A Naive Prover for First-Order Logic
A Minimal Example of Analytic Completeness

From, Asta Halkjær; Villadsen, Jørgen

Published in:
Proceedings of the 32nd International Conference on Automated Reasoning with Analytic Tableaux and Related Methods, TABLEAUX 2023

Link to article, DOI:
10.1007/978-3-031-43513-3_25

Publication date:
2023

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

Citation (APA):
From, A. H., & Villadsen, J. (2023). A Naive Prover for First-Order Logic: A Minimal Example of Analytic Completeness. In *Proceedings of the 32nd International Conference on Automated Reasoning with Analytic Tableaux and Related Methods, TABLEAUX 2023* (Vol. 14278, pp. 468–480). Springer. https://doi.org/10.1007/978-3-031-43513-3_25

A Naive Prover for First-Order Logic: A Minimal Example of Analytic Completeness

Asta Halkjær From and Jørgen Villadsen(✉)

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

Abstract. The analytic technique for proving completeness gives a very operational perspective: build a countermodel to the unproved formula from a failed proof attempt in your calculus. We have to be careful, however, that the proof attempt did not fail because our strategy in finding it was flawed. Overcoming this concern requires designing a prover. We design and formalize in Isabelle/HOL a sequent calculus prover for first-order logic with functions. We formalize soundness and completeness theorems using an existing framework and extract executable code to Haskell. The crucial idea is to move complexity from the prover itself to a stream of instructions that it follows. The result serves as a minimal example of the analytic technique, a naive prover for first-order logic, and a case study in formal verification.

Keywords: First-Order Logic · Prover · Completeness · Isabelle/HOL

1 Introduction

We present a sound and complete (naive) prover for classical first-order logic with functions. There are several ways to prove that a proof system for first-order logic is complete. Gödel's approach [14], later refined by Henkin [15] is now known as the *synthetic* way. This technique abstractly builds *maximal consistent (and saturated) sets* of formulas as a bridge between the proof system and the semantics. This is a useful technique and has been used in formalizations of the completeness of axiomatic systems for first-order logic [9] and epistemic logic [8], a tableau system for hybrid logic [7] and more. Unfortunately, as pointed out by Blanchette et al. [5] in the context of formalization in Isabelle/HOL, there is no useful connection between this technique and the execution of an actual prover.

The technique by Beth and Hintikka [17] offers a more operational perspective. Here, we consider unsuccessful proof attempts in the given calculus and build countermodels from these. Such a countermodel refutes the validity of the formula that we tried to prove. To build such a countermodel, however, we must ensure that the proof attempt was sufficiently sophisticated and, essentially, that it would have found a proof if one existed. In proving this property of the proof strategy, we are effectively designing a prover based on the calculus. This means that, in practice, we can extract a prover from our completeness proof.

Blanchette et al. [5] have made this very concrete by developing a framework in Isabelle/HOL for analytic completeness proofs. Their paper includes a first-order logic example, but their entry in the Archive of Formal Proofs [3] only includes a propositional example. In this paper, we describe a *naive prover* based on the framework, designed to be as simple as possible. This augments the framework with a concrete first-order logic example showcasing the analytic technique. Moreover it serves as an introduction to automated reasoning by making explicit the requirements for completeness of a prover for first-order logic. It also serves as a small case study for formal verification in a proof assistant.

Then the question remains of how to design this proof strategy. We want it to be sufficiently intricate to be both sound and complete, but we also want it to be simple enough that we can reasonably demonstrate these properties (in a proof assistant). We might follow something like Ben-Ari's tableau algorithm [1] (essentially sequent calculus), but we discover that it is surprisingly complex. There are nodes with labels, branches with markings, and concerns about which kinds of formulas to process first, later or even together. Instead, we will design a prover with minimal structure that tries to apply sequent calculus proof rules over and over, in the belief that we will eventually apply the right ones.

The problem changes from working out which rule to apply in a given situation, to designing a stream of instructions that will cover whatever we encounter and embedding enough structure into these instructions to keep the prover itself elementary. This perspective shift greatly simplifies the prover: the rules are indexed by formulas and specify exactly what the prover should do in each case. Moreover, the nodes in the proof tree are simply sequents, no additional state is needed. The rules apply straightforwardly to these sequents to form the next nodes of the tree. This simplifies the completeness proof and makes it a non-issue to handle first-order logic with functions, which can otherwise require extra consideration.

The formalization of the (naive) prover is available in the Archive of Formal Proofs [11]. It consists of less than 900 lines of Isabelle/HOL listings, the majority of which are proofs that are not included when exporting Haskell code for the prover. A short, manually written `Main.hs` file augments the exported code with a command line interface and pretty-printed output. The Isabelle theory *Export.thy* includes instructions on how to export and compile the Haskell code (which closely resembles the programs listed here). The code in this paper is exported to LaTeX by Isabelle from the formalization, but differs slightly in names and layout for presentation reasons. Likewise, to focus on essentials, we often omit the technical commands needed in the formalization.

2 Related Work

Blanchette [2] gives an overview of a number of verification efforts including the metatheory of SAT and SMT solvers, the resolution and superposition calculi, and a series of proof systems for propositional logic [18]. The aim is to develop a methodology for formalizing modern research in automated reasoning and the present work points in this direction with a minimal example of a formally verified prover for classical first-order logic based on the sequent calculus.

The prover is based on the abstract completeness framework by Blanchette, Popescu and Traytel [4,5]. Their formalization contains a simple example prover for propositional logic, while their paper contains the ideas for a (naive) prover for first-order logic. Our prover realizes these ideas by formalizing them in Isabelle/HOL. Instead of a prover, Blanchette et al. [5] used the framework to formalize soundness and completeness of a *calculus* for first-order logic with equality in negation normal form. From and Jacobsen [10,12] used the framework to formalize a much less naive prover for first-order logic based on the SeCaV proof system [13]. Instead of indexed rules, they employ "multi-rules" that apply to every applicable formula in a sequent at once and they store more than just the sequent at each node in the proof tree. Their prover performs better, but the formalization does not enjoy the simplicity of the naive prover, with close to 3000 lines of Isabelle/HOL against 900 lines.

The indexed rules of the naive prover automatically yield readable proofs. In the same vein, THINKER by Pelletier [21] is a natural deduction proof system and attached automated theorem prover, designed for "direct proofs", as opposed to proofs based on reduction to a resolution system. MUSCADET by Pastre [20] is another automated theorem prover based on natural deduction. Neither of these has been formally verified. Schulz and Pease [24] focused on readable code rather than proofs. They have developed a saturation-based theorem prover in Python for first-order logic to teach automated theorem proving by example. They have not formally verified soundness and completeness, but our projects are similar.

In the world of formalization, Schlichtkrull et al. [23] formalized an ordered resolution prover for *clausal* first-order logic in Isabelle/HOL. Jensen et al. [16] formalized the soundness, but not the completeness, of a prover for first-order logic with equality in Isabelle/HOL. Villadsen et al. [25] verified a simple prover for first-order logic in Isabelle/HOL aiming for students to understand both the prover and the formalization. That work simplified a formalization by Ridge and Margetson [22]. Neither of the last two provers support functions.

3 Isabelle/HOL Overview

We give a quick overview of the Isabelle/HOL features used in the present paper. Nipkow and Klein [19, Part 1] give a more complete introduction.

The **datatype** command defines a new inductive type from a series of constructors, where each can be given custom syntax. The natural numbers are built from the nullary constructor *0* and unary *Suc*. The constructors *True* and *False* belong to the built-in type *bool*. The usual connectives and quantifiers from first-order logic ($\longrightarrow$, $\forall$, etc.) are available for *bool*, as well as *if-then-else* expressions. The parametric *'a list* is the type of lists with elements of type *'a*. The type variable *'a* stands in the place of another type. Lists are built from [], the empty list, and #, an infix constructor that adjoins an element to an

```
datatype tm                          datatype fm
  = Var nat (#)                        = Falsity (⊥)
  | Fun nat (tm list) (†)              | Pre nat (tm list) (‡)
                                       | Imp fm fm (infixr ⟶ 55)
                                       | Uni fm (∀)
```

Fig. 1. The first-order logic syntax in Isabelle/HOL.

existing list. The notation [*a, b, c*] is shorthand for these primitive operations. The function *set* turns a list into a set of its elements, *map* applies a given function to every element of a list, @ appends two lists, *concat* flattens a list of lists and *upt j k* creates the list $[j, j+1, \ldots, k-1]$. We use [∈] for list membership and [÷] to remove all occurrences of a given element from a list. The two types *′a set* and *′a fset* form sets and finite sets respectively. The usual operations are available on sets. On finite sets they are typically prefixed by *f* as in *fimage*. Two additional types are important: sum types with the two unary constructors *Inl* and *Inr*, and *option* types constructed by the unary *Some* or nullary *None*. Constructors can be examined using *case* expressions.

The **codatatype** command defines a new coinductive type from a series of constructors. The canonical example is the type *′a stream* of "lists with no base case", i.e. infinite sequences. The functions *shd* and *stl* return the head and tail of a stream, respectively, while *flat* transforms a stream of lists into a stream of all the elements in the constituent lists, *sset* returns a set of its elements, *smap* applies a function to every element, !! returns the element at a given index and *sdrop-while* removes a prefix of a stream that satisfies a given predicate. The stream *nats* contains all natural numbers.

The type $A \Rightarrow B$ denotes a function from A to B. Type signatures are specified after "::". Types can be shortened using type synonyms. The term *UNIV* stands for the set of all values of a given type. In this paper, both = and ≡ are used to form new definitions. Function application resembles functional programming languages: $f(x, y)$ is written as $f\ x\ y$ and partial application is allowed. Anonymous functions are built using λ-expressions, e.g. $\lambda n.\ n + n$ for $f(n) = n + n$.

A **locale** in Isabelle/HOL **fixes** a number of terms, then **assumes** a number of properties about those terms. The meta-logical implication $\Longrightarrow$ separates premises from conclusions in each assumption. The keyword **and** acts as a separator. A locale for a group, for instance, *fixes* a set and a binary operation and *assumes* the group axioms.

4 First-Order Logic in Isabelle/HOL

Figure 1 contains a formalization of the syntax of first-order logic as a datatype in Isabelle/HOL. The syntax is *deeply embedded* as an object in the meta-logic so we can manipulate it. We use de Bruijn indices [6] to represent binding: each variable n is bound by the quantifier that is n quantifiers away, moving outwards.

$$\begin{array}{l}
\textbf{type-synonym}\ 'a\ \mathit{var\text{-}denot} = \mathit{nat} \Rightarrow\ 'a \\
\textbf{type-synonym}\ 'a\ \mathit{fun\text{-}denot} = \mathit{nat} \Rightarrow\ 'a\ \mathit{list} \Rightarrow\ 'a \\
\textbf{type-synonym}\ 'a\ \mathit{pre\text{-}denot} = \mathit{nat} \Rightarrow\ 'a\ \mathit{list} \Rightarrow \mathit{bool}
\end{array}$$

$$\begin{array}{l}
\fatsemi :: {'a} \Rightarrow (\mathit{nat} \Rightarrow {'a}) \Rightarrow \mathit{nat} \Rightarrow {'a} \\
(t \fatsemi s)\ \mathit{0} = t \\
(t \fatsemi s)\ (\mathit{Suc}\ n) = s\ n
\end{array}$$

$$\begin{array}{l}
\llparenthesis \text{-}, \text{-} \rrparenthesis :: {'a}\ \mathit{var\text{-}denot} \Rightarrow {'a}\ \mathit{fun\text{-}denot} \Rightarrow \mathit{tm} \Rightarrow {'a} \\
\llparenthesis E,\ F \rrparenthesis\ (\#n) = E\ n \\
\llparenthesis E,\ F \rrparenthesis\ (\dagger f\ \mathit{ts}) = F\ f\ (\mathit{map}\ \llparenthesis E,\ F \rrparenthesis\ \mathit{ts})
\end{array}$$

$$\begin{array}{l}
\llbracket \text{-}, \text{-}, \text{-} \rrbracket :: {'a}\ \mathit{var\text{-}denot} \Rightarrow {'a}\ \mathit{fun\text{-}denot} \Rightarrow {'a}\ \mathit{pre\text{-}denot} \Rightarrow \mathit{fm} \Rightarrow \mathit{bool} \\
\llbracket \text{-}, \text{-}, \text{-} \rrbracket\ \bot = \mathit{False} \\
\llbracket E,\ F,\ G \rrbracket\ (\ddagger P\ \mathit{ts}) = G\ P\ (\mathit{map}\ \llparenthesis E,\ F \rrparenthesis\ \mathit{ts}) \\
\llbracket E,\ F,\ G \rrbracket\ (p \longrightarrow q) = (\llbracket E,\ F,\ G \rrbracket\ p \longrightarrow \llbracket E,\ F,\ G \rrbracket\ q) \\
\llbracket E,\ F,\ G \rrbracket\ (\forall\, p) = (\forall\, x.\ \llbracket x \fatsemi E,\ F,\ G \rrbracket\ p)
\end{array}$$

Fig. 2. The semantics of first-order logic in Isabelle/HOL.

A term t, type *tm*, is then either a variable $\#n$ for some de Bruijn index n (a natural number) or a function application $\dagger f$ [...] for some natural number f representing the function name and list of argument terms. [...]. A formula p, type *fm*, is the constant for falsity, $\bot$, a predicate $\ddagger P$ [...] for some natural number P representing the predicate name and list of argument terms [...], an implication $p_1 \longrightarrow p_2$ between two formulas p_1, p_2 or a universally quantified formula $\forall p$.

Figure 2 contains a formalization of the semantics in Isabelle/HOL. A model consists of three denotations: one each for variables (E), function symbols (F) and predicate symbols (G). Terms evaluate to a member of the domain, here represented as a type variable, while formulas evaluate to truth values in the higher-order logic. We can use the connectives and quantifiers of Isabelle/HOL to interpret the first-order logic syntax. For the universal quantifier, we modify the environment such that we evaluate the quantified variable 0 as every element of the domain.

Figure 3 lists the rules for instantiating a quantifier with a term without capturing any free variables in the process. The operation *lift-tm* increments every variable in the term t by one. The operation *sub-tm* s t applies the substitution s to every variable in term t. The operation *sub-fm* s p applies the substitution s to the formula p, taking account of binders. In the case for $\forall p$, the substitution is augmented using $\fatsemi$ to preserve the bound variable $\#0$ in p and to *lift* the variables in the output of the substitution s to point past the binder. We write the instantiation of a quantified formula $\forall p$ with a concrete term t as $\langle t \rangle p$. The notation $\langle t \rangle$ represents the simultaneous substitution that maps variable 0 to t and every other variable $n+1$ to n to account for the removed binder. Figure 4 lists the operations for generating a variable *fresh* to a list of formulas, i.e. one that does not appear in any formula in the list.

$$\begin{array}{l}
\textit{lift-tm} :: \textit{tm} \Rightarrow \textit{tm} \\
\textit{lift-tm}\ (\#n) = \#(n{+}\mathit{1}) \\
\textit{lift-tm}\ (\dagger f\ ts) = \dagger f\ (\textit{map lift-tm}\ ts)
\end{array}$$

$$\begin{array}{l}
\textit{sub-tm} :: (\textit{nat} \Rightarrow \textit{tm}) \Rightarrow \textit{tm} \Rightarrow \textit{tm} \\
\textit{sub-tm}\ s\ (\#n) = s\ n \\
\textit{sub-tm}\ s\ (\dagger f\ ts) = \dagger f\ (\textit{map}\ (\textit{sub-tm}\ s)\ ts)
\end{array}$$

$$\begin{array}{l}
\textit{sub-fm} :: (\textit{nat} \Rightarrow \textit{tm}) \Rightarrow \textit{fm} \Rightarrow \textit{fm} \\
\textit{sub-fm}\ \text{-}\ \bot = \bot \\
\textit{sub-fm}\ s\ (\ddagger P\ ts) = \ddagger P\ (\textit{map}\ (\textit{sub-tm}\ s)\ ts) \\
\textit{sub-fm}\ s\ (p \longrightarrow q) = \textit{sub-fm}\ s\ p \longrightarrow \textit{sub-fm}\ s\ q \\
\textit{sub-fm}\ s\ (\forall\, p) = \forall\, (\textit{sub-fm}\ (\#\mathit{0} \mathbin{;} \lambda n.\ \textit{lift-tm}\ (s\ n))\ p)
\end{array}$$

$$\begin{array}{l}
\langle\text{-}\rangle :: \textit{tm} \Rightarrow \textit{fm} \Rightarrow \textit{fm} \\
\langle t\rangle \equiv \textit{sub-fm}\ (t \mathbin{;} \#)
\end{array}$$

Fig. 3. The simultaneous substitution and quantifier instantiation in Isabelle/HOL.

$$\begin{array}{l}
\textit{vars-tm} :: \textit{tm} \Rightarrow \textit{nat list} \\
\textit{vars-tm}\ (\#n) = [n] \\
\textit{vars-tm}\ (\dagger\text{-}\ ts) = \textit{concat}\ (\textit{map vars-tm}\ ts)
\end{array}$$

$$\begin{array}{l}
\textit{vars-fm} :: \textit{fm} \Rightarrow \textit{nat list} \\
\textit{vars-fm}\ \bot = [] \\
\textit{vars-fm}\ (\ddagger\text{-}\ ts) = \textit{concat}\ (\textit{map vars-tm}\ ts) \\
\textit{vars-fm}\ (p \longrightarrow q) = \textit{vars-fm}\ p\ @\ \textit{vars-fm}\ q \\
\textit{vars-fm}\ (\forall\, p) = \textit{vars-fm}\ p
\end{array}$$

$$\begin{array}{l}
\textit{vars-fms} :: \textit{fm list} \Rightarrow \textit{nat list} \\
\textit{vars-fms}\ A \equiv \textit{concat}\ (\textit{map vars-fm}\ A)
\end{array}$$

$$\begin{array}{l}
\textit{max-list} :: \textit{nat list} \Rightarrow \textit{nat} \\
\textit{max-list}\ [] = \mathit{0} \\
\textit{max-list}\ (x\ \#\ xs) = \textit{max}\ x\ (\textit{max-list}\ xs)
\end{array}$$

$$\begin{array}{l}
\textit{fresh} :: \textit{fm list} \Rightarrow \textit{nat} \\
\textit{fresh}\ A \equiv \textit{Suc}\ (\textit{max-list}\ (\textit{vars-fms}\ A))
\end{array}$$

Fig. 4. The rules for generating a fresh variable in Isabelle/HOL.

$$\textbf{type-synonym}\ \textit{sequent} = \textit{fm list} \times \textit{fm list}$$

$$\begin{array}{l}
\textit{sc} :: ('a\ \textit{var-denot} \times 'a\ \textit{fun-denot} \times 'a\ \textit{pre-denot}) \Rightarrow \textit{sequent} \Rightarrow \textit{bool} \\
\textit{sc}\ (E,\ F,\ G)\ (A,\ B) = ((\forall\, p\ [\in]\ A.\ [\![E,\ F,\ G]\!]\ p) \longrightarrow (\exists\, q\ [\in]\ B.\ [\![E,\ F,\ G]\!]\ q))
\end{array}$$

Fig. 5. The syntax and semantics of sequents in Isabelle/HOL.

$$\textsc{Idle}\ \frac{A \vdash B}{A \vdash B} \qquad \textsc{Axiom}\ P\ ts\ \frac{}{A \vdash B}\ \text{if}\ \ddagger P\ ts\ [\in]\ A\ \text{and}\ \ddagger P\ ts\ [\in]\ B$$

$$\textsc{FlsL}\ \frac{}{A \vdash B}\ \text{if}\ \bot\ [\in]\ A \qquad \textsc{FlsR}\ \frac{A \vdash B\ [\div]\ \bot}{A \vdash B}\ \text{if}\ \bot\ [\in]\ B$$

$$\textsc{ImpL}\ p\ q\ \frac{A\ [\div]\ (p \longrightarrow q) \vdash p\ \#\ B \qquad q\ \#\ A\ [\div]\ (p \longrightarrow q) \vdash B}{A \vdash B}\ \text{if}\ (p \longrightarrow q)\ [\in]\ A$$

$$\textsc{ImpR}\ p\ q\ \frac{p\ \#\ A \vdash q\ \#\ B\ [\div]\ (p \longrightarrow q)}{A \vdash B}\ \text{if}\ (p \longrightarrow q)\ [\in]\ B$$

$$\textsc{UniL}\ t\ p\ \frac{\langle t \rangle p\ \#\ A \vdash B}{A \vdash B}\ \text{if}\ \forall p\ [\in]\ A$$

$$\textsc{UniR}\ p\ \frac{A \vdash \langle \#\mathit{fresh}(A@B) \rangle p\ \#\ B\ [\div]\ \forall p}{A \vdash B}\ \text{if}\ \forall p\ [\in]\ B$$

Fig. 6. The rules of the sequent calculus presented visually.

The calculus works on two-sided sequents, of type *sequent*, which are represented as pairs of lists of formulas (cf. Fig. 5). We can think of the left-hand side as assumptions and the right-hand side as conclusions. Moreover, the left-hand side is conjunctive, so we can assume all of the formulas there to be true, while the right-hand side is disjunctive, so we only need to prove one.

Sequent calculus has the benefit of the *subformula property*: to prove a formula we only need to look at its subformulas. Contrast this with axiomatic systems using modus ponens (from $p \longrightarrow q$ and p infer q), where we need to guess a suitable "lemma" formula. However, a sequent calculus may still leave too much freedom for comfort. In particular, we want to remove the need for structural rules, since these are too applicable.

Figure 6 lists the underlying rules of the prover in a somewhat idiosyncratic manner. The reason will become apparent later. Each rule has a name to the left of the horizontal line. Below the horizontal line is the conclusion and above are the premises, if any. Any side conditions are given to the right of the line. Note that each rule is indexed by the exact (sub)formulas it works on: the rule Axiom 0 [] is distinct from the rule Axiom 1 [] etc. This rigidity means that we do not need any structural rules. It also means that there is no pattern matching in any of the rules and that the three primary operations are membership checking ($[\in]$), removal of concrete formulas ($[\div]$) and adding new formulas to a list (#).

The Idle rule appears for technical reasons (there should always be an enabled rule). The Axiom rule is indexed by a predicate symbol P and argument list *ts* and checks whether such a predicate appears on both sides of the sequent: if so, the rule applies and there are no child sequents. The FlsL rule checks if $\bot$ occurs among the assumptions, in which case the sequent is proved. The FlsR rule, when it applies, drops all occurrences of $\bot$ from the conclusions, since we can never prove any of them. The ImpL and ImpR rules decompose implications on either side of the sequent in the standard way. The UniL rule is indexed by a term t and a formula p. If $\forall p$ occurs on the left, then the rule instantiates it with t, adding $\langle t\rangle p$ to the left-hand side of the child sequent. The UniR rule is only indexed by a formula p. When $\forall p$ occurs on the right, it is instantiated with a fresh variable and removed.

In order to obtain a prover based on the rules of the sequent calculus we use the abstract completeness framework for Isabelle/HOL developed by Blanchette, Popescu and Traytel [3,5]. This framework formalizes the mechanics of sequent calculus and semantic tableaux provers in an abstract way that we can instantiate with concrete rules. There are two possible perspectives on the framework: (i) the proof perspective, where we use the framework to obtain theorems about proof trees built from our rules and (ii) the code generation perspective, where we use the framework to generate an executable prover. In this paper, both perspectives come into play but the two perspectives can be used on their own.

The framework needs: a stream of rules, a function describing their effect, a proof that some rule is always enabled and a guarantee that rules are persistent. We formalize the calculus in Isabelle/HOL as a datatype of rules, *rule*, with constructors *Idle*, *Axiom*, *FlsL*, *FlsR*, *ImpL*, *ImpR*, *UniL* and *UniR*, and an effect function, *eff*, that encodes the relationship between premises and conclusions in the manner expected by the framework.

5 Soundness and Completeness

Soundness requires that we do not prove a sequent without having proper reasons to do so. It is a local property of our calculus that we can easily check. Completeness, on the other hand, requires that we have sufficient rules available to prove every valid formula. Thus, proving completeness requires a more involved strategy.

Lemma 1 (Local soundness). *If all premises of a rule are valid, then its conclusion is valid. In Isabelle, if eff r* $(A,\ B) = $ *Some ss and* $\forall\, A\ B.\ (A,\ B)$ $|\in|$ *ss* $\longrightarrow (\forall (E :: \text{-} \Rightarrow {'a}).\ sc\ (E,\ F,\ G)\ (A,\ B))$*, then* $sc\ (E,\ F,\ G)\ (A,\ B)$*.*

Proof. By induction on the call structure of *eff*. The induction hypothesis then applies to the sequents produced by *eff*. All cases except UniR are trivial. For UniR, by the induction hypothesis, the premise holds under all variable denotations: no matter the assignment to the fresh variable. This justifies forming the universal quantifier and since the fresh variable does not appear elsewhere in the sequent, the semantics there are unaffected.

Theorem 1 (Prover soundness). *If a proof tree (attempt) is well formed and finite, then the root sequent is valid. In Isabelle, if tfinite t and wf t, then* $sc\ (E,\ F,\ G)\ (fst\ (root\ t))$*.*

Proof. By induction on the *finite* proof tree using Lemma 1.

$$
\begin{array}{l}
\textbf{locale } \textit{Hintikka} = \\
\quad \textbf{fixes } A\ B :: \textit{fm set} \\
\quad \textbf{assumes} \\
\quad\quad \textit{Basic}\text{: } \ddagger P\ ts \in A \Longrightarrow \ddagger P\ ts \in B \Longrightarrow \textit{False} \textbf{ and} \\
\quad\quad \textit{FlsA}\text{: } \bot \notin A \textbf{ and} \\
\quad\quad \textit{ImpA}\text{: } p \longrightarrow q \in A \Longrightarrow p \in B \lor q \in A \textbf{ and} \\
\quad\quad \textit{ImpB}\text{: } p \longrightarrow q \in B \Longrightarrow p \in A \land q \in B \textbf{ and} \\
\quad\quad \textit{UniA}\text{: } \forall\, p \in A \Longrightarrow \forall\, t.\ \langle t \rangle p \in A \textbf{ and} \\
\quad\quad \textit{UniB}\text{: } \forall\, p \in B \Longrightarrow \exists\, t.\ \langle t \rangle p \in B
\end{array}
$$

$$M\ A \equiv [\![\#, \dagger, \lambda P\ ts.\ \ddagger P\ ts \in A]\!]$$

Fig. 7. Formalizations of Hintikka sets and the countermodel $M\ A$.

For completeness we must now show that, for every valid sequent, the prover finds a proof. We do so contrapositively: if the prover does not find a proof, we produce a countermodel to the sequent. To do so, we characterize saturated escape paths syntactically using Hintikka sets and show that such sets induce countermodels. Figure 7 characterizes Hintikka sets in our setting. There are two perspectives on these: one, that they characterize saturated escape paths and two, that they characterize the semantics of the countermodel.

To understand the first perspective, read the set A as consisting of all formulas that appear as assumptions on the saturated escape path (on the left-hand side of sequents) and the set B as consisting of all formulas that appear as conclusions (on the right-hand side of sequents). The Isabelle/HOL functions *treeA* and *treeB* collect these sets, respectively.

Lemma 2 (Hintikka sets characterize saturated escape paths). *Let A and B be sets of assumption and conclusion formulas on a saturated escape path. Then they fulfill all Hintikka requirements. In Isabelle, if epath steps and Saturated steps, then Hintikka (treeA steps) (treeB steps).*

Proof. We check each condition separately.

Basic states that a predicate cannot appear as both assumption and conclusion on the epath. Otherwise the Axiom rule would have terminated the (infinite) epath.

FlsA states that $\bot$ does not appear among the assumptions. Similar to the above, the FlsL rule would have terminated the epath if so.

ImpA and *ImpB* break down implications in accordance with the ImpL and ImpR rules. For a given p, q, if $p \longrightarrow q$ appears in A (respectively B), then at some point in the proof tree attempt, the rule ImpL p q (respectively ImpR p q) becomes enabled. Since the epath is saturated, any enabled rule is eventually taken and the effect matches the thesis.

UniA states that any universally quantified formula $\forall p$ on the left is instantiated with all possible terms. Fix an arbitrary term t. Since $\forall p$ occurs as an assumption, the specific rule UniL p t is eventually enabled, taken, and has the desired effect.

UniB is similar, except the witnessing term is the fresh variable.

Remark 1. We see the usefulness of indexed rules in the above proof. If we simply had an ImpR rule, rather than an ImpR $p\ q$ rule for each formula p and q, we would have to further argue that this rule eventually applies to exactly the implication $p \longrightarrow q$ we need it to. Perhaps we need to argue first that $p \longrightarrow q$ eventually reaches the front of the sequent or similar delicate reasoning. This is where fairness concerns would show up. We have sidestepped the issue by using very specific rules.

Consider now the second perspective. The countermodel in Fig. 7 uses the term universe (also called Herbrand universe) where every variable and function symbol evaluates to itself. Thus, the universal quantifier, which ranges over a given domain, ranges over terms. Now, read the sets A and B as formulas we wish to satisfy and falsify, respectively.

Lemma 3 (A Hintikka set induces a countermodel). *Let A and B be sets of formulas fulfilling the Hintikka requirements. Then $M\ A$ satisfies formulas in A and falsifies formulas in B. In Isabelle, if Hintikka $A\ B$ then $(p \in A \longrightarrow M\ A\ p) \wedge (p \in B \longrightarrow \neg\ M\ A\ p)$.*

Proof. By well founded induction on the size of the formula, such that the induction hypothesis applies to subformulas and instances of universally quantified formulas.

For $\bot \in A$, this contradicts *FlsA* so the thesis holds vacuously. For $\bot \in B$, the thesis holds trivially since $\bot$ is falsified by every model.

For $\dagger P\ ts \in A$, the thesis holds by the definition of *M*. For $\dagger P\ ts \in B$, we cannot have $\dagger P\ ts \in A$ due to *Basic* and so the thesis holds by the definition of *M*.

For $p \longrightarrow q \in A$ and $p \longrightarrow q \in B$ the theses hold by the induction hypotheses at p and q and the conditions *ImpA* and *ImpB*, respectively.

For $\forall p \in A$ and $\forall p \in B$ the theses hold by the induction hypotheses at $\langle t \rangle p$ for all t and by the conditions *UniA* and *UniB*, respectively.

Any saturated escape path induces a countermodel, contradicting validity.

Theorem 2 (Prover completeness). *For any valid sequent, the prover terminates.*

Proof. If the prover does not find a proof, then by the framework, the proof attempt contains a saturated escape path. By Lemma 2, this epath fulfills the Hintikka requirements. By Lemma 3, we can build a model that satisfies every assumption formula and falsifies every conclusion formula. This model contradicts the validity of the sequent.

We join the soundness and completeness theorems in a corollary on formulas.

Corollary 1. *The prover terminates if, and only if, the given formula is valid. In Isabelle, fix p :: fm and let $t \equiv$ prover $([], [p])$, then tfinite $t \wedge$ wf $t \longleftrightarrow (\forall\, (E :: \text{-} \Rightarrow tm)\ F\ G.\ [\![E,\ F,\ G]\!]\ p)$.*

References

1. Ben-Ari, M.: Mathematical Logic for Computer Science. Springer, Cham (2012). https://doi.org/10.1007/978-1-4471-4129-7
2. Blanchette, J.C.: Formalizing the metatheory of logical calculi and automatic provers in Isabelle/HOL (invited talk). In: Mahboubi, A., Myreen, M.O. (eds.) Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, pp. 1–13. ACM (2019). https://doi.org/10.1145/3293880.3294087
3. Blanchette, J.C., Popescu, A., Traytel, D.: Abstract completeness. Archive of Formal Proofs (2014). https://isa-afp.org/entries/Abstract_Completeness.html. Formal proof development
4. Blanchette, J.C., Popescu, A., Traytel, D.: Unified classical logic completeness. In: Demri, S., Kapur, D., Weidenbach, C. (eds.) IJCAR 2014. LNCS (LNAI), vol. 8562, pp. 46–60. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08587-6_4
5. Blanchette, J.C., Popescu, A., Traytel, D.: Soundness and completeness proofs by coinductive methods. J. Autom. Reason. **58**(1), 149–179 (2016). https://doi.org/10.1007/s10817-016-9391-3
6. de Bruijn, N.: Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. In: Nederpelt, R., Geuvers, J., de Vrijer, R. (eds.) Selected Papers on Automath, Studies in Logic and the Foundations of Mathematics, vol. 133, pp. 375–388. Elsevier (1994). https://doi.org/10.1016/S0049-237X(08)70216-7, reprinted from: Indagationes Math, 34, 5, pp. 381–392, by courtesy of the Koninklijke Nederlandse Akademie van Wetenschappen, Amsterdam
7. From, A.H.: Synthetic completeness for a terminating Seligman-style tableau system. In: de'Liguoro, U., Berardi, S., Altenkirch, T. (eds.) 26th International Conference on Types for Proofs and Programs, TYPES 2020, University of Turin, Italy, 2–5 March 2020. LIPIcs, vol. 188, pp. 5:1–5:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020). https://doi.org/10.4230/LIPIcs.TYPES.2020.5
8. From, A.H.: Formalized soundness and completeness of epistemic logic. In: Silva, A., Wassermann, R., de Queiroz, R.J.G.B. (eds.) WoLLIC 2021. LNCS, vol. 13038, pp. 1–15. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-88853-4_1
9. From, A.H.: A succinct formalization of the completeness of first-order logic. In: Basold, H., Cockx, J., Ghilezan, S. (eds.) 27th International Conference on Types for Proofs and Programs, TYPES 2021, Leiden, The Netherlands, 14–18 June 2021 (Virtual Conference). LIPIcs, vol. 239, pp. 8:1–8:24. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021). https://doi.org/10.4230/LIPIcs.TYPES.2021.8
10. From, A.H., Jacobsen, F.K.: Verifying a sequent calculus prover for first-order logic with functions in Isabelle/HOL. In: Andronick, J., de Moura, L. (eds.) 13th International Conference on Interactive Theorem Proving, ITP 2022, Haifa, Israel, 7–10 August 2022. LIPIcs, vol. 237, pp. 13:1–13:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022). https://doi.org/10.4230/LIPIcs.ITP.2022.13
11. From, A.H.: A Naive prover for first-order logic. Archive of Formal Proofs (2022). https://isa-afp.org/entries/FOL_Seq_Calc3.html, Formal proof development
12. From, A.H., Jacobsen, F.K.: A sequent calculus prover for first-order logic with functions. Archive of Formal Proofs (2022). https://isa-afp.org/entries/FOL_Seq_Calc2.html, Formal proof development
13. From, A.H., Jensen, A.B., Schlichtkrull, A., Villadsen, J.: Teaching a formalized logical calculus. Electron. Proc. Theor. Comput. Sci. **313**, 73–92 (2020). https://doi.org/10.4204/EPTCS.313.5

14. Gödel, K.: Die Vollständigkeit der Axiome des logischen Funktionenkalküls. Monatshefte für Mathematik und Physik **37**(1), 349–360 (1930). https://doi.org/10.1007/BF01696781
15. Henkin, L.: The discovery of my completeness proofs. Bull. Symb. Log. **2**(2), 127–158 (1996). https://doi.org/10.2307/421107
16. Jensen, A.B., Larsen, J.B., Schlichtkrull, A., Villadsen, J.: Programming and verifying a declarative first-order prover in Isabelle/HOL. AI Commun. Eur. J. Artif. Intell. **31**(3), 281–299 (2018). https://doi.org/10.3233/AIC-180764
17. Kleene, S.C.: Mathematical Logic. Courier Corporation (2002)
18. Michaelis, J., Nipkow, T.: Formalized proof systems for propositional logic. In: Abel, A., Forsberg, F.N., Kaposi, A. (eds.) 23rd International Conference on Types for Proofs and Programs (TYPES 2017). Leibniz International Proceedings in Informatics (LIPIcs), vol. 104, pp. 5:1–5:16. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2018). https://doi.org/10.4230/LIPIcs.TYPES.2017.5
19. Nipkow, T., Klein, G.: Concrete Semantics - With Isabelle/HOL. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10542-0
20. Pastre, D.: Muscadet 2.3: a knowledge-based theorem prover based on natural deduction. In: Goré, R., Leitsch, A., Nipkow, T. (eds.) IJCAR 2001. LNCS, vol. 2083, pp. 685–689. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45744-5_56
21. Pelletier, F.J.: Automated natural deduction in THINKER. Stud. Logica. **60**(1), 3–43 (1998). https://doi.org/10.1023/A:1005035316026
22. Ridge, T., Margetson, J.: A mechanically verified, sound and complete theorem prover for first order logic. In: Hurd, J., Melham, T. (eds.) TPHOLs 2005. LNCS, vol. 3603, pp. 294–309. Springer, Heidelberg (2005). https://doi.org/10.1007/11541868_19
23. Schlichtkrull, A., Blanchette, J.C., Traytel, D.: A verified prover based on ordered resolution. In: Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, pp. 152–165. Association for Computing Machinery, New York (2019). https://doi.org/10.1145/3293880.3294100
24. Schulz, S., Pease, A.: Teaching automated theorem proving by example: PyRes 1.2. In: Peltier, N., Sofronie-Stokkermans, V. (eds.) IJCAR 2020. LNCS (LNAI), vol. 12167, pp. 158–166. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-51054-1_9
25. Villadsen, J., Schlichtkrull, A., From, A.H.: A verified simple prover for first-order logic. In: Konev, B., Urban, J., Rümmer, P. (eds.) Proceedings of the 6th Workshop on Practical Aspects of Automated Reasoning. CEUR Workshop Proceedings, vol. 2162, pp. 88–104. CEUR-WS.org (2018). https://ceur-ws.org/Vol-2162/paper-08.pdf