Programming and Verifying a Declarative First-Order Prover in Isabelle/HOL

Jensen, Alexander Birch; Larsen, John Bruntse; Schlichtkrull, Anders; Villadsen, Jørgen

Published in:
AI Communications

Link to article, DOI:
10.3233/AIC-180764

Publication date:
2018

Document Version
Peer reviewed version

AI Communications Special Issue on Automated Reasoning (2018) 1–20
IOS Press

Programming and Verifying a Declarative First-Order Prover in Isabelle/HOL

Alexander Birch Jensen [a], John Bruntse Larsen [a], Anders Schlichtkrull [a] and Jørgen Villadsen [a,*]
[a] *DTU Compute, Technical University of Denmark, 2800 Kongens Lyngby, Denmark*
E-mails: aleje@dtu.dk, jobla@dtu.dk, andschl@dtu.dk, jovi@dtu.dk

Abstract. We certify in the proof assistant Isabelle/HOL the soundness of a declarative first-order prover with equality. The LCF-style prover is a translation we have made, to Standard ML, of a prover in John Harrison's *Handbook of Practical Logic and Automated Reasoning*. We certify it by replacing its kernel with a certified version that we program, certify and generate code from; all in Isabelle/HOL. In a declarative proof each step of the proof is declared, similar to the sentences in a thorough paper proof. The prover allows proofs to mix the declarative style with automatic theorem proving by using a tableau prover. Our motivation is teaching how automated and declarative provers work and how they are used. The prover allows studying concrete code and a formal verification of correctness. We show examples of proofs and how they are made in the prover. The entire development runs in Isabelle's ML environment as an interactive application or can be used standalone in OCaml or Standard ML (or in other functional programming languages like Haskell and Scala with some additional work).

Keywords: Isabelle, verification, declarative proofs for first-order logic with equality, soundness, LCF-style prover

1. Introduction

There are two styles of writing proofs in provers – the procedural style and the declarative style. In the procedural style, users write a script of instructions that tells the prover how to prove a theorem. Only by executing each instruction can the user see what happens in the proof. In the declarative style, proofs resemble thorough proofs on paper because they are written as a chain of sentences of varying level of detail. Thus, a user can read and understand a declarative proof without executing the prover. The declarative style is supported in advanced proof assistants such as Isabelle/HOL [29].

We develop a declarative prover intended mainly for educational purposes that users can quite easily inspect and for which a formal soundness proof is also accessible in Isabelle/HOL. We do this by translating, to the functional programming language Standard ML (SML), John Harrison's interactive theorem prover for classical first-order logic with equality from his *Handbook of Practical Logic and Automated Reasoning* [13]. The kernel of his prover is based on a proof system that uses equality which is advantageous because it means that it avoids substitution.

*Corresponding author. E-mail: jovi@dtu.dk.

1.1. The LCF-style Prover

The main aim of the present work has been to evaluate the prospects of a simple LCF-style prover as a certified declarative first-order prover that is generated from a specification in Isabelle/HOL and that can be inspected for educational purposes.

The prover follows the LCF-style of having a trusted kernel on which other components are built [14]. The main benefit is that if the user trusts the kernel, then she can also trust the other components.

We take advantage of this in our certification. By certifying the soundness of the kernel, we ensure the soundness of all the other components because they rely on the kernel to generate theorems.

We therefore program a kernel similar to Harrison's in Isabelle/HOL and certify its soundness by defining a semantics on the first-order formulas with equality. Then we use Isabelle's code reflection facility to generate a module that represents the kernel and import it into the special Isabelle/ML environment for Standard ML. Hereafter we load the rest of the prover into Isabelle/ML along with a number of examples.

The whole prover, its verification and many examples of proofs are available in the Archive of Formal Proofs (AFP) [18]. The AFP entry is a single theory file which is structured as follows:

- A definition of the syntax of FOL with equality.
- A definition of the axioms and rules of the kernel.
- A definition of a semantics of FOL with equality.
- Definitions of a proof system consisting of the axioms and rules of the kernel.
- A soundness proof of the proof system.
- Code reflection of the axioms and rules.
- The prover that builds on top of the kernel.
- Examples of proofs in the prover.

The prover includes a tableau prover which allows proofs to mix the declarative proof style with automatic theorem proving, and we show several examples of such proofs.

1.2. Declarative Proof of Pelletier's Problem 46

As test cases for our declarative prover we have considered several of the most difficult first-order logic problems in Pelletier's *Seventy-Five Problems for Testing Automatic Theorem Provers* [31].

As mentioned earlier, the prover allows proofs to mix the declarative style with use of automatic theorem proving. Let us illustrate this by considering Pelletier's problem 46 in Figure 1.

We prove the formula in a declarative style with our prover as shown in Figure 2 including the use of the automatic tableau prover. To explain the intuition of the proof, Figure 3 shows the proof recast in natural language. The proof is structured in the same way as the declarative proof.

The idea of the proof is that we break down the structure of the formula until we are in a state where the automation can take care of the rest.

In the next section we provide a short introduction to Isabelle/HOL and in the subsequent sections we describe first the architecture for the declarative prover and then the formalization in Isabelle/HOL.

Parts of this paper are adapted from our previous workshop paper [17].

2. Isabelle/HOL

Isabelle/HOL is a proof assistant for higher-order logic. Higher-order logic can be thought of as a mix of logic and typed functional programming. Isabelle/HOL includes the usual logical connectives $\longrightarrow$, $\longleftrightarrow$, $\vee$, $\wedge$, $\neg$ as well as equality $=$ and non-equality $\neq$. Additionally Isabelle/HOL allows us to specify rules using $\Longrightarrow$. For instance Isabelle/HOL axiomatizes modus-ponens as $(P \longrightarrow Q) \Longrightarrow P \Longrightarrow Q$. It is convention to separate the assumptions from the conclusions of theorems and lemmas using $\Longrightarrow$ even though, at least logically, one might as well use $\longrightarrow$.

Isabelle/HOL also includes commands for defining types, defining functions and declaring theorems. We list these in Table 1.

In making our certified prover, we found the following tools of Isabelle/HOL essential:

- The structured proof language Isabelle/Isar [41], which offers ample features for writing declarative proofs, as well as proof methods such as simp, fastforce and metis, which can discharge proof goals [44].
- Sledgehammer [2], which can discharge proof goals by employing multiple automatic theorem provers (ATPs) as well as satisfiability-modulo-theories (SMT) solvers and proof reconstruction in Isabelle/Isar.
- Isabelle/ML [43], which is a way to use Standard ML (SML) inside the Isabelle environment. It can be embedded in Isabelle/Isar which means that it can be used side by side with Isabelle/HOL.
- Isabelle's code generation [10] and its code reflection is used to generate code from Isabelle/HOL definitions and load it into the Isabelle/ML environment.
- The Isabelle/jEdit Prover IDE (Integrated Development Environment) [42], which allows both for navigating, stating and checking formalizations in Isabelle/Isar and for programming and debugging in Isabelle/ML.

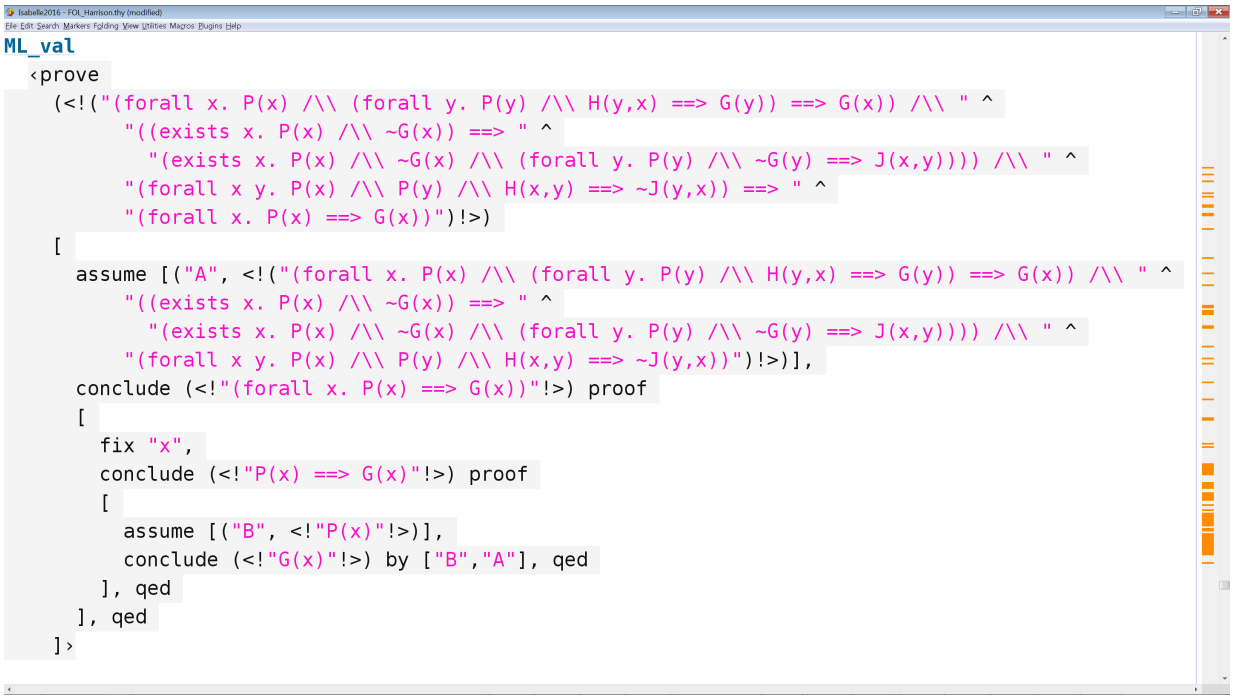
Isabelle/ML and in particular Isabelle's code generation have been most relevant for the integration of Isabelle/HOL and Standard ML code, and furthermore Isabelle/jEdit is used for our declarative prover too. In addition, Sledgehammer was particularly useful for the starting point of our work, namely Alexander B. Jensen's thesis [16], since it allows Isabelle novices to prove theorems without having deep knowledge about Isabelle's library of theorems.

3. Architecture for Declarative Prover

Figure 4 shows the architecture of the entire development. The development consists of a single Isabelle theory file, which has an Isabelle/HOL part and an

$$
\begin{aligned}
&(\forall x.\ P(x) \wedge (\forall y.\ P(y) \wedge H(y,x) \longrightarrow G(y)) \longrightarrow G(x)) \wedge \\
&((\exists x.\ P(x) \wedge \neg G(x)) \longrightarrow \\
&\quad (\exists x.\ P(x) \wedge \neg G(x) \wedge (\forall y.\ P(y) \wedge \neg G(y) \longrightarrow J(x,y)))) \wedge \\
&(\forall xy.\ P(x) \wedge P(y) \wedge H(x,y) \longrightarrow \neg J(y,x)) \longrightarrow \\
&(\forall x.\ P(x) \longrightarrow G(x))
\end{aligned}
$$

Fig. 1. Pelletier's problem 46



```
ML_val
  ‹prove
    (<!("(forall x. P(x) /\\ (forall y. P(y) /\\ H(y,x) ==> G(y)) ==> G(x)) /\\ " ^
          "((exists x. P(x) /\\ ~G(x)) ==> " ^
            "(exists x. P(x) /\\ ~G(x) /\\ (forall y. P(y) /\\ ~G(y) ==> J(x,y)))) /\\ " ^
          "(forall x y. P(x) /\\ P(y) /\\ H(x,y) ==> ~J(y,x)) ==> " ^
          "(forall x. P(x) ==> G(x))")!>)
    [
      assume [("A", <!("(forall x. P(x) /\\ (forall y. P(y) /\\ H(y,x) ==> G(y)) ==> G(x)) /\\ " ^
          "((exists x. P(x) /\\ ~G(x)) ==> " ^
            "(exists x. P(x) /\\ ~G(x) /\\ (forall y. P(y) /\\ ~G(y) ==> J(x,y)))) /\\ " ^
          "(forall x y. P(x) /\\ P(y) /\\ H(x,y) ==> ~J(y,x))")!>)],
      conclude (<!"(forall x. P(x) ==> G(x))"!>) proof
      [
        fix "x",
        conclude (<!"P(x) ==> G(x)"!>) proof
        [
          assume [("B", <!"P(x)"!>)],
          conclude (<!"G(x)"!>) by ["B","A"], qed
        ], qed
      ], qed
    ]›
```

Fig. 2. Example of a declarative proof in our prover running inside Isabelle/jEdit. The initial string encodes Pelletier's problem 46 and the prover function calls in black (`prove`, `assume`, `conclude`, `proof`, `fix` and `qed`) mark steps in the proof. The prover runs in the Isabelle/ML environment.

Isabelle/ML part. The two parts are connected with code reflection.

The part in Isabelle/HOL defines types for formulas and theorems, as well as functions for axioms and rules. These are then used inductively to define the proof system which is proved sound with respect to a semantics.

Hereafter, code reflection connects the Isabelle/HOL part with the Isabelle/ML part: Isabelle is instructed to generate code from the Isabelle/HOL definitions, and the code is then loaded into the Isabelle/ML environment. The loaded code consists of an ML module with a signature consisting of the type of formulas, the type of theorems, the axiom functions and the rule functions. It is the kernel of the declarative prover.

The part in Isabelle/ML defines the declarative prover. The declarative prover is a number of ML-functions that make calls into the kernel. These functions include derived rules, a tableau prover and various tactics.

The idea of the architecture is that we prove the axioms and rules sound in Isabelle/HOL. We load the axioms and rules into Isabelle/ML by using reflection, and the ML signature system then ensures that all values of the type for a theorem are built from the loaded axioms and rules. Thus these values represent theo-

We will prove Pelletier's problem 46 (Figures 1 and 2).

Since its outermost structure is an implication, we start by assuming the three formulas in the conjunction on the left-hand side of the arrow:

- We assume $\forall x.\ P(x) \wedge (\forall y.\ P(y) \wedge H(y,x) \longrightarrow G(y)) \longrightarrow G(x)$ and $(\exists x.\ P(x) \wedge \neg G(x)) \longrightarrow (\exists x.\ P(x) \wedge \neg G(x) \wedge (\forall y.\ P(y) \wedge \neg G(y) \longrightarrow J(x,y)))$ and $\forall xy.\ P(x) \wedge P(y) \wedge h(x,y) \longrightarrow \neg J(y,x)$.
  These assumptions are labeled A.
- From this we conclude the right-hand side $\forall x.\ P(x) \longrightarrow G(x)$.
  Since it is a universal quantification we do it as follows:
  - We fix an arbitrary element, x.
  - Then we conclude $P(x) \longrightarrow G(x)$ as follows:
    - We assume $P(x)$ and label the assumption B.
    - Then we conclude $G(x)$ which follows by assumptions A and B.
      This is proved automatically with the tableau prover.

Fig. 3. The declarative proof of Pelletier's problem 46 recast as a structured proof in natural language

Command	Description
type_synonym	Defines a syntactic abbreviation of a type.
datatype	Defines an ML-style datatype.
definition	Defines a (non-recursive) function or constant.
abbreviation	Defines a syntactical abbreviation of a term.
primrec	Defines a primitive recursive function.
inductive	Defines an inductive predicate based on a set of introduction rules.
lemma	Declares a lemma and is followed by a proof.
theorem	Declares a theorem and is followed by a proof.
corollary	Declares a corollary and is followed by a proof.
code_reflect	Generates code that is reflected into the Isabelle/ML environment.

Table 1
A subset of Isabelle/HOL's commands

rems of the sound proof system.

Our entire development can run from a single file in a window in the Isabelle/jEdit IDE. As already mentioned the file is available and maintained in the Archive of Formal Proofs against the current release of Isabelle and the file includes both the Isabelle/HOL and the Isabelle/ML part [18]. It is possible to replace the code-reflection with a code generation tool that exports the kernel to a source code file in either OCaml, Haskell, Scala or SML.

4. Formalization of Terms and Formulas

We formalize formulas in the same style as Harrison's OCaml code which has the parameter *'a* for the type of atoms and variable identifiers represented as strings.

type_synonym *id* = *String.literal*

This way the atoms can be instantiated for either propositional logic or first-order logic.

datatype *'a fm* = *Truth* | *Falsity* | *Atom 'a* |
Imp ⟨*'a fm*⟩ ⟨*'a fm*⟩ | *Iff* ⟨*'a fm*⟩ ⟨*'a fm*⟩ |
And ⟨*'a fm*⟩ ⟨*'a fm*⟩ | *Or* ⟨*'a fm*⟩ ⟨*'a fm*⟩ |
Not ⟨*'a fm*⟩ | *Exists id* ⟨*'a fm*⟩ | *Forall id* ⟨*'a fm*⟩

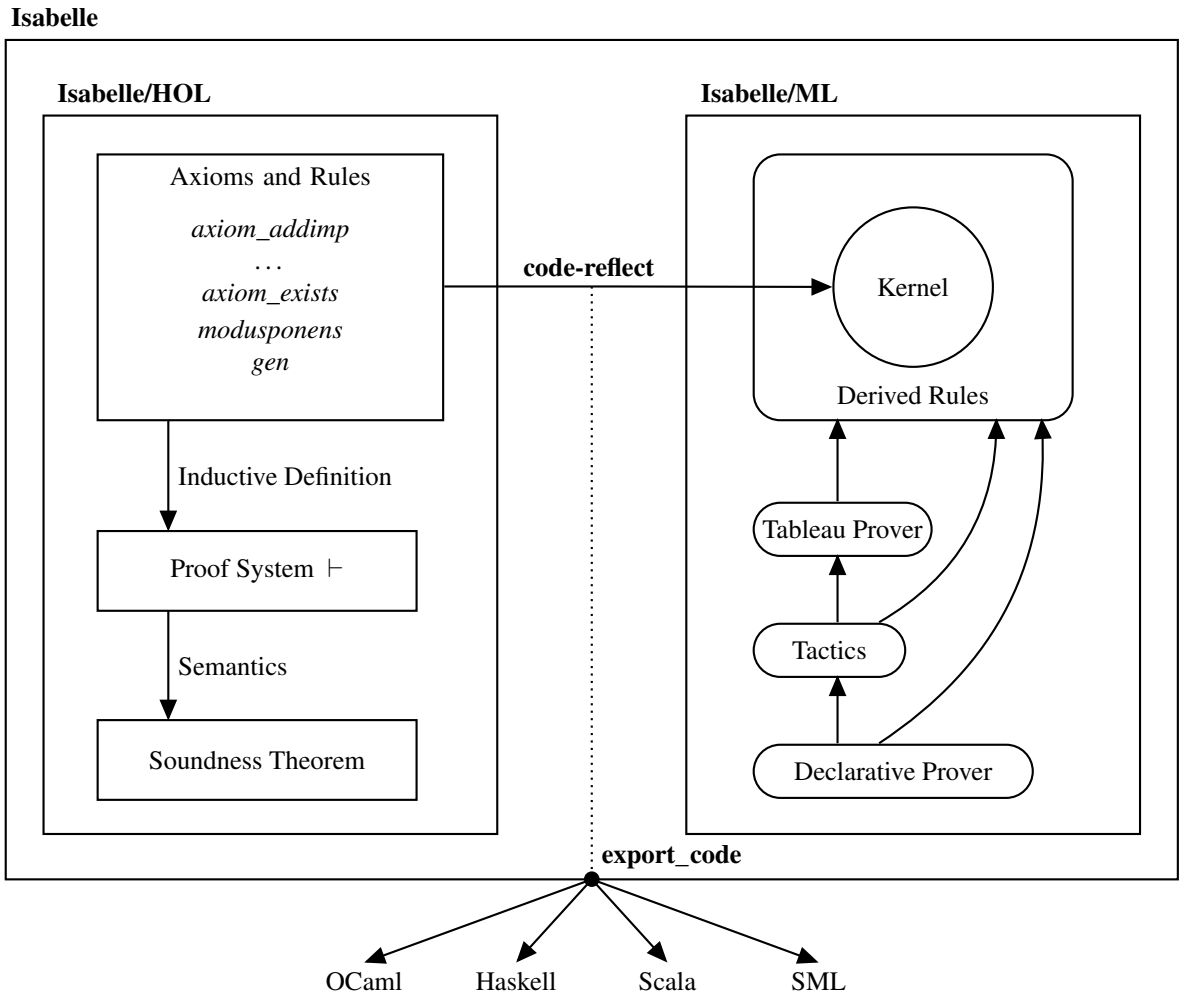


Fig. 4. Architecture of the certified declarative prover in Isabelle.

We similarly formalize the terms and first-order atoms. Function identifiers and predicate identifiers are also represented by strings:

datatype *tm* = *Var id* | *Fn id* ⟨*tm list*⟩

datatype *fol* = *Rl id* ⟨*tm list*⟩

Thus the first-order formulas are represented by the type *fol fm*.

5. Proof System

The entire axiomatic proof system can be seen in Figure 5 and comes from Harrison's textbook. The idea is that the validity of the rules and axioms should be evident and that they should be easy to implement. Harrison recognizes that substitution with named variable bindings is not entirely trivial. There are ways to alleviate these complications, for instance by using de Bruijn indices or nominals, but Harrison takes another approach. He takes inspiration from proof systems for first-order logic with equality by Tarski [39] and Monk [28] which avoid substitution entirely in their axioms and rules.

Harrison's rules and axioms in Figure 5 are structured as follows:

Inference rules (1-2). The inference rules are modus ponens (1) and generalization (2).

Propositional axioms (3-5). The propositional axioms, together with modus ponens (1), form a proof system of the propositional logic with $\longrightarrow$ and $\bot$ as the only operators. Harrison refers to the proof system P_0 by Church [7] which consists exactly of these three axioms and modus ponens.

First-order axioms (6-11). The first-order axioms, together with the propositional axioms and the inference rules, form a proof system for first-order logic with only the operators $\longrightarrow$, $\bot$, $\forall$ and the rest defined as abbreviations. Axioms 6-9 appear in the axiomatic systems by Tarski and Monk and so does an axiom similar to the congruence axioms 10-11.

Further operator axioms (12-19). These further operator axioms characterize $\longleftrightarrow$, $\top$, $\neg$, $\wedge$, $\vee$ and $\exists$ in terms of $\longrightarrow$, $\bot$ and $\forall$.

In addition to the advantage of leading to a simple kernel, the approach allows Harrison to present named variable bindings to the user without any conversion from an internal representation.

The same approach is used by the proof checker Metamath [25] which uses a similar set of axioms also inspired by Tarski [39].

6. Formalization of Axioms and Proof Rules

Since axioms and proof rules will be formalized as functions, they should be functions that return theorems. Therefore we introduce a datatype for the theorems, as well as a selector *concl*, such that *concl (Thm x) = x*.

datatype *"thm" = Thm (concl: ⟨fol fm⟩)*

We can then define the rules and axioms of the proof system as functions in Isabelle/HOL.

Let us consider the simplest such function, namely *axiom_addimp*.

definition *axiom_addimp :: "fol fm ⇒ fol fm ⇒ thm"*
where
"axiom_addimp p q ≡ Thm (Imp p (Imp q p))"

This axiom simply implements the well-known axiom $p \longrightarrow (q \longrightarrow p)$. Notice also the type annotation. The axiom takes two formulas and returns a theorem.

We also consider a proof rule, namely *gen*, which is the generalization rule.

definition *gen :: "id ⇒ thm ⇒ thm"*
where
"gen x s ≡ Thm (Forall x (concl s))"

This implements the rule $\frac{\vdash s}{\vdash \forall x.\ s}$. Notice that this function takes a theorem as input since it is a proof rule.

6.1. Side Conditions

The axioms *axiom_impall* and *axiom_existseq* have the side condition that x is not, respectively, free in p or occurs in t.

$$\frac{\neg \mathit{free\_in}\ x\ p}{p \longrightarrow (\forall x.\ p)} \qquad \frac{\neg \mathit{occurs\_in}\ x\ t}{\exists x.\ x = t}$$

Therefore we have to choose what the functions should return when the side conditions are not fulfilled.

Harrison chose to throw an exception but these are not available in Isabelle/HOL. We therefore considered several alternatives. One possibility would be to return *undefined*. Another possibility would be to return a *thm option* which would be *None* when the side conditions are not fulfilled.

We choose instead that the implementation returns *Thm Truth* when the side conditions are not fulfilled. This solution simplifies the code and the proofs. It clearly ensures soundness since, when things go wrong, we return a formula that is obviously valid.

abbreviation *(input) "fail_thm ≡ Thm Truth"*

We define the following functions for terms and lists of terms:

primrec
occurs_in :: "id ⇒ tm ⇒ bool"
and
occurs_in_list :: "id ⇒ tm list ⇒ bool"
where
"occurs_in i (Var x) = (i = x)" |
"occurs_in i (Fn _ l) = occurs_in_list i l" |
"occurs_in_list _ [] = False" |
"occurs_in_list i (h # t) =
(occurs_in i h ∨ occurs_in_list i t)"

We define the following function for formulas:

primrec *free_in :: "id ⇒ fol fm ⇒ bool"*
where
"free_in _ Truth = False" |
"free_in _ Falsity = False" |
"free_in i (Atom a) =
(case a of Rl _ l ⇒ occurs_in_list i l)" |
"free_in i (Imp p q) = (free_in i p ∨ free_in i q)" |
"free_in i (Iff p q) = (free_in i p ∨ free_in i q)" |
"free_in i (And p q) = (free_in i p ∨ free_in i q)" |
"free_in i (Or p q) = (free_in i p ∨ free_in i q)" |
"free_in i (Not p) = free_in i p" |
"free_in i (Exists x p) = (i ≠ x ∧ free_in i p)" |
"free_in i (Forall x p) = (i ≠ x ∧ free_in i p)"

1. modus ponens $\dfrac{p \longrightarrow q \qquad p}{q}$

2. generalization $\dfrac{p}{\forall x.\, p}$

3. axiom addimp $\dfrac{}{p \longrightarrow q \longrightarrow p}$

4. axiom distribimp $\dfrac{}{(p \longrightarrow q \longrightarrow r) \longrightarrow (p \longrightarrow q) \longrightarrow p \longrightarrow r}$

5. axiom doubleneg $\dfrac{}{((p \longrightarrow \bot) \longrightarrow \bot) \longrightarrow p}$

6. axiom allimp $\dfrac{}{(\forall x.\, p \longrightarrow q) \longrightarrow (\forall x.\, p) \longrightarrow (\forall x.\, q)}$

7. axiom impall $\dfrac{\neg \mathit{free\_in}\ x\ p}{p \longrightarrow (\forall x.\, p)}$

8. axiom existseq $\dfrac{\neg \mathit{occurs\_in}\ x\ t}{\exists x.\, x = t}$

9. axiom eqrefl $\dfrac{}{t = t}$

10. axiom funcong $\dfrac{}{s_1 = t_1 \longrightarrow \cdots \longrightarrow s_n = t_n \longrightarrow f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n)}$

11. axiom predcong $\dfrac{}{s_1 = t_1 \longrightarrow \cdots \longrightarrow s_n = t_n \longrightarrow P(s_1, \ldots, s_n) \longrightarrow P(t_1, \ldots, t_n)}$

12. axiom iffimp1 $\dfrac{}{(p \longleftrightarrow q) \longrightarrow p \longrightarrow q}$

13. axiom iffimp2 $\dfrac{}{(p \longleftrightarrow q) \longrightarrow q \longrightarrow p}$

14. axiom impiff $\dfrac{}{(p \longrightarrow q) \longrightarrow (q \longrightarrow p) \longrightarrow (p \longleftrightarrow q)}$

15. axiom true $\dfrac{}{\top \longleftrightarrow (\bot \longrightarrow \bot)}$

16. axiom not $\dfrac{}{\neg p \longleftrightarrow (p \longrightarrow \bot)}$

17. axiom and $\dfrac{}{(p \wedge q) \longleftrightarrow ((p \longrightarrow q \longrightarrow \bot) \longrightarrow \bot)}$

18. axiom or $\dfrac{}{(p \vee q) \longleftrightarrow \neg(\neg p \wedge \neg q)}$

19. axiom exists $\dfrac{}{(\exists x.\, p) \longleftrightarrow \neg(\forall x.\, \neg p)}$

Fig. 5. The axiomatic proof system.

```
definition axiom_impall :: "id ⇒ fol fm ⇒ thm"
where
 "axiom_impall x p ≡
   if ¬ free_in x p then Thm (Imp p (Forall x p))
   else fail_thm"
```

Axiom *axiom_existseq* is defined in the same way as axiom *axiom_impall*.

6.2. Congruence Axioms

The most complicated axioms are the congruence axioms, *axiom_funcong* and *axiom_predcong*.

$$\frac{}{s_1 = t_1 \longrightarrow \cdots \longrightarrow s_n = t_n \longrightarrow f(s_1, \cdots, s_n) = f(t_1, \cdots, t_n)}$$

$$\frac{}{s_1 = t_1 \longrightarrow \cdots \longrightarrow s_n = t_n \longrightarrow P(s_1, \cdots, s_n) \longrightarrow P(t_1, \cdots, t_n)}$$

For *axiom_funcong*, Harrison's implementation takes the two lists $lefts = [s_1, \ldots, s_n]$ and $rights = [t_1, \ldots, t_n]$ as input, and constructs the above nested implication.

```
let axiom_funcong f lefts rights =
      itlist2
        (fun s t p -> Imp (mk_eq s t,p)) lefts rights
          (mk_eq (Fn (f,lefts)) (Fn (f,rights)))
```

The function *itlist2* is defined as

```
let rec itlist2 f l1 l2 b =
  match (l1,l2) with
    ([],[]) -> b
  | (h1::t1,h2::t2) -> f h1 h2 (itlist2 f t1 t2 b)
  | _ -> failwith "itlist2";;
```

His idea is that we have a function which adds an equality of two terms as an antecedent to a formula. Then we can use that function and *itlist2* to iteratively add equalities of the terms in our lists as antecedents starting from the formula $f(t_1, \ldots, t_n) = f(s_1, \ldots, s_n)$.

Our formalization instead splits the functionality of *axiom_funcong* into two functions:

- *foldr Imp* takes a list of formulas $[F_1, \ldots, F_n]$ and adds them as antecedents to a formula F to build a nested implication $F_1 \longrightarrow \cdots \longrightarrow F_n \longrightarrow F$.
- *zip_eq* takes two lists of formulas, $[s_1, \ldots, s_n]$, $[t_1, \ldots, t_n]$ and builds the corresponding list of equalities $[s_1 = t_1, \ldots, s_n = t_n]$.

```
definition zip_eq :: "tm list ⇒ tm list ⇒ fol fm list"
where
 "zip_eq l l' ≡ map (λ(t, t'). Atom (Rl (STR ''='') [t, t'])
                    (zip l l')"
```

The idea of our approach is that we can separately reason about constructing equalities and adding antecedents, and this will make it easier to prove soundness. We now implement *axiom_funcong* as follows by first constructing the equalities, and then the nested implication.

```
definition axiom_funcong :: "id ⇒ tm list ⇒ tm list ⇒ thm"
where
 "axiom_funcong i l l' ≡
   if equal_length l l' then
     Thm (foldr Imp (zip_eq l l')
           (Atom (Rl (STR ''='') [Fn i l, Fn i l'])))
   else fail_thm"
```

We formalize *axiom_predcong* in a similar way.

```
definition axiom_predcong :: "id ⇒ tm list ⇒ tm list ⇒ thm"
where
 "axiom_predcong i l l' ≡
   if equal_length l l' then
     Thm (foldr Imp (zip_eq l l')
           (Imp (Atom (Rl i l)) (Atom (Rl i l'))))
   else fail_thm"
```

7. Formalization of Axiomatic Proof System

Since we want to prove the whole system sound, we need to characterize the theorems, which are built exclusively from the axioms and the rules. We therefore define the proof system as an inductive predicate *OK*. Writing (*"⊢ _" 0*) we introduce the turnstile as syntax for the *OK* predicate where the underscore denotes that the formula follows the turnstile. The 0 denotes the precedence of the notation. After **where** follows each of the rules and axioms as introduction rules in the inductive predicate. The underscores there are dummy variables, that is, each one of them corresponds to a fresh Isabelle/HOL variable.

```
inductive OK :: "fol fm ⇒ bool" ("⊢ _" 0)
where
```

```
modusponens:
  "⊢ concl s ⟹
   ⊢ concl s' ⟹     ⊢ concl (modusponens s s')" |
gen:
  "⊢ concl s ⟹     ⊢ concl (gen _ s)" |
axiom_addimp:      "⊢ concl (axiom_addimp _ _)" |
axiom_distribimp:  "⊢ concl (axiom_distribimp _ _ _)" |
axiom_doubleneg:   "⊢ concl (axiom_doubleneg _)" |
axiom_allimp:      "⊢ concl (axiom_allimp _ _ _)" |
axiom_impall:      "⊢ concl (axiom_impall _ _)" |
axiom_existseq:    "⊢ concl (axiom_existseq _ _)" |
axiom_eqrefl:      "⊢ concl (axiom_eqrefl _)" |
axiom_funcong:     "⊢ concl (axiom_funcong _ _ _)" |
axiom_predcong:    "⊢ concl (axiom_predcong _ _ _)" |
axiom_iffimp1:     "⊢ concl (axiom_iffimp1 _ _)" |
axiom_iffimp2:     "⊢ concl (axiom_iffimp2 _ _)" |
axiom_impiff:      "⊢ concl (axiom_impiff _ _)" |
axiom_true:        "⊢ concl axiom_true" |
axiom_not:         "⊢ concl (axiom_not _)" |
axiom_and:         "⊢ concl (axiom_and _ _)" |
axiom_or:          "⊢ concl (axiom_or _ _)" |
axiom_exists:      "⊢ concl (axiom_exists _ _)"
```

8. Semantics

To prove the rules sound, we of course need a semantics of terms and formulas. The formalization is mostly straightforward. We represent universes as types, and therefore the semantics refers to the universe by a type variable $'a$. A noteworthy case of the semantics is the one for the atoms, where we interpret the $=$ predicate applied to two terms as an equality. This is done by evaluating the terms and seeing if their values are equal.

```
primrec — Semantics of terms
 semantics_term ::
  "(id ⇒ 'a) ⇒ (id ⇒ 'a list ⇒ 'a) ⇒ tm ⇒ 'a"
and
 semantics_list ::
  "(id ⇒ 'a) ⇒ (id ⇒ 'a list ⇒ 'a) ⇒ tm list ⇒ 'a list"
where
 "semantics_term e _ (Var x) = e x" |
 "semantics_term e f (Fn i l) = f i (semantics_list e f l)" |
 "semantics_list _ _ [] = []" |
 "semantics_list e f (t # l) =
   semantics_term e f t # semantics_list e f l"

primrec — Semantics of formulas
 semantics
  :: "(id ⇒ 'a) ⇒ (id ⇒ 'a list ⇒ 'a) ⇒
          (id ⇒ 'a list ⇒ bool) ⇒ fol fm ⇒ bool"
where
 "semantics _ _ _ Truth = True" |
 "semantics _ _ _ Falsity = False" |
 "semantics e f g (Atom a) =
   (case a of Rl i l ⇒
     if i = STR ''='' ∧ length2 l then
       (semantics_term e f (hd l) =
        semantics_term e f (hd (tl l)))
     else 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 (Iff p q) =
   (semantics e f g p ⟷ semantics e f g q)" |
 "semantics e f g (And p q) =
   (semantics e f g p ∧ semantics e f g q)" |
 "semantics e f g (Or p q) =
   (semantics e f g p ∨ semantics e f g q)" |
 "semantics e f g (Not p) = (¬ semantics e f g p)" |
 "semantics e f g (Exists x p) =
   (∃ v. semantics (e(x := v)) f g p)" |
 "semantics e f g (Forall x p) =
   (∀ v. semantics (e(x := v)) f g p)"
```

9. Soundness of the Proof System

Harrison only presents a very high-level soundness proof which leaves most of the exercise up to the reader. Furthermore, his proof is about the proof system, not its implementation. Our approach is therefore to device a proof ourselves, using Isabelle/jEdit to explore proofs and to help us reveal the necessary lemmas.

We prove the soundness by rule induction on the proof system, and thus need to prove one case for each axiom and rule. Cases for the axioms without side conditions and the *gen* rule are proved sound using only the automation of Isabelle/HOL.

Cases for the axioms with side conditions are not as easy to prove. Here, we need to come up with appropriate lemmas to prove them sound. We present and explain these lemmas.

The *modus_ponens* case is proved with a short declarative proof that also relies on automation.

9.1. Axioms with Non-Free or Non-Occurring Variables

The first challenge in the soundness proof is the two axioms, *axiom_impall* and *axiom_existseq*, that require a variable to be respectively non-free or non-occurring in an expression. For *axiom_existseq* it is clear that the formula holds if we assign the value of x to t. By inspecting the semantics of the existential quantifier, we

realize that the variable x must not occur in t. It is however not clear to Isabelle that this problem is avoided when x does not occur in t. The lemma *map'* explicitly states that if x does not occur in t, then the semantic value of t is the same for all values of x. The statement is extended to hold for lists of terms due to the inductive definition of terms.

lemma *map'*:
"¬ occurs_in x t ⟹
semantics_term e f t = semantics_term (e(x := v)) f t"
"¬ occurs_in_list x l ⟹
semantics_list e f l = semantics_list (e(x := v)) f l"

The lemma *map* is similar, but states that if the variable x does not occur freely in p (or if it does not occur at all) then the truth value of p is the same for all values of x.

lemma *map*:
"¬ free_in x p ⟹
semantics e f g p ⟷ semantics (e(x := v)) f g p"

By inspecting the semantics of the universal quantifier, we see that this exactly states that the semantics of p is the same as $\forall x.\ p$. This fact is an even stronger result than what we need to prove *axiom_impall* valid. We are now ready to prove *axiom_impall* using *map* and *axiom_existseq* using *map'*.

9.2. Congruence Axioms

The next challenge is to prove the congruence axioms, *axiom_funcong* and *axiom_predcong* sound. We now take advantage of the *foldr Imp* function we introduced earlier, and prove a lemma explaining its semantics. The lemma states that a nested implication is true exactly when the truth of its antecedents implies the truth of its conclusion.

lemma *imp_chain_equiv*:
"semantics e f g (foldr Imp l p) ⟷
(∀ q ∈ set l. semantics e f g q) ⟶ semantics e f g p"

We then also state a lemma which explains the semantics of *foldr Imp (zip_eq l l') p*. The lemma states that it holds exactly when the semantical equality between l and l' implies the truth of p.

lemma *imp_chain_zip_eq*:
"equal_length l l' ⟹
semantics e f g (foldr Imp (zip_eq l l') p) ⟷
semantics_list e f l = semantics_list e f l' ⟶
semantics e f g p"

With this we prove the congruence axioms sound using automation in lemmas *funcong* and *predcong*. It is easy to see that the lemma *funcong* proves a theorem in the first-order logic, since its conclusion is encapsulated by *semantics*. The formula is the main content of the Isabelle definition of *axiom_funcong*.

lemma *funcong*:
"equal_length l l' ⟹
semantics e f g (foldr Imp (zip_eq l l')
(Atom (Rl (STR ''='') [Fn i l, Fn i l'])))"

Likewise in the lemma *predcong*, we see a theorem in the first-order logic. The formula is the main content of the Isabelle definition of *axiom_predcong*.

lemma *predcong*:
"equal_length l l' ⟹
semantics e f g (foldr Imp (zip_eq l l')
(Imp (Atom (Rl i l)) (Atom (Rl i l'))))"

9.3. Soundness Theorem

We then prove soundness. Often, soundness is expressed as provability implying validity. Therefore we would like a HOL predicate expressing validity. That is unfortunately not possible, because of our choice of representing universes as types, which one cannot quantify over inside HOL.

Instead, we express soundness as follows:

theorem *soundness*:
"⊢ p ⟹ semantics e f g p"

This also expresses soundness, since it states that the provability of any formula p implies its truth, for any environment e, function denotation f and predicate denotation g. The proof is by rule-induction on the proof system as described.

From our main theorem we immediately obtain a consistency corollary which states that there is a formula that we cannot prove:

corollary *"¬ (⊢ Falsity)"*
using *soundness*
by *fastforce*

10. Prover

Since we have defined all the necessary datatypes for our logic as well as the axioms and rules used to construct theorems, we are ready to expose them to the Isabelle/ML environment using code-reflection. To do

this we use the Isabelle **code_reflect** command which takes a structure name as well as a list of datatypes and their constructors, as well as a list of functions. It then generates a signature and structure based on them and exposes it to the Isabelle/ML environment. In particular, we tell Isabelle that the datatypes *fm*, *tm* and *fol* should be in the signature along with their respective constructors. Likewise we tell Isabelle that the signature should include functions *modusponens, gen, ..., concl*.

```
code_reflect
  Proven
datatypes
  fm = Falsity | Truth | Atom | Imp | Iff |
       And | Or | Not | Exists | Forall
and
  tm = Var | Fn
and
  fol = Rl
functions
  modusponens gen axiom_addimp axiom_distribimp
  axiom_doubleneg axiom_allimp axiom_impall
  axiom_existseq axiom_eqrefl axiom_funcong
  axiom_predcong axiom_iffimp1 axiom_iffimp2
  axiom_impiff axiom_true axiom_not axiom_and
  axiom_or axiom_exists concl
```

Let us inspect the signature of the generated module:

```
structure Proven:
  sig
    val axiom_addimp: fol fm -> fol fm -> thm
    val axiom_allimp: string -> fol fm -> fol fm -> thm
    val axiom_and: fol fm -> fol fm -> thm
    val axiom_distribimp: fol fm -> fol fm -> fol fm -> thm
    val axiom_doubleneg: fol fm -> thm
    val axiom_eqrefl: tm -> thm
    val axiom_exists: string -> fol fm -> thm
    val axiom_existseq: string -> tm -> thm
    val axiom_funcong: string -> tm list -> tm list -> thm
    val axiom_iffimp1: fol fm -> fol fm -> thm
    val axiom_iffimp2: fol fm -> fol fm -> thm
    val axiom_impall: string -> fol fm -> thm
    val axiom_impiff: fol fm -> fol fm -> thm
    val axiom_not: fol fm -> thm
    val axiom_or: fol fm -> fol fm -> thm
    val axiom_predcong: string -> tm list -> tm list -> thm
    val axiom_true: thm
    val concl: thm -> fol fm
    datatype 'a fm =
      And of 'a fm * 'a fm
    | Atom of 'a
    | Exists of string * 'a fm
    | Falsity
    | Forall of string * 'a fm
    | Iff of 'a fm * 'a fm
    | Imp of 'a fm * 'a fm
    | Not of 'a fm
    | Or of 'a fm * 'a fm
    | Truth
    datatype fol = Rl of string * tm list
    val gen: string -> thm -> thm
    val modusponens: thm -> thm -> thm
    type num
    type thm
    datatype tm = Fn of string * tm list | Var of string
  end
```

By inspecting the signature of the reflected module we see that the only functions that return values of type *thm* are the axioms and rules. This fact and the soundness proof certify the soundness of the kernel assuming that we trust ML's type-system and Isabelle's code generator.

Notice that a user cannot write e.g. *Thm Falsity* in ML since the *Thm* value constructor is not exposed in the signature and thus unavailable to the user.

11. Declarative Prover

The signature above fits with the one in Harrison's prover. We translate his prover from OCaml to SML such that we can run the prover inside of Isabelle in the Isabelle/ML environment. The translation was not too difficult, but there were some challenges arising from the differences between SML (as defined in the revised standard from 1997 [27]) and OCaml.

- In SML there is no built-in polymorphic ordering and hashing. Therefore we, when needed, define orderings and hash functions explicitly for each datatype.
- In SML there is no shallow/pointer comparison. All places it is used in the OCaml version we can fortunately replace it with structural equality.
- In SML one cannot put guards on case-expressions. Therefore we use if-then-else instead in these cases.
- OCaml has widely used preprocessors (camlp4 and camlp5). Harrison uses them when parsing formulas. We choose not to use a preprocessor. One unfortunate consequence of this is that when we want to use formulas as input, they are strings, and thus /\ needs to written as /\\ in order to escape the backslash.

Harrison's OCaml code contains many examples that are run when executing his code. We have collected these examples and translated them to SML. We have systematically tested that both versions produce the same output. The results are available online [37].

Let us take a look at how (our translation of) Harrison's proof assistant works and how it plugs into our generated kernel.

11.1. Derived Rules

The rules and axioms are functions that return theorems. By combining them Harrison obtains new such functions, i.e. derived rules. For instance the rule which takes a theorem $\vdash q$ and produces $\vdash p \longrightarrow q$ where p is some formula. The ML-implementation of this looks as follows:

```
fun add_assum p th =
    modusponens (axiom_addimp (concl th) p) th;
```

The idea is that the above code implements the following proof where we think of q as *concl th*:

1. $\vdash q$
2. $\vdash q \longrightarrow p \longrightarrow q$ (axiom_addimp)
3. $\vdash p \longrightarrow q$ (modusponens 1 2)

We can also inspect the type of *add_assum* and see that it indeed takes a formula and a theorem and returns a theorem:

```
fm -> thm -> thm
```

11.2. Tableau Prover

Harrison implements a tableau prover for first-order logic on top of the kernel. It is implemented as code and thus calls into the kernel. The prover implements a tableau system with unification — see e.g. Wikipedia [45] or Hähnle's chapter in the *Handbook of Automated Reasoning* [11]. The tableau is expanded in a preorder-fashion. Whenever a pair of complementary literals is found the resulting unifier will be applied to an environment that is passed on to the next node to be expanded. In the code, branching on a disjunctive formula is handled by working on the left branch immediately and delaying the work on the right branch by building a continuation function.

The tableau prover takes as parameter a number n indicating how many times universal quantifiers are allowed to introduce fresh variables. An outer function tries to build tableaux with larger and larger n until it, if the formula can be refuted, succeeds.

Harrison proves informally that the tableau prover is complete for first-order logic without equality.

11.3. Tactics

Tactics are a way to implement backwards reasoning in a proof assistant. When a user wants to use tactics he first states the goal he wants to prove. He can then use a tactic to reduce the goal to a number of subgoals from which the goal follows. The subgoals can likewise be reduced with tactics until they become trivial and then the proof is done.

A state in this process is represented by a datatype *goals* which looks as follows in the ML-implementation:

```
datatype goals =
  Goals of ((string * fol fm) list * fol fm)list *
       (thm list -> thm);
```

Each (*string* $*$ *fol fm*) *list* $*$ *fol fm* represents a subgoal with a number of assumptions. More precisely, the subgoal value $([p_1, \ldots, p_i], q)$ represents the implication $p_1 \wedge ... \wedge p_i \longrightarrow q$.

The ((*string* $*$ *fol fm*) *list* $*$ *fol fm*) *list* represents a list of subgoals:

$$\begin{array}{l} p_{11} \wedge \ldots \wedge p_{1i_1} \longrightarrow q_1 \\ \vdots \\ p_{n1} \wedge \ldots \wedge p_{ni_n} \longrightarrow q_n \end{array}$$

The *string* in the type allows us to label the assumptions.

The (*thm list* −> *thm*) is called a justification function and represents a rule which will bring us from the subgoals to the goal P we ultimately want to prove. It should thus represent a rule on the following form:

$$\begin{pmatrix} \vdash p_{11} \wedge \ldots \wedge p_{1i_1} \longrightarrow q_1 \\ \vdots \\ \vdash p_{n1} \wedge \ldots \wedge p_{ni_n} \longrightarrow q_n \end{pmatrix} \Longrightarrow \vdash P$$

Let's consider a simple example of a *goals*. Say we want to prove $\top \wedge \top$. A *goals* for this could be a list with the single subgoal

$$\top \wedge \top$$

together with a justification:

$$(\vdash \top \wedge \top) \Longrightarrow (\vdash \top \wedge \top)$$

In ML this could be the value ([([],*And*(*Truth*,*Truth*)],*hd*) where *hd* gives the head of a list, and so indeed when it is given $[\vdash \top \wedge \top]$ it will return $\vdash \top \wedge \top$.

A tactic is then simply a function of the type *goals* -> *goals* that should reduce the subgoals to something simpler and change the justification function accordingly. This is similar to how it was done in LCF [9, 26].

For instance we could apply a conjunction introduction tactic to our current example which would then produce the subgoals

$$\begin{array}{l}\top\\ \top\end{array}$$

together with a justification:

$$\begin{pmatrix}\vdash \top\\ \vdash \top\end{pmatrix} \Longrightarrow \vdash \top \wedge \top$$

A simple example of a tactic is the conjunction introduction tactic which replaces a subgoal of the form $a \longrightarrow p \wedge q$ with two subgoals $a \longrightarrow p$ and $a \longrightarrow q$.

Let us look at how to program the conjunction introduction tactic. In general, the tactic is supposed to go from a *goals* with the following justification (and a corresponding list of subgoals)

$$\begin{pmatrix}\vdash p_{11} \wedge \ldots \wedge p_{1i_1} \longrightarrow a \wedge b\\ \vdash p_{21} \wedge \ldots \wedge p_{2i_2} \longrightarrow q_2\\ \vdots\\ \vdash p_{n1} \wedge \ldots \wedge p_{ni_n} \longrightarrow q_n\end{pmatrix} \Longrightarrow \vdash P$$

to the following justification (and a corresponding list of subgoals)

$$\begin{pmatrix}\vdash p_{11} \wedge \ldots \wedge p_{1i_1} \longrightarrow a\\ \vdash p_{11} \wedge \ldots \wedge p_{1i_1} \longrightarrow b\\ \vdash p_{21} \wedge \ldots \wedge p_{2i_2} \longrightarrow q_2\\ \vdots\\ \vdash p_{n1} \wedge \ldots \wedge p_{ni_n} \longrightarrow q_n\end{pmatrix} \Longrightarrow \vdash P$$

The tactic is implemented as follows. The call it makes to *imp_trans_chain* and *and_pair* is described below, but for brevity we leave out the function definitions.

```
fun conj_intro_tac (Goals((asl,And(a,b))::gls,jfn)) =
 let fun jfn' (tha::thb::ths) =
  jfn(imp_trans_chain [tha, thb] (and_pair a b)::ths) in
 Goals((asl,a)::(asl,b)::gls,jfn')
 end;
```

The subgoals are changed as described – namely from (*asl*,*And*(*a*,*b*))::*gls* to (*asl*,*a*)::(*asl*,*b*)::*gls*. There is also a new justification function *jfn'*. In its definition the function call *imp_trans_chain* [*tha, thb*] (*and_pair a b*) takes theorems $\vdash asl \longrightarrow a$ and $\vdash asl \longrightarrow b$ and from these produces $\vdash asl \longrightarrow a \wedge b$ by calls into the kernel. When this is done we have the list of theorems that the original justification function expected and we can then simply apply it to produce the theorem we finally want.

Harrison implements several functions that are, or return, tactics:

- *conj_intro_tac* – conjunction introduction
- *forall_intro_tac* – forall introduction
- *exists_intro_tac* – existential introduction
- *imp_intro_tac* – implication introduction
- *auto_tac* – tableau prover
- *lemma_tac* – adding a new assumption
- *exists_elim_tac* – existential elimination
- *disj_elim_tac* – disjunction elimination

11.4. Declarative Proofs

Harrison builds the deductive prover on top of the tactics. The first step is to define a function *prove* which takes a formula and a list of tactics. It then sets up a *goals* with that formula and applies the tactics in the list one after another. In the end it returns the formula as a theorem if the tactics were successful in proving it.

We use this function in Fig. 6 to conduct a proof. Notice that the proof is actually an SML-expression directly calling *prove*. Because proofs are SML-expressions it is easy to extend the prover's syntax by writing new functions. The proof has a nested structure with some subproofs introduced by the function *proof* that are processed in a similar way. Some of these proofs use the *have* function to state intermediate steps towards proving the final goal. Let us first look at how this works if we for instance write *have p using [q]* in some goals *g*. Here *p* is some formula and *q* is some theorem. The *have* function calls *lemma_tac* with *p* and *using [q]*, and the following happens:

The *goals g* has a first subgoal of the form $asl \longrightarrow w$. This subgoal is replaced with $p \longrightarrow asl \longrightarrow w$.

Furthermore the justification function is changed. The new one calls *using [q]* which constructs the theorem $\vdash asl \longrightarrow univeral\_closure\ q$. Hereafter it will use the tableau prover to construct $\vdash universal\_closure\ q \longrightarrow p$ if possible. From this follows $\vdash asl \longrightarrow p$. The new justification function expects as input $\vdash p \longrightarrow asl \longrightarrow w$ and therefore it can now construct $\vdash asl \longrightarrow w$. This is what the old justification function expected as input and thus it is applied.

As we saw, *have* and *using* can be used to prove an intermediate step with a previously proved theorem. Likewise, *have* and *by* can be used to refer to a previously established fact in the proof. In the implementation this can be done by ensuring that the steps that introduce facts put them in the assumption list of the goals – as we saw *have* did. Then *by* can simply find them there by their name. Combining *have* and *proof* allows subproofs to prove intermediate steps in a similar manner.

Other tactics to be used in declarative proofs are

- *note* – similar to *have*, but the intermediate step is named.
- *fix* – which is simply a forall-introduction rule.
- *assume* – which does implication introduction.
- *consider* – which does existential elimination and introduces an appropriate variable.
- *so* – which modifies e.g. the *have* tactic to use the previous fact to prove its intermediate step.
- *conclude* – indicates that we prove a subgoal.
- *qed* – indicates the end of the proof.

The declarative prover is able to give the user a rudimentary form of feedback when developing the proof:

- The type system of SML will tell the user if she enters a proof wrongly on the highest level
- On the lower level, the tactics will throw exceptions if they are applied on a *goals* they did not expect.

We have tried several workflows for building proofs:

- It is possible, but arguably a bit tedious, to build the proof from scratch with the help from the SML type system and the exceptions.
- Another possibility is to start with a formula that can be proved with the tableau prover, and then expand the proof more and more to the desired granularity, each time filling in the next part to be expanded with a call to the tableau prover. This of course only works on formulas that the prover can prove.
- A third possibility is to write the proof first by manually applying tactics to goals and printing the resulting goals until one has a proof. Hereafter the proof can be reconstructed in the declarative style.

Isabelle/jEdit presents both type errors and exceptions directly in its output panel, which is updated live while the user is writing her proof. However, there is definitely room for improvement when it comes to the usability of the prover.

For declarative proofs it is a huge advantage to have powerful proof automation that can take care of some of the simpler steps. As a rather challenging example, consider the following formalization with predicate r for *rich* and function f for *father* [23, page 128]:

> *If every person that is not rich has a rich father, then some rich person must have a rich grandfather.*
>
> $\forall x(\neg r(x) \rightarrow r(f(x))) \rightarrow \exists x(r(x) \wedge r(f(f(x))))$

The tableau prover can in fact find the proof automatically and almost instantaneously. We can easily use the tableau prover and/or the declarative prover as a stand-alone program as follows. In Isabelle, we can introduce a Standard ML function *auto* and test it on some examples including the above one (of course a more advanced version of the function *auto* is possible, also using some helper functions):

```
ML {*
 fun auto s = prove (<!s!>) [our thesis at once, qed]
*}

ML_val {* auto "A ==> A" *}

ML_val {* auto "exists x. D(x) ==> forall x. D(x)" *}

ML_val {* auto "(forall x. ~R(x) ==> R(f(x)))
                    ==> exists x. R(x) /\ R(f(f(x)))" *}
```

Using Isabelle's code generator we obtain a stand-alone Standard ML program *auto* for the certified automated theorem prover. We have then used the tool SMLtoJs ("SML toys") to translate the Standard ML code to JavaScript such that we can use it in our NaDeA system [40] running in a browser (here *auto* is run for just a fraction of a second and if necessary terminated).

12. Evaluation of the Declarative Prover

We first evaluate the usability of the declarative prover and then we evaluate the adequacy of the soundness proof.

12.1. Usability of the Declarative Prover

Recall Figures 1 and 2 with Pelletier's problem 46. With this example we have already shown that we can prove a challenging theorem in the prover with the declarative style.

We now wish to further evaluate the prover by using it to prove a mathematical theorem. We therefore consider Pelletier's problem 43. The problem defines from a relation P another relation Q as follows:

$$Q(x,y) \longleftrightarrow (\forall z.\ P(z,x) \longleftrightarrow P(z,y))$$

The problem then claims that Q is symmetric.

Additionally, we to want construct a declarative proof with a stronger resemblance to our understanding of thorough paper proofs as chains of sentences. Thus, the proof should break down the structure of the formula to an appropriate level, and on that level resemble a thorough paper proof consisting of a chain of sentences.

Figure 6 shows such a proof in the declarative proof language. We also discuss this and some alternative proofs in Alexander B. Jensen's thesis [16].

The first step of the proof is to show $\forall x\,y.\ Q(x,y) \longleftrightarrow Q(y,x)$ assuming $(\forall x\,y.\ Q(x,y) \longleftrightarrow \forall z.\ P(z,x) \longleftrightarrow P(z,y))$. We assume the left-hand side of the implication in the main formula with the *assume* command and give it the name A. The command is in many ways similar to its Isabelle counterpart. We then fix variables x and y using *fix x, fix y* to eliminate the quantifiers. We further break down the problem and show the conjunction of both directions of the bi-implication in $Q(x,y) \longleftrightarrow Q(y,x)$. We show the formula using the command *have* which is similar to *conclude* except that it does not have to directly solve a sub-goal. The sub-goal $\forall z.\ P(z,y) \longleftrightarrow P(z,x)$ for the $\longrightarrow$ direction is solved by using the assumption A which is achieved by *so have* (< *!"forall z. P(z,x) <=> P(z,y)"!*>) *by* ["*A*"]. In the following sub-goal, where the left-hand side and right-hand side are swapped, we use only the previous fact and no assumptions. The command *at once* can be used when the goal can be solved by pure first-order reasoning from the previous fact. From the conjunction of implications, we show that it is equivalent to the bi-implication using *so our thesis at once* and thus finish the proof.

To show that this proof is comparable to the declarative proofs in Isabelle/Isar and Isabelle/HOL, we present in Figure 7 a similar proof in that system. The correspondence is clear.

12.2. Adequacy of the Soundness Proof

Let us also evaluate the soundness proof. In order to believe it, we need to convince ourselves that the $\vdash$ predicate indeed represents the ML-type *thm*. In order to do this we need to check that all the axioms and rules for which we generate code, indeed appear in the definition of $\vdash$. The process is easy but allows for mistakes due to human error. Imagine for instance that someone writes the following axiom, and generates code for it, but forgets to add it to $\vdash$ – thus bypassing the soundness proof.

definition *axiom_false* :: *"thm"* **where**
"axiom_false $\equiv$ *Thm Falsity"*

It is not a catastrophe, since his peers can spot his mistake by inspection of $\vdash$, but is none the less undesirable.

One way to remedy this problem is to disallow, also in Isabelle/HOL, constructions such as *Thm Falsity*. One way to do this is to define the axioms, rules and $\vdash$ on formulas instead of theorems. Hereafter one can define the type of theorems as the set of formulas derivable with $\vdash$ using Isabelle's *typedef* command. The lifting package of Isabelle can then lift the axioms and rules to work on this new type. The *concl* function will then be defined to convert theorems back to formulas.

With such a definition we can express soundness:

theorem *soundness*: *"semantics e f g* (*concl p*)*"*

And consistency:

theorem *consistency*: *"concl p* $\neq$ *Falsity"*

Another, similar, way to remedy the problem is to define a predicate characterizing the valid formulas and then define the theorems as the valid formulas using *typedef*. Unfortunately, as we noticed in Section 9.3, this is not possible. One way to overcome this is to introduce a new type U, using Isabelle's *typedecl* command, and then assume absolutely nothing about it. Then if a formula evaluates to true for all environments and interpretations over this universe, we can, informally, argue that it must be valid, since U is com-

```
prove
    (<!"(forall x y. Q(x,y) <=> forall z. P(z,x) <=> P(z,y)) ==> forall x y. Q(x,y) <=> Q(y,x)"!>)
    [
        assume [("A", <!"forall x y. Q(x,y) <=> forall z. P(z,x) <=> P(z,y)"!>)],
        conclude (<!"forall x y. Q(x,y) <=> Q(y,x)"!>) proof
        [
            fix "x", fix "y",
            conclude (<!"Q(x,y) <=> Q(y,x)"!>) proof
            [
                have (<!"(Q(x,y) ==> Q(y,x)) /\\ (Q(y,x) ==> Q(x,y))"!>) proof
                [
                    conclude (<!"Q(x,y) ==> Q(y,x)"!>) proof
                    [
                        assume [("", <!"Q(x,y)"!>)],
                        so have (<!"forall z. P(z,x) <=> P(z,y)"!>) by ["A"],
                        so have (<!"forall z. P(z,y) <=> P(z,x)"!>) at once,
                        so conclude (<!"Q(y,x)"!>) by ["A"],
                        qed
                    ],
                    conclude (<!"Q(y,x) ==> Q(x,y)"!>) proof
                    [
                        assume [("", <!"Q(y,x)"!>)],
                        so have (<!"forall z. P(z,y) <=> P(z,x)"!>) by ["A"],
                        so have (<!"forall z. P(z,x) <=> P(z,y)"!>) at once,
                        so conclude (<!"Q(x,y)"!>) by ["A"],
                        qed
                    ],
                    qed
                ],
                so our thesis at once,
                qed
            ],
            qed
        ],
        qed
    ]
```

Fig. 6. A detailed proof of Pelletier's problem 43 in the declarative prover.

pletely arbitrary. Again we can then define the rules and axioms on the type of formulas, and then lift them to work on theorems. With this approach soundness is captured in the types – any theorem value or function that returns a theorem is valid. Consistency can be expressed and proved in the same way as when we lifted $\vdash$. One can, however, argue that defining the theorems as the valid formulas goes against the meaning of the word theorem – theorem being a syntactic notion and validity being a semantic notion. In first-order logic the two words capture the same meaning for sound and complete proof systems, but for other logics such as ZFC there are no sound and complete proof systems with respect to their usual semantics, and thus the words have very distinct meanings.

We have implemented both approaches of having *thm* as a type. Their code is available online [19]. We, however, prefer our current approach because we feel that for teaching purposes there are already enough concepts to talk about and adding lifting to the mix might confuse more than help.

13. Related Work

The literature contains several other formalizations of logic and contains also declarative provers. Let us first look at some other formalizations of logic.

```
lemma "(∀x y. Q(x,y) ⟷ (∀z. P(z,x) ⟷ P(z,y))) ⟶ (∀x y. Q(x,y) ⟷ Q(y,x))"
proof
    assume A: "∀x y. Q(x,y) ⟷ (∀z. P(z,x) ⟷ P(z,y))"
    show "∀x y. Q(x,y) ⟷ Q(y,x)"
    proof (rule, rule)
        fix x y
        show "Q(x,y) ⟷ Q(y,x)"
        proof −
            have "(Q(x,y) ⟶ Q(y,x)) ∧ (Q(y,x) ⟶ Q(x,y))"
            proof
                show "Q(x,y) ⟶ Q(y,x)"
                proof
                    assume "Q(x,y)"
                    then have "∀z. P(z,x) ⟷ P(z,y)" using A by blast
                    then have "∀z. P(z,y) ⟷ P(z,x)" by blast
                    then show "Q(y,x)" using A by blast
                qed
            next
                show "Q(y,x) ⟶ Q(x,y)"
                proof
                    assume "Q(y,x)"
                    then have "∀z. P(z,y) ⟷ P(z,x)" using A by blast
                    then have "∀z. P(z,x) ⟷ P(z,y)" by blast
                    then show "Q(x,y)" using A by blast
                qed
            qed
            then show "Q(x,y) ⟷ Q(y,x)" by blast
        qed
    qed
qed
```

Fig. 7. A detailed proof of Pelletier's problem 43 in Isabelle/HOL.

Harrison [12] formalized, in HOL Light, soundness and consistency proofs for the HOL of HOL Light without definitions. More precisely he considered three different logics: HOL, HOL + I and HOL − ∞. HOL + I is HOL extended with an axiom claiming the existence of a very large cardinal, and HOL − ∞ is HOL where the axiom of infinity is removed. His results are to prove in HOL + I that HOL is sound and consistent, and to prove in HOL that HOL − ∞ is sound and consistent. Kumar et al. [20] extended Harrison's work by proving, in HOL4, that HOL with definitions is sound and consistent. Their proofs rely on assuming a specification of a set-theory. Our work differs from this by using the meta-logic of Isabelle/HOL and the object logic of FOL. Using Isabelle/HOL on the meta-level has at least two advantages seen from a teaching perspective. Firstly, Isabelle/HOL provides a complete integrated package of proof assistant, prover integrated development environment and code-generation. This enables students to load the entire development directly in Isabelle including verification, code-reflection and the execution of the prover. Secondly, having FOL on the object level has pedagogical advantages, since it is a logic that students are often familiar with and thus we can assume they have some understanding of its syntax and semantics. Thus, we see our development as a pedagogical stepping stone students can take towards the self-verifications of Harrison and Kumar et al.

Other provers based on verified proof systems for first-order logics are our NaDeA system [40] and Breitner's The Incredible Proof Machine [6]. They offer, by design, only limited automation and the connection between the verification and the implementation is, furthermore, entirely informal. Margetson and Ridge's automatic prover for first-order logic in negation normal form without first-order terms [24, 34, 35] makes the connection explicit, opting for execution within Isabelle/HOL's rewrite engine. Our prover stands out from these in two ways. Firstly, it is an interactive theorem prover where users can employ techniques of

declarative proving, automation, tactics, etc. as they wish. Secondly, the connection between the verification and the system is made explicit using code-generation.

There are many other formalizations of logic such as e.g. Persson's constructive completeness of intuitionistic predicate logic [33], Braselmann, Koepke and Schlöder's sequent calculus for uncountable languages [4, 5, 38], Berghofer's natural deduction [1], Ilik's constructive completeness results for classical and intuitionistic logic [15], Blanchette, Popescu and Traytel's abstract completeness library [3], Schlichtkrull's resolution calculus [36], Peltier's superposition calculus [32], and Paulson's proof of Gödel's incompleteness theorems [30]. These formalizations, however, do not formalize provers.

Let us now look at some other declarative provers. Geuvers [8] studies the history, ideas and future of proof assistants. For instance, he emphasizes the advantage of having declarative provers since they allow proofs in proof assistants to look like the texts that mathematicians write and understand. Furthermore he emphasizes that declarative proofs are easier to adapt when a definition is changed, since they explicitly document in each step which facts are supposed to hold there. He also gives an overview of declarative proofs in the proof assistants Mizar, Isabelle/Isar, Coq/C-Zar, and HOL Light (Mizar mode).

Our prover stands out from these provers in that it relies on a verified kernel. Furthermore, it is not meant as an advanced production scale proof assistant, but instead as a smaller program that is easy to understand and whose inner workings can be taught. None the less, the prover still has the advantages of being declarative that Geuvers described.

14. Conclusion

We have in Isabelle/HOL certified the soundness of the underlying axiomatic proof system of the declarative first-order prover. Using code reflection, we obtain from the proof system a certified kernel that is loaded into the Isabelle/ML environment. The declarative prover uses the certified kernel, and thus we also consider the soundness of the prover certified.

Declarative proofs mention explicitly the intermediate proof states, in contrast to procedural proofs that merely explain what method is used to go to the next state. We have given example proofs using the prover in Isabelle. Due to the compactness and transparent approach we think that the certified declarative prover is useful as a tool for teaching logic.

Many well-known theorems can be proved by full automation using the tableau prover, e.g. Pelletier's problems 1–46 except for problems 34 (also known as Andrews's challenge), 43 and 46.

For problem 43 and 46 that could not be proved automatically in reasonable time with the current tableau prover, we have shown how the proofs can be written as declarative proofs that resemble paper proofs and combine the declarative language with a high level of automation. It would be interesting to improve the tableau prover or to add, say, a resolution prover, which would be certified by using the certified kernel. We have not considered the tricky problem 34 yet.

Our declarative prover follows the LCF-style of having a trusted kernel on which other components are built. In a single Isabelle theory file we certify the soundness of the kernel and use code reflection to obtain a simple yet quite powerful interactive theorem prover. Our combination of derived rules, a tableau prover, tactics and a declarative prover opens up for easy experimentation with reliable combinations of automatic and interactive proof techniques — and such techniques are in high demand as the following quote suggests:

> *In view of the practical limitations of pure automation, it seems today that, whether one likes it or not, interactive proof is likely to be the only way to formalize most non-trivial theorems in mathematics or computer system correctness.* [14]

Learning the declarative style is of course beneficial for a computer science student who wants to use one of the aforementioned provers. Even for those who will never use a proof assistant again, it can be a helpful learning experience. Lamport recommends a structured style even for paper proofs [21, 22]. His experience is that this style helps reveal mistakes and cope with details. He also suggests using this style for teaching because it allows for additional explanation and has a clear logical structure that is easy to learn from. The concrete style he uses resembles very much that of our declarative prover. Furthermore, the style is implemented in the TLAPS prover [22]. We conjecture that a good way to learn this structured style is by studying and understanding a concrete prover. Our prover emphasizes the connection between the logical systems, its semantics and the prover that implements them. A student can study all these aspects in the package we provide.

Acknowledgements

We would like to thank Jasmin Christian Blanchette for valuable feedback on the workshop paper [17] on which parts of this paper are based. We would also like to thank Andrei Popescu for helpful discussions, and finally, we would like to thank Andreas Halkjær From and the anonymous referees for constructive comments on drafts of the paper.

References

[1] S. Berghofer. First-order logic according to Fitting. *Archive of Formal Proofs*, Aug. 2007. http://isa-afp.org/entries/FOL-Fitting.shtml, Formal proof development.

[2] J. C. Blanchette, S. Böhme, and L. C. Paulson. Extending Sledgehammer with SMT solvers. In N. Bjørner and V. Sofronie-Stokkermans, editors, *Automated Deduction – CADE-23: 23rd International Conference on Automated Deduction, Wrocław, Poland, July 31 - August 5, 2011. Proceedings*, pages 116–130, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[3] J. C. Blanchette, A. Popescu, and D. Traytel. Soundness and completeness proofs by coinductive methods. *Journal of Automated Reasoning*, 58(1):149–179, 2017.

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

[5] P. Braselmann and P. Koepke. Gödel completeness theorem. *Formalized Mathematics*, 13(1):49–53, 2005.

[6] J. Breitner. Visual theorem proving with the Incredible Proof Machine. In *International Conference on Interactive Theorem Proving*, pages 123–139. Springer, 2016.

[7] A. Church. *Introduction to Mathematical Logic*. Princeton: Princeton University Press, 1956.

[8] H. Geuvers. Proof assistants: History, ideas and future. *Sadhana*, 34(1):3–25, 2009.

[9] M. J. Gordon, A. J. Milner, and C. P. Wadsworth. *Edinburgh LCF – A Mechanised Logic of Computation*. LNCS. Springer, 1979.

[10] F. Haftmann and T. Nipkow. Code generation via higher-order rewrite systems. In M. Blume, N. Kobayashi, and G. Vidal, editors, *Functional and Logic Programming (FLOPS 2010)*, volume 6009 of *LNCS*, pages 103–117. Springer, 2010.

[11] R. Hähnle. Tableaux and related methods. In *Handbook of Automated Reasoning*, volume 1, pages 101–178. MIT Press, 2001.

[12] J. Harrison. Towards self-verification of HOL Light. In U. Furbach and N. Shankar, editors, *IJCAR 2006*, volume 4130 of *LNCS*, pages 177–191. Springer, 2006.

[13] J. Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, 2009.

[14] J. Harrison, J. Urban, and F. Wiedijk. History of interactive theorem proving. In J. Siekmann, editor, *Handbook of the History of Logic vol. 9 (Computational Logic)*, pages 135–214. Elsevier, 2014.

[15] D. Ilik. *Constructive Completeness Proofs and Delimited Control*. PhD thesis, École Polytechnique, 2010.

[16] A. B. Jensen. Development and verification of a proof assistant. Master's thesis, Technical University of Denmark, 2016.

[17] A. B. Jensen, A. Schlichtkrull, and J. Villadsen. Verification of an LCF-style first-order prover with equality. *Isabelle Workshop*, 2016.

[18] A. B. Jensen, A. Schlichtkrull, and J. Villadsen. First-order logic according to Harrison. *Archive of Formal Proofs*, Jan. 2017. ISSN 2150-914x. http://isa-afp.org/entries/FOL_Harrison.shtml,Formal proof development.

[19] A. B. Jensen, A. Schlichtkrull, and J. Villadsen. https://bitbucket.org/isafol/isafol/src/master/FOL_Harrison/, 2017. IsaFoL Entry – First-Order Logic According to Harrison.

[20] R. Kumar, R. Arthan, M. O. Myreen, and S. Owens. Self-formalisation of higher-order logic: Semantics, soundness, and a verified implementation. *Journal of Automated Reasoning*, 56(3):221–259, 2016.

[21] L. Lamport. How to write a proof. *Global Analysis in Modern Mathematics*, pages 311–321, 1993. Also published in *American Mathematical Monthly*, 102(7):600-608, August-September 1995.

[22] L. Lamport. How to write a 21st century proof. *Journal of fixed point theory and applications*, 11(1):43–63, 2012.

[23] R. Letz. First-order tableau methods. In M. D'Agostino, D. M. Gabbay, and R. Hähnle, editors, *Handbook of Tableau Methods*, pages 125–196. Kluwer Academic Publishers, 1999.

[24] J. Margetson and T. Ridge. Completeness theorem. *Archive of Formal Proofs*, Sept. 2004. http://isa-afp.org/entries/Completeness.shtml, Formal proof development.

[25] N. D. Megill. *Metamath: A Computer Language for Pure Mathematics*. Lulu Press, Morrisville, North Carolina, 2007. http://us.metamath.org/downloads/metamath.pdf.

[26] R. Milner. LCF: A way of doing proofs with a machine. In J. Bečvář, editor, *Mathematical Foundations of Computer Science 1979: Proceedings, 8th Symposium, Olomouc, Czechoslovakia, September 3–7, 1979*, pages 146–159. Springer Berlin Heidelberg, 1979.

[27] R. Milner, M. Tofte, and D. Macqueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997.

[28] J. Monk. *Mathematical Logic*. Graduate Texts in Mathematics. Springer New York, 1976. ISBN 9780387901701.

[29] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[30] L. C. Paulson. A mechanised proof of Gödel's incompleteness theorems using Nominal Isabelle. *Journal of Automated Reasoning*, 55(1):1–37, 2015.

[31] F. J. Pelletier. Seventy-five problems for testing automatic theorem provers. *Journal of Automated Reasoning*, 2(2):191–216, 1986.

[32] N. Peltier. A variant of the superposition calculus. *Archive of Formal Proofs*, Sept. 2016. http://isa-afp.org/entries/SuperCalc.shtml, Formal proof development.

[33] H. Persson. *Constructive completeness of intuitionistic predicate logic*. PhD thesis, Chalmers University of Technology, 1996.

[34] T. Ridge. A mechanically verified, efficient, sound and complete theorem prover for first order logic. *Archive of Formal Proofs*, Sept. 2004. http://isa-afp.org/entries/Verified-Prover.shtml, Formal proof development.

[35] T. Ridge and J. Margetson. A mechanically verified, sound and complete theorem prover for first order logic. In J. Hurd and T. Melham, editors, *TPHOL's 2005*, volume 3603 of *LNCS*, pages 294–309. Springer, 2005.

[36] A. Schlichtkrull. Formalization of the resolution calculus for first-order logic. In *International Conference on Interactive Theorem Proving*, pages 341–357. Springer, 2016.

[37] A. Schlichtkrull and J. Villadsen. https://github.com/logic-tools/sml-handbook/tree/master/code, 2017. README: SML version of code for John Harrison's "Handbook of Practical Logic and Automated Reasoning".

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

[39] A. Tarski. A simplified formalization of predicate logic with identity. *Journal of Symbolic Logic*, 39(3):602–603, 1974.

[40] J. Villadsen, A. B. Jensen, and A. Schlichtkrull. NaDeA: A natural deduction assistant with a formalization in Isabelle. *IfCoLog Journal of Logics and their Applications*, 4(1):55–82, 2017.

[41] M. Wenzel. Isar - A generic interpretative approach to readable formal proof documents. In *Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs'99, Nice, France, September, 1999, Proceedings*, pages 167–184, 1999.

[42] M. Wenzel. System description: Isabelle/jEdit in 2014. In *Proceedings Eleventh Workshop on User Interfaces for Theorem Provers, UITP 2014, Vienna, Austria, 17th July 2014*, pages 84–94, 2014.

[43] M. Wenzel. Isabelle/jEdit, 2017. http://isabelle.in.tum.de/dist/doc/jedit.pdf.

[44] M. Wenzel. The Isabelle/Isar reference manual, 2017. http://isabelle.in.tum.de/dist/doc/isar-ref.pdf.

[45] Wikipedia. Method of analytic tableaux, 2017. https://en.wikipedia.org/wiki/Method_of_analytic_tableaux.