip Marić, Mirko Spasić, and René Thiemann. An incremental simplex algorithm with unsatisfiable core generation. *Archive of Formal Proofs*, August 2018. Formal proof development. URL: `https://isa-afp.org/entries/Simplex.html`.

32 Julius Michaelis and Tobias Nipkow. Propositional proof systems. *Archive of Formal Proofs*, June 2017. Formal proof development. URL: `https://isa-afp.org/entries/Propositional_Proof_Systems.html`.

33 Julius Michaelis and Tobias Nipkow. Formalized Proof Systems for Propositional Logic. In Andreas Abel, Fredrik Nordvall Forsberg, and Ambrus Kaposi, editors, *23rd International Conference on Types for Proofs and Programs (TYPES 2017)*, volume 104 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 5:1–5:16, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/LIPIcs.TYPES.2017.5`.

34 Dominique Pastre. MUSCADET 2.3: A knowledge-based theorem prover based on natural deduction. *Lecture Notes in Computer Science (including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2083:685–689, 2001. `doi:10.1007/3-540-45744-5_56`.

35 Francis Jeffry Pelletier. Automated natural deduction in THINKER. *Studia Logica*, 60(1):3–43, 1998. `doi:10.1023/A:1005035316026`.

36 Nicolas Peltier. Propositional resolution and prime implicates generation. *Archive of Formal Proofs*, March 2016. Formal proof development. URL: `https://isa-afp.org/entries/PropResPI.html`.

37 Nicolas Peltier. A variant of the superposition calculus. *Archive of Formal Proofs*, September 2016. Formal proof development. URL: `https://isa-afp.org/entries/SuperCalc.html`.

38 Tom Ridge and James Margetson. A mechanically verified, sound and complete theorem prover for first order logic. *Lecture Notes in Computer Science*, 3603:294–309, 2005. `doi:10.1007/11541868_19`.

39 Anders Schlichtkrull. Formalization of the resolution calculus for first-order logic. *Journal of Automated Reasoning*, 61(1–4):455–484, 2018. `doi:10.1007/s10817-017-9447-z`.

40 Anders Schlichtkrull, Jasmin Christian Blanchette, and Dmitriy Traytel. A verified prover based on ordered resolution. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2019, pages 152–165, New York, NY, USA, 2019. Association for Computing Machinery. `doi:10.1145/3293880.3294100`.

41 Anders Schlichtkrull, Jasmin Christian Blanchette, Dmitriy Traytel, and Uwe Waldmann. Formalizing Bachmair and Ganzinger's ordered resolution prover. In Didier Galmiche, Stephan Schulz, and Roberto Sebastiani, editors, *Automated Reasoning*, pages 89–107, Cham, 2018. Springer International Publishing. doi:10.1007/978-3-319-94205-6_7.

42 Anders Schlichtkrull and Jørgen Villadsen. Paraconsistency. *Archive of Formal Proofs*, December 2016. Formal proof development. URL: https://isa-afp.org/entries/Paraconsistency.html.

43 Natarajan Shankar and Marc Vaucher. The mechanical verification of a DPLL-based satisfiability solver. *Electronic Notes in Theoretical Computer Science*, 269:3–17, 2011. Proceedings of the Fifth Logical and Semantic Frameworks, with Applications Workshop (LSFA 2010). doi:10.1016/j.entcs.2011.03.002.

44 Raymond M. Smullyan. *First-order logic.* Dover Publications, 1995.

45 Mirko Spasić and Filip Marić. Formalization of incremental simplex algorithm by stepwise refinement. In Dimitra Giannakopoulou and Dominique Méry, editors, *FM 2012: Formal Methods*, pages 434–449, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. doi:10.1007/978-3-642-32759-9_35.

46 Jørgen Villadsen, Asta Halkjær From, Alexander Birch Jensen, and Anders Schlichtkrull. Interactive theorem proving for logic and information. In Roussanka Loukanova, editor, *Natural Language Processing in Artificial Intelligence — NLPinAI 2021*, pages 25–48, Cham, 2022. Springer International Publishing. doi:10.1007/978-3-030-90138-7_2.

47 Jørgen Villadsen and Frederik Krogsdal Jacobsen. Using Isabelle in two courses on logic and automated reasoning. In João F. Ferreira, Alexandra Mendes, and Claudio Menghi, editors, *Formal Methods Teaching*, pages 117–132, Cham, 2021. Springer International Publishing. doi:10.1007/978-3-030-91550-6_9.

48 Jørgen Villadsen and Anders Schlichtkrull. Formalizing a paraconsistent logic in the Isabelle proof assistant. In Abdelkader Hameurlain, Josef Küng, Roland Wagner, and Hendrik Decker, editors, *Transactions on Large-Scale Data- and Knowledge-Centered Systems XXXIV: Special Issue on Consistency and Inconsistency in Data-Centric Applications*, pages 92–122, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg. doi:10.1007/978-3-662-55947-5_5.

49 Jørgen Villadsen, Andreas Halkjær From, and Anders Schlichtkrull. Natural deduction assistant (NaDeA). *Electronic Proceedings in Theoretical Computer Science*, 290(290):14–29, 2019. doi:10.4204/EPTCS.290.2.

50 Jørgen Villadsen, Anders Schlichtkrull, and Andreas Halkjær From. A verified simple prover for first-order logic. *CEUR Workshop Proceedings*, 2162:88–104, 2018. URL: http://ceur-ws.org/Vol-2162/#paper-08.