A Micro Prover for Teaching Automated Reasoning

Villadsen, Jørgen

Publication date:
2020

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

Link back to DTU Orbit

Citation (APA):
Villadsen, J. (2020). *A Micro Prover for Teaching Automated Reasoning*. Paper presented at 7th Workshop on Practical Aspects of Automated Reasoning, Paris, France.

A Micro Prover for Teaching Automated Reasoning

Jørgen Villadsen

Technical University of Denmark

Abstract

We present a simple prover for classical propositional logic. The prover is based on the sequent calculus and is formally verified in the Isabelle/HOL proof assistant. We use the prover for teaching automated reasoning to computer science students. The micro prover is available online and is simple enough to be the first example in a course. It shows how to use Isabelle/HOL and it also shows a prover program with termination, soundness and completeness proofs.

1 Introduction

In an invited talk last year, at the Certified Programs and Proofs (CPP) conference co-located with POPL, Jasmin Blanchette made the following observation [2]:

> *At programming language conferences such as POPL and ICFP, submissions are often accompanied by formalizations... Paradoxically, the automated reasoning community has largely stood on the sidelines of these developments. Like the shoemaker's children who go barefoot, we reflexively turn to "pen and paper" – by which we usually mean LaTeX – to define our logics, specify our proof systems, and establish their soundness and completeness.*

The proof assistant Isabelle/HOL [13] is one of the most used proof assistants and in 2015 the IsaFoL (Isabelle Formalization of Logic) effort started (`https://bitbucket.org/isafol`). The goal is to develop lemma libraries and methodology for formalizing modern research in automated reasoning and for supporting modern teaching in automated reasoning too. More than 20 researchers have contributed so far and most of the formalizations are available in the Archive of Formal Proofs (`https://isa-afp.org/`) and are therefore assured updates for future releases of Isabelle.

These formalizations of logic provide specifications of the syntax, semantics and various proof systems with formally verified soundness and preferably also completeness theorems. The state-of-the-art is a series of formalizations of resolution for first-order logic [16, 18, 17]. While such formalizations are indeed very valuable for researchers and PhD students, they are in our opinion not at the moment suitable for teaching logic to bachelor and master students. The amount of details is simply too large.

Since 2015 we have rather successfully used simpler formalizations of logic for bachelor and master computer science students. In particular, our Natural Deduction Assistant (NaDeA) [22, 21] has been used by more than 300 students in the BSc course "Logical Systems and Logic Programming" (`https://kurser.dtu.dk/course/02156`). NaDeA allows students to develop and test their natural deduction skills interactively for first-order logic. NaDeA is open source, runs in a standard browser and has a log-in system called ProofJudge for students to upload proofs and teaching assistants to assess them. However, the salient point of NaDeA is the integration with Isabelle/HOL — upon completion of a proof in NaDeA the student obtains a formal proof in Isabelle/HOL from the specification of the syntax, semantics and natural deduction proof system — this formal proof relies only on the formalization of first-order logic in Isabelle/HOL, replaying the proof so to speak.

For selected students we have created special courses and projects based on the following more advanced formalizations of logic:

- Declarative Prover / Students' Proof Assistant (SPA) [10, 19]
- Simple Prover (Negation Normal Form) [15, 24]

Both provide formally verified provers for first-order logic. The former has a formal soundness proof and allows for declarative proofs as described in John Harrison's Handbook of Practical Logic and Automated Reasoning, Cambridge University Press, 2009. The latter has a formal soundness proof as well as a formal completeness proof for a fixed proof method. Both provers can either be executed in Isabelle/HOL using code reflection or the code generator can be used to obtain source code for several programming languages. So while NaDeA is an interactive prover with very little automation and the "declarative prover" is an interactive prover with quite some automation, the "simple prover" is fully automatic.

However, after 5 years we have reached the conclusion that NaDeA and the other provers are a bit too complicated for the bachelor course. Furthermore, we have a new MSc course "Automated Reasoning" (`https://kurser.dtu.dk/course/02256`) where NaDeA and the other provers can be better utilized. For the new MSc course "Automated Reasoning" we have also developed a formally verified prover for propositional logic that is much simpler, essentially 7 lines with rewriting rules, for which we have formal proofs in Isabelle/HOL ensuring:

- Termination
- Soundness
- Completeness

The starting point of our formalization has been the recent work by Julius Michaelis and Tobias Nipkow on proof systems for propositional logic [12]. We use a sequent calculus and our aim is to keep it as simple as possible.

In Section 2 we consider related work. In Section 3 we give a quick recap of the sequent calculus. In Section 4 we provide an overview of the prover. In Section 5 we describe the termination proof. In Section 6 we describe a number of examples. In Section 7 we describe the proof of soundness and completeness. In Section 8 we briefly look at code generation (OCaml). Finally Section 9 provides the conclusion.

The micro prover is available online here (149 lines):

`https://bitbucket.org/isafol/isafol/src/master/Sequent_Calculus/Micro_Prover.thy`

The Isabelle/HOL document is checked in a few seconds. On a fast computer with many cores it takes less than a second for the whole document.

2 Related Work

We have already mentioned in Section 1 that we have used formalizations of logic like NaDeA in teaching logic [21, 19, 7, 24, 23]. The Incredible Proof Machine by Joachim Breitner [6] has also been used for teaching but as far as we know the formalization in Isabelle/HOL is not used in the actual teaching and similar observations hold for other contributions, in particular at the venues dedicated to the topic, such as Tools for Teaching Logic (TTL) and Theorem proving components for Educational software (ThEdu). The starting point of our formalization [12] has also not been used for teaching logic and is not immediately understandable for most bachelor and master students. Small provers like leanTaP and leanCoP [14] are difficult to understand for beginners (but their performance is in general stronger).

In Section 1 we have also mentioned some state-of-the-art formalizations of logic [16, 17, 18]. Other advanced formalizations of first-order logic [3, 4, 5, 9, 20] and even higher-order logic [11, 8] are available but are much more involved than NaDeA.

3 Prerequisites: Sequent Calculus

For the course on automated reasoning we do not assume that the students have a strong background in logic and computer science. We start with a brief description of the sequent calculus for classical propositional logic.

We use Moti Ben-Ari's textbook as a starting point [1]. Formulas A, B, ... in the classical propositional logic are generated from falsity ($\bot$), propositional symbols (p, q, ...) and implications ($\rightarrow$). The relevant sequent calculus axioms and rules are as follows.

The axioms of the system are of the form

$$U \cup \{A\} \vdash V \cup \{A\}$$

or

$$U \cup \{\bot\} \vdash V$$

where U and V are sets of formulas.

The rules of the system are left and right introduction rules:

$$\frac{U \vdash V \cup \{A\} \qquad U \cup \{B\} \vdash V}{U \cup \{A \rightarrow B\} \vdash V} \qquad \frac{U \cup \{A\} \vdash V \cup \{B\}}{U \vdash V \cup \{A \rightarrow B\}}$$

Whereas the right introduction rule is quite straightforward we find the left introduction rule more difficult to explain to students. The best approach seems to be to consider the case where $A \rightarrow B$ is false and what it means classically for A and B.

4 Overview of the Prover

We now provide an overview of the prover. We use the following definition:

> Formulas are either $\bot$ (falsity), a propositional symbol (a natural number) or an implication ($p \rightarrow q$).

When programming in Isabelle/HOL it is not a problem to use logical/mathematical symbols like $\bot$ and $\rightarrow$ and in contrast to most programming languages the natural numbers are available as a datatype.

Two comments about programming in Isabelle/HOL using functions. Firstly, an underscore _ is used when the argument is irrelevant (wildcard). Secondly, we prefer to use the `if ... then ... else` expression instead of the implication operator of Isabelle/HOL in order to make it easier to understand for (computer science) students just starting to learn logic.

We define the primitive recursive functions `member` and `common` as follows.

```
member _ [] = False
member m (n # A) = (if m = n then True else member m A)

common _ [] = False
common A (m # B) = (if member m A then True else common A B)
```

In Isabelle/HOL lists are constructed from head and tail using the # operator. The functions are quite easy to understand for computer science students, in particular if they know functional and/or logic programming. But it is important that the students have a very good understanding of recursion and we start with the primitive recursive functions `member` and `common` in order to make sure that all students understand the list datatype, termination and correctness with formal proofs:

lemma *member_set*: ‹*member m A* ⟷ *m* ∈ *set A*›
  by (*induct A*) *simp_all*

lemma *common_set*: ‹*common A B* ⟷ *set A* ∩ *set B* ≠ {}›
  by (*induct B*) (*simp_all add*: *member_set*)

At this point the students cannot fully understand the above proofs in Isabelle/HOL but it is important to point out from the start the possibility of mixing proving and programming.

We then define a (non-primitive) recursive function μ as follows.

```
μ A B (Pro n # C) [] = μ (n # A) B C []
μ A B C (Pro n # D) = μ A (n # B) C D
μ _ _ (⊥ # _) [] = True
μ A B C (⊥ # D) = μ A B C D
μ A B ((p → q) # C) [] = (if μ A B C [p] then μ A B (q # C) [] else False)
μ A B C ((p → q) # D) = μ A B (p # C) (q # D)
μ A B [] [] = common A B
```

The first and second arguments are the basic propositions from the left and right sides of the sequents and the third and fourth arguments are the left and right sides of the sequents (with formulas not yet processed). The programming aspects of the individual lines in the prover are not too difficult to understand but even with a firm grasp of the sequent calculus it is by no means obvious to the students that the above program is correct. In the following sections we present the formal proof in Isabelle/HOL and also explain the details of the micro prover.

5 Termination

We start the Isabelle/HOL theory with the datatype for formulas.

theory *Micro_Prover* **imports** *Main* **begin**

datatype *form* = *Pro nat* | *Falsity* (‹⊥›) | *Imp form form* (**infix** ‹→› *0*)

For simplicity in the definition we do not make the implication operator right associative and we do not introduce any precedences.

We then define the functions for the micro prover. The primitive recursive functions do not need any termination proof.

primrec *member* **where**
⟨*member* _ [] = *False*⟩ |
⟨*member m* (*n* # *A*) = (*if m* = *n then True else member m A*)⟩

primrec *common* **where**
⟨*common* _ [] = *False*⟩ |
⟨*common A* (*m* # *B*) = (*if member m A then True else common A B*)⟩

function μ **where**
⟨μ *A B* (*Pro n* # *C*) [] = μ (*n* # *A*) *B C* []⟩ |
⟨μ *A B C* (*Pro n* # *D*) = μ *A* (*n* # *B*) *C D*⟩ |
⟨μ _ _ ($\bot$ # _) [] = *True*⟩ |
⟨μ *A B C* ($\bot$ # *D*) = μ *A B C D*⟩ |
⟨μ *A B* (($p \to q$) # *C*) [] = (*if* μ *A B C* [*p*] *then* μ *A B* (*q* # *C*) [] *else False*)⟩ |
⟨μ *A B C* (($p \to q$) # *D*) = μ *A B* (*p* # *C*) (*q* # *D*)⟩ |
⟨μ *A B* [] [] = *common A B*⟩
by *pat_completeness simp_all*

We prove that the argument patterns are complete using the *pat_completeness simp_all* formal proof. This is more or less the default approach. Finally termination is proved using *simp_all* with an appropriate measure using the sizes of the lists of formulas and the sizes of the individual formulas.

termination by (*relation* ⟨*measure* (λ(_,_,*C*,*D*). *size* (*C* @ *D*) + *2*∗($\sum$ *p* ← *C* @ *D*. *size p*))⟩) *simp_all*

6 Examples

We show six examples, first as ordinary propositions in Isabelle/HOL and then as proofs using the micro prover.

proposition ⟨(($p \longrightarrow$ *False*) $\longrightarrow$ *False*) $\longrightarrow p$⟩ **by** *fast*

theorem ⟨μ [] [] [] [((*Pro 0* $\to \bot$) $\to \bot$) $\to$ *Pro 0*]⟩ **by** *eval*

proposition ⟨$p \longrightarrow p$⟩ **by** *fast*

theorem ⟨μ [] [] [] [*Pro 0* $\to$ *Pro 0*]⟩ **by** *eval*

proposition ⟨$p \longrightarrow q \longrightarrow p$⟩ **by** *fast*

theorem ⟨μ [] [] [] [*Pro 0* $\to$ (*Pro 1* $\to$ *Pro 0*)]⟩ **by** *eval*

proposition ⟨($p \longrightarrow q \longrightarrow r$) $\longrightarrow$ ($p \longrightarrow q$) $\longrightarrow p \longrightarrow r$⟩ **by** *fast*

theorem ⟨μ [] [] [] [(*Pro 0* $\to$ (*Pro 1* $\to$ *Pro 2*)) $\to$ ((*Pro 0* $\to$ *Pro 1*) $\to$ (*Pro 0* $\to$ *Pro 2*))]⟩ **by** *eval*

proposition $\langle p \longrightarrow q \longrightarrow q \longrightarrow p\rangle$ **by** *fast*

theorem $\langle \mu\ []\ []\ []\ [Pro\ 0 \rightarrow (Pro\ 1 \rightarrow (Pro\ 1 \rightarrow Pro\ 0))]\rangle$ **by** *eval*

proposition $\langle p \longrightarrow (p \longrightarrow q) \longrightarrow q\rangle$ **by** *fast*

theorem $\langle \mu\ []\ []\ []\ [Pro\ 0 \rightarrow ((Pro\ 0 \rightarrow Pro\ 1) \rightarrow Pro\ 1)]\rangle$ **by** *eval*

7 Soundness and Completeness

As preliminaries we define another prover that returns a list of basic proposition to be used for building counter-examples.

function μ' **where**
$\langle \mu'\ A\ B\ (Pro\ n\ \#\ C)\ [] = \mu'\ (n\ \#\ A)\ B\ C\ []\rangle\ |$
$\langle \mu'\ A\ B\ C\ (Pro\ n\ \#\ D) = \mu'\ A\ (n\ \#\ B)\ C\ D\rangle\ |$
$\langle \mu'\ \_\ \_\ (\bot\ \#\ \_)\ [] = []\rangle\ |$
$\langle \mu'\ A\ B\ C\ (\bot\ \#\ D) = \mu'\ A\ B\ C\ D\rangle\ |$
$\langle \mu'\ A\ B\ ((p \rightarrow q)\ \#\ C)\ [] = \mu'\ A\ B\ C\ [p]\ @\ \mu'\ A\ B\ (q\ \#\ C)\ []\rangle\ |$
$\langle \mu'\ A\ B\ C\ ((p \rightarrow q)\ \#\ D) = \mu'\ A\ B\ (p\ \#\ C)\ (q\ \#\ D)\rangle\ |$
$\langle \mu'\ A\ B\ []\ [] = (\mathit{if}\ set\ A \cap set\ B = \{\}\ then\ [A]\ else\ [])\rangle$
by *pat_completeness simp_all*

termination by $(relation\ \langle measure\ (\lambda(\_,\_,C,D).\ size\ (C\ @\ D) + 2*(\sum p \leftarrow C\ @\ D.\ size\ p))\rangle)$ *simp_all*

The new prover is proved to work as the original prover.

lemma *member_set*: $\langle member\ m\ A \longleftrightarrow m \in set\ A\rangle$
by (*induct A*) *simp_all*

lemma *common_set*: $\langle common\ A\ B \longleftrightarrow set\ A \cap set\ B \neq \{\}\rangle$
by (*induct B*) (*simp_all add*: *member_set*)

lemma *micro*: $\langle \mu\ A\ B\ C\ D \longleftrightarrow \mu'\ A\ B\ C\ D = []\rangle$
by (*induct rule*: *μ.induct*) (*simp_all add*: *common_set*)

We then define a primitive recursive function *semantics* as follows. The first argument is the interpretation, namely a function from propositional symbols (natural numbers) to truth values (*True*/*False*).

primrec *semantics* **where**
$\langle semantics\ i\ (Pro\ n) = i\ n\rangle\ |$
$\langle semantics\ \_\ \bot = False\rangle\ |$
$\langle semantics\ i\ (p \rightarrow q) = (\mathit{if}\ semantics\ i\ p\ then\ semantics\ i\ q\ else\ True)\rangle$

The semantics is extended to sequents.

abbreviation $\langle semantics'\ i\ X\ Y \equiv (\forall p \in set\ X.\ semantics\ i\ p) \longrightarrow (\exists p \in set\ Y.\ semantics\ i\ p)\rangle$

```
inductive SC (‹_ ≫ _› 0) where
  Fls_L: ‹⊥ # _ ≫ _› |
  Fls_R: ‹X ≫ ⊥ # Y› if ‹X ≫ Y› |
  Imp_L: ‹(p → q) # X ≫ Y› if ‹X ≫ p # Y› and ‹q # X ≫ Y› |
  Imp_R: ‹X ≫ (p → q) # Y› if ‹p # X ≫ q # Y› |
  Set_L: ‹X' ≫ Y› if ‹X ≫ Y› and ‹set X' = set X› |
  Set_R: ‹X ≫ Y'› if ‹X ≫ Y› and ‹set Y' = set Y› |
  Basic: ‹p # _ ≫ p # _›
```

The sequent calculus *SC* is proved sound using the automation of Isabelle/HOL.

```
lemma proper: ‹X ≫ Y ⟹ semantics' i X Y›
  by (induct rule: SC.induct) auto
```

The new prover is proved complete using counter-examples and really using the automation of Isabelle/HOL.

```
lemma cex: ‹L ∈ set (μ' A B C D) ⟹ ¬ semantics' (λn. n ∈ set L) (map Pro A @ C) (map Pro B @ D)›
  by (induct A B C D rule: μ'.induct) auto
```

The following lemma is used in the soundness proof to follow. We find that the proof is an excellent example of the advanced natural deduction proof style available in Isabelle/HOL (again combined with automation).

```
lemma base: ‹set A ∩ set B ≠ {} ⟹ map Pro A ≫ map Pro B›
proof −
  assume ‹set A ∩ set B ≠ {}›
  then obtain n A' B' where ‹set (n # A') = set A› ‹set (n # B') = set B›
    by auto
  moreover have ‹map Pro (n # A') ≫ map Pro (n # B')›
    using Basic by simp
  ultimately show ?thesis
    using Set_L Set_R set_map by metis
qed
```

The new prover is also proved sound. Here the proof by induction needs help in three cases (number 3, 5 and 6).

```
lemma just: ‹μ' A B C D = [] ⟹ map Pro A @ C ≫ map Pro B @ D›
proof (induct A B C D rule: μ'.induct)
  case (3 A B C)
  have ‹⊥ # map Pro A @ C ≫ map Pro B›
    using Fls_L by simp
  then show ?case
    using Set_L by simp
next
  case (5 A B p q C)
  then have ∗: ‹map Pro A @ C ≫ map Pro B @ [p]› ‹map Pro A @ q # C ≫ map Pro B›
    by simp_all
  have ‹map Pro A @ C ≫ p # map Pro B› ‹q # map Pro A @ C ≫ map Pro B›
    by (use ∗ Set_R in simp) (use ∗ Set_L in simp)
  then show ?case
    using Imp_L Set_L by fastforce
```

next
  case (*6 A B C p q D*)
  then have ⟨*map Pro A* @ *p* # *C* ≫ *map Pro B* @ *q* # *D*⟩
    by *simp*
  then have ⟨*p* # *map Pro A* @ *C* ≫ *q* # *map Pro B* @ *D*⟩
    using *Set_L Set_R Un_insert_right list.set*(*2*) *set_append* **by** *metis*
  then show *?case*
    using *Imp_R Set_R* **by** *fastforce*
qed (*auto simp*: *base intro*: *SC.intros split*: *if_splits*)

Finally we prove the soundness and completeness of the micro prover.

theorem *main*: ⟨μ [] [] [] [*p*] ⟷ (∀ *i*. *semantics i p*)⟩
proof −
  have ⟨μ [] [] [] [*p*] ⟹ *semantics i p*⟩ **for** *i p*
    using *just micro proper* **by** *fastforce*
  then show *?thesis*
    using *cex micro list.set_intros*(*1*) *neq_Nil_conv set_append Un_iff* **by** *metis*
qed

end

8 Code Generation (OCaml)

In Isabelle/HOL it is possible to generate code for Standard ML, Haskell, OCaml and Scala. ML and OCaml are arguably the closest to programming in Isabelle/HOL and while ML is entirely integrated in Isabelle it is overall a good experience for students to have a look at the code generated for OCaml because nowadays OCaml is a widely used general-purpose industrial-strength programming language with an emphasis on expressiveness and safety.

The following OCaml program is obtained in Isabelle/HOL as `export_code test in OCaml` and can be tested here:

https://try.ocamlpro.com/

The program includes a simple test of trying to prove falsity and hence the result is simply `- : bool = false` for the call `Micro_Prover.test` after the following 57 lines:

```
module HOL : sig
  type 'a equal = {equal : 'a -> 'a -> bool}
  val equal : 'a equal -> 'a -> 'a -> bool
  val eq : 'a equal -> 'a -> 'a -> bool
end = struct

type 'a equal = {equal : 'a -> 'a -> bool};;
let equal _A = _A.equal;;

let rec eq _A a b = equal _A a b;;

end;; (*struct HOL*)
```

```
module Arith : sig
  type nat
  val equal_nat : nat HOL.equal
end = struct

type nat = Zero_nat | Suc of nat;;

let rec equal_nata x0 x1 = match x0, x1 with Zero_nat, Suc x2 -> false
                     | Suc x2, Zero_nat -> false
                     | Suc x2, Suc y2 -> equal_nata x2 y2
                     | Zero_nat, Zero_nat -> true;;

let equal_nat = ({HOL.equal = equal_nata} : nat HOL.equal);;

end;; (*struct Arith*)


module Micro_Prover : sig
  type form
  val test : bool
end = struct

type form = Pro of Arith.nat | Falsity | Imp of form * form;;

let rec member _A
  uu x1 = match uu, x1 with uu, [] -> false
    | m, n :: a -> (if HOL.eq _A m n then true else member _A m a);;

let rec common _A
  uu x1 = match uu, x1 with uu, [] -> false
    | a, m :: b -> (if member _A m a then true else common _A a b);;

let rec mu
  a b c x3 = match a, b, c, x3 with a, b, Pro n :: c, [] -> mu (n :: a) b c []
    | a, b, c, Pro n :: d -> mu a (n :: b) c d
    | uu, uv, Falsity :: uw, [] -> true
    | a, b, c, Falsity :: d -> mu a b c d
    | a, b, Imp (p, q) :: c, [] ->
        (if mu a b c [p] then mu a b (q :: c) [] else false)
    | a, b, c, Imp (p, q) :: d -> mu a b (p :: c) (q :: d)
    | a, b, [], [] -> common Arith.equal_nat a b;;

let test : bool = mu [] [] [] [Falsity];;

end;; (*struct Micro_Prover*)
```

9 Conclusion and Future Work

We have used Julius Michaelis and Tobias Nipkow's formalization [12] as a starting point but have changed both the overall structure and the details of the programs in many ways such that the termination, soundness and completeness proofs are easier to understand. The formalization has recently been successfully used in a new computer science course on automated reasoning at DTU (40 students):

https://kurser.dtu.dk/course/02256

The present approach to teaching logic is to our knowledge unique and will be further tested as soon as possible. As future work we in particular consider developing more teaching materials for the micro prover.

We include a supplement with a revised micro prover but without the sequent calculus *SC*. We prove soundness and completeness directly from the semantics. The file has just 36 lines including blank lines — or 25 lines of code.

Acknowledgements

We thank Asta Halkjær From, Alexander Birch Jensen and Anders Schlichtkrull for discussions.

References

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

[2] Jasmin Christian Blanchette. Formalizing the metatheory of logical calculi and automatic provers in Isabelle/HOL (invited talk). In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP)*, pages 1–13, 2019.

[3] Jasmin Christian Blanchette, Andrei Popescu, and Dmitriy Traytel. Soundness and completeness proofs by coinductive methods. *Journal of Automated Reasoning*, 58(1):149–179, 2017.

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

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

[6] Joachim Breitner. Visual theorem proving with the incredible proof machine. In *ITP*, volume 9807 of *Lecture Notes in Computer Science*, pages 123–139. Springer, 2016.

[7] Asta Halkjær From, Alexander Birch Jensen, Anders Schlichtkrull, and Jørgen Villadsen. Teaching a Formalized Logical Calculus. In *Proceedings of the 8th International Workshop on Theorem proving components for Educational software (ThEdu'19)*, 2020.

[8] John Harrison. Towards self-verification of HOL Light. In Ulrich Furbach and Natarajan Shankar, editors, *IJCAR 2006*, volume 4130 of *LNCS*, pages 177–191. Springer, 2006.

[9] Hugo Herbelin, Sun Young Kim, and Gyesik Lee. Formalizing the meta-theory of first-order predicate logic. *Journal of the Korean Mathematical Society*, 54(5):1521–1536, September 2017.

[10] 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.

[11] Ramana Kumar, Rob Arthan, Magnus O. Myreen, and Scott Owens. Self-formalisation of higher-order logic: Semantics, soundness, and a verified implementation. *Journal of Automated Reasoning*, 56(3):221–259, 2016.

[12] Julius Michaelis and Tobias Nipkow. Formalized proof systems for propositional logic. In *TYPES*, volume 104 of *LIPIcs*, pages 5:1–5:16. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017.

[13] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[14] Jens Otten and Wolfgang Bibel. leanCoP: lean connection-based theorem proving. *Journal of Symbolic Computation*, 36:139–161, 2003.

[15] Tom Ridge and James Margetson. A mechanically verified, sound and complete theorem prover for first order logic. In *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK, August 22-25, 2005, Proceedings*, pages 294–309, 2005.

[16] Anders Schlichtkrull. Formalization of the resolution calculus for first-order logic. *Journal of Automated Reasoning*, 61(1):455–484, 2018.

[17] Anders Schlichtkrull, Jasmin Christian Blanchette, and Dmitriy Traytel. A verified prover based on ordered resolution. In *CPP*, pages 152–165. ACM, 2019.

[18] 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. Springer, 2018.

[19] Anders Schlichtkrull, Jørgen Villadsen, and Andreas Halkjær From. Students' Proof Assistant (SPA). In Pedro Quaresma and Walther Neuper, editors, *Proceedings 7th International Workshop on Theorem proving components for Educational Software (ThEdu)*, volume 290 of *EPTCS*, pages 1–13, 2019.

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

[21] Jørgen Villadsen, Andreas Halkjær From, and Anders Schlichtkrull. Natural Deduction Assistant (NaDeA). In Pedro Quaresma and Walther Neuper, editors, *Proceedings 7th International Workshop on Theorem proving components for Educational Software (ThEdu)*, volume 290 of *EPTCS*, pages 14–29, 2019.

[22] Jørgen Villadsen, Asta Halkjær From, Alexander Birch Jensen, and Anders Schlichtkrull. NaDeA: A natural deduction assistant with a formalization in Isabelle. Website with interactive theorem prover: `https://nadea.compute.dtu.dk`.

[23] Jørgen Villadsen, Anders Schlichtkrull, and Andreas Halkjær From. SimPro - Simple Prover - With a Formalization in Isabelle. `https://github.com/logic-tools/simpro`.

[24] Jørgen Villadsen, Anders Schlichtkrull, and Andreas Halkjær From. A verified simple prover for first-order logic. In Boris Konev, Josef Urban, and Philipp Rümmer, editors, *6th Workshop on Practical Aspects of Automated Reasoning (PAAR)*, number 2162 in CEUR Workshop Proceedings, pages 88–104, Aachen, 2018.

Supplement: A Revised Micro Prover

```
theory Prover imports Main begin

datatype 'a form = Pro 'a | Falsity (‹⊥›) | Imp ‹'a form› ‹'a form› (infix ‹→› 0)

primrec semantics where
  ‹semantics i (Pro n) = i n› |
  ‹semantics _ ⊥ = False› |
  ‹semantics i (p → q) = (semantics i p ⟶ semantics i q)›

abbreviation ‹sc X Y i ≡ (∀ p ∈ set X. semantics i p) ⟶ (∃ q ∈ set Y. semantics i q)›

function μ where
  ‹μ A B (Pro n # C) [] = μ (n # A) B C []› |
  ‹μ A B C (Pro n # D) = μ A (n # B) C D› |
  ‹μ _ _ (⊥ # _) [] = {}› |
  ‹μ A B C (⊥ # D) = μ A B C D› |
  ‹μ A B ((p → q) # C) [] = μ A B C [p] ∪ μ A B (q # C) []› |
  ‹μ A B C ((p → q) # D) = μ A B (p # C) (q # D)› |
  ‹μ A B [] [] = (if set A ∩ set B = {} then {A} else {})›
  by pat_completeness simp_all

termination by (relation ‹measure (λ(_,_,C,D). ∑ p ← C @ D. size p)›) simp_all

lemma sat: ‹sc (map Pro A @ C) (map Pro B @ D) (λn. n ∈ set L) ⟹ L ∉ μ A B C D›
  by (induct rule: μ.induct) auto

theorem main: ‹(∀ i. sc (map Pro A @ C) (map Pro B @ D) i) ⟷ μ A B C D = {}›
  by (induct rule: μ.induct) (auto simp: sat)

definition ‹prover p ≡ μ [] [] [] [p] = {}›

corollary ‹prover p ⟷ (∀ i. semantics i p)›
  unfolding prover_def by (simp flip: main)

end
```

Source:

https://bitbucket.org/isafol/isafol/src/master/Sequent_Calculus/Prover.thy