Verification of an LCF-Style First-Order Prover with Equality

Alexander Birch Jensen, Anders Schlichtkrull, and Jørgen Villadsen

DTU Compute, Technical University of Denmark, 2800 Kongens Lyngby, Denmark

Abstract. We formalize in Isabelle/HOL the kernel of an LCF-style prover for first-order logic with equality from John Harrison's *Handbook of Practical Logic and Automated Reasoning*. We prove the kernel sound and generate Standard ML code from the formalization. The generated code can then serve as a verified kernel. By doing this we also obtain verified components such as derived rules, a tableau prover, tactics, and a small declarative interactive theorem prover. We test that the kernel and the components give the same results as Harrison's original on all the examples from his book. The formalization is 600 lines and is available online.

Keywords: Isabelle/HOL,verification, first-order logic, equality, soundness, LCF-style prover, OCaml, code generation, Standard ML (SML), Isabelle/ML

Quote from Alwen Tiu's review of John Harrison's *Handbook of Practical Logic and Automated Reasoning*:

> *This book is an extensive overview of automated reasoning methods for classical first-order logic. The author follows a rather unusual presentation style, where "pure logic and automated theorem proving are explained in a closely intertwined manner", and "automated theorem proving methods are explained with reference to actual concrete implementations ..." (page xi). The implementations are done in the functional programming language OCaml.*

(The Bulletin of Symbolic Logic, 16(2) p. 279 2010)

1 Introduction

In his *Handbook of Practical Logic and Automated Reasoning*, John Harrison presents a small LCF-style interactive theorem prover for first order-logic with equality [5]. The prover consists of a kernel and several other components such as derived rules, a tableau prover, tactics, and a small declarative interactive theorem prover whose proofs look similar to those of Mizar or Isar. We wish to teach these concepts to students, and we find that presenting them as components of a larger system is an excellent way to motivate them. By formalizing the kernel in Isabelle/HOL, we also point students towards self-verification studies such as Harrisons own verification of HOL Light [4] and the extension by Kumar, Arthan, Myreen and Owens [6]. It also gives us a chance to introduce students to formal verification and code generation.

An example of a proof by Harrison of the following theorem can be seen in Figure 1.

$$(\forall x\ y.\ x \le y \longleftrightarrow x * y = x) \wedge (\forall x\ y.\ f(x * y) = f(x) * f(y)) \\ \longrightarrow (\forall x\ y.\ x \le y \longrightarrow f(x) \le f(y))$$

```
let ewd954 = prove
 <<(forall x y. x <= y <=> x * y = x) /\
   (forall x y. f(x * y) = f(x) * f(y))
   ==> forall x y. x <= y ==> f(x) <= f(y)>>
 [note("eq_sym",<<forall x y. x = y ==> y = x>>)
    using [eq_sym <<|x|>> <<|y|>>];
  note("eq_trans",<<forall x y z. x = y /\ y = z ==> x = z>>)
    using [eq_trans <<|x|>> <<|y|>> <<|z|>>];
  note("eq_cong",<<forall x y. x = y ==> f(x) = f(y)>>)
    using [axiom_funcong "f" [<<|x|>>] [<<|y|>>]];
  assume ["le",<<forall x y. x <= y <=> x * y = x>>;
          "hom",<<forall x y. f(x * y) = f(x) * f(y)>>];
  fix "x"; fix "y";
  assume ["xy",<<x <= y>>];
  so have <<x * y = x>> by ["le"];
  so have <<f(x * y) = f(x)>> by ["eq_cong"];
  so have <<f(x) = f(x * y)>> by ["eq_sym"];
  so have <<f(x) = f(x) * f(y)>> by ["eq_trans"; "hom"];
  so have <<f(x) * f(y) = f(x)>> by ["eq_sym"];
  so conclude <<f(x) <= f(y)>> by ["le"];
  qed];;
```

Fig. 1. A declarative proof by Harrison in his LCF-style prover.

Harrison implements the kernel of his LCF-style prover as an OCaml program. Following the LCF style, he uses the kernel to build the other components. The benefit is that if the user trusts the kernel, then she can also trust the other components. For verification of the system there is a similar benefit. If we can verify the soundness of the kernel, then we have also verified the soundness of all the components. Thus, by making a verified kernel, we also obtain verified derived rules, a verified tableau prover, verified tactics, and a verified small declarative interactive theorem prover. This is the approach we will pursue by generating code for a formalization of the kernel.

Our formalization `Proven.thy` is named after Harrisons LCF-style kernel. It is available online [9]. The formalization is 600 lines including blank lines but excluding comments, and it takes 5 seconds for Isabelle to load. Available online is also the generated code `Proven.sml`, the same using opaque ascription `Proven-lcf.sml`, and files that load the program; the file `Proven-init.sml` can load it in Moscow ML and `Proven-init_nj.sml` can load it in Standard ML of New Jersey and Poly/ML .

This paper can be seen as a case study in the use of a proof assistant. We will explain how we used the different tools of the system to do the formalization. The code generator was obviously central to the project. We opted to use Isar to conduct the proofs since we want to obtain a humanly readable proof that students can study. Other tools we took advantage of were Isabelle/JEdit, the proof methods *auto*, *simp*, *fastforce* and *metis*, as well as the Sledgehammer tool which was especially helpful in dispensing of more complicated proof goals and finding relevant theorems from the libraries.

2 First-Order Logic

Our formalization of first-order logic is a straight forward translation of Harrison's datatype to Isabelle/HOL:

datatype *'a fm* =
  T | *F* | *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)*

Harrison used the names *True* and *False* where we instead use *T* and *F* because *True* and *False* are already used as the boolean values in Isabelle/HOL. We also formalize the terms and first-order atoms similarly to Harrison's datatypes:

datatype *tm* = *Var id* | *Fn id (tm list)*

datatype *fol* = *R id (tm list)*

3 Proof System and Kernel

The proof system can be seen in the appendix. It is based on systems by Tarski and others [7,11] and substitution is derivable. Harrison's implementation follows the LCF style. Therefore he defines a signature *Proofsystem* which abstractly defines the type of theorems and a number of constructors of theorems, corresponding to axioms and rules. The signature also contains *concl* which for a theorem gives the formula that expresses the theorem.

module type *Proofsystem* =
  sig type *thm*

```
    val modusponens : thm -> thm -> thm
    val gen : string -> thm -> thm
    val axiom_addimp : fol formula -> fol formula -> thm
    ...
    val axiom_exists : string -> fol formula -> thm
    val concl : thm -> fol formula
end;;
```

He then defines a structure *Proven* which is assigned the signature. It is assigned opaquely using OCaml's : operator, which means that the structure is assigned the *Proofsystem* signature exactly as it is written. The type *thm* is defined as *fol formula* and each of the constructors is an implementation of an axiom.

```
module Proven : Proofsystem =
  struct
    type thm = fol formula
    let modusponens pq p =
      match pq with
        Imp(p',q) when p = p' -> q
      | _ -> failwith "modusponens"
    let gen x p = Forall(x,p)
    let axiom_addimp p q = Imp(p,Imp(q,p))
    ...
    let axiom_exists x p = Iff(Exists(x,p),Not(Forall(x,Not p)))
    let concl c = c
end;;
```

The idea is that the only way to construct a value of type *thm* is to use the axioms. We will discuss our formalization of this kernel, how it differs from Harrison's, and why.

3.1 Type of theorems

For a theory file, Isabelle/HOL's code generator can also create a signature of the functions we specify and the types they use. It can also create a structure that implements the signature using the functions and types.

We thus need to introduce a type of theorems to the generated signature and structure. We call it *fol-thm* instead of *thm* because it is already used in Isabelle/HOL. A type synonym makes the code look similar to *Proven*.

```
type-synonym fol-thm = fol fm
```

This does not work, however, since it does not introduce a new type; it only introduces a synonym. We instead introduce *fol-thm* as a datatype containing a *fol formula*, and with constructor *Thm* and selector *concl*:

```
datatype fol-thm = Thm (concl: fol fm)
```

Harrison's system did not have this constructor *Thm*, however, it is not a problem because the implementation of the structure is hidden behind the signature. And the user will only construct theorems from the axioms and rules.

3.2 Exceptions

Harrison's implementation of the system uses exceptions when a rule is used on values that do not comply with the side conditions. For instance, his implementation of the rule *axiom−impall* gives an exception if the variable x is free in p.

$$\frac{\neg\,\text{free\_in}\ \ x\ p}{p \longrightarrow (\forall x.\, p)}$$

```
let axiom_impall x p =
  if not (free_in (Var x) p) then Imp(p,Forall(x,p))
  else failwith "axiom_impall: variable free in formula"
```

Since the logic of Isabelle/HOL does not have exceptions we cannot translate this directly. We therefore consider several alternatives. One would be to return *undefined*, because this value is code-generated as throwing an exception. Another would be to return a *fol fm option* which would be *None* when failing. A third solution could be to use an exception monad. We instead choose that the implementation returns the value T, in this case. This solution makes the code very simple. It also clearly preserves soundness since, when things go wrong, we return a formula that is obviously valid.

```
abbreviation (input) fail-thm ≡ Thm T

definition axiom-impall :: id ⇒ fol fm ⇒ fol-thm where
  axiom-impall x p ≡ if ¬free-in (Var x) p then Thm (Imp p (Forall x p)) else
fail-thm
```

3.3 Implications

Another change we make is in the implementation of *axiom-funcong*.

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

Harrison's implementation takes the 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";;
```

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

Our formalization instead splits the functionality of *axiom-funcong* in to two named functions. The first one is *zip-eq* which takes two lists of formulas, $[s_1, \ldots, s_n]$, $[t_1, \ldots, t_n]$ and builds the 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 r ≡ map (λ(u, v). mk-eq u v) (zip l r)
```

The second is *imp-chain*. This function takes a list of formulas $[F_1, \ldots, F_n]$ and adds them as antecedents to a formula F to build a nested implication $F_1 \longrightarrow \cdots \longrightarrow F_n \longrightarrow F$.

```
primrec
  imp-chain :: fol fm list ⇒ fol fm ⇒ fol fm
where
  imp-chain [] p = p |
  imp-chain (q # l) p = Imp q (imp-chain l p)
```

The idea of our approach is that we can reason about the two functions separately. With this approach, we can implement *axiom-funcong* as follows by first constructing the equalities, and then the nested implication.

```
definition axiom-funcong :: id ⇒ tm list ⇒ tm list ⇒ fol-thm where
  axiom-funcong i l r ≡ if length l = length r
     then Thm (imp-chain (zip-eq l r) (mk-eq (Fn i l) (Fn i r))) else fail-thm
```

We implement *axiom-predcong* in a similar way.

4 Semantics

To prove the rules sound, we of course need a semantics of terms and formulas. We introduce a semantics similar to that of Berghofer [1] and the NaDeA system [12]. The first major difference is that our semantics uses named variables instead of de Bruijn indices. The other major difference is that 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
$\quad$ *semantics-term* :: $(id \Rightarrow 'a) \Rightarrow (id \Rightarrow 'a\ list \Rightarrow 'a) \Rightarrow tm \Rightarrow 'a$ **and**
$\quad$ *semantics-list* :: $(id \Rightarrow 'a) \Rightarrow (id \Rightarrow 'a\ list \Rightarrow 'a) \Rightarrow tm\ list \Rightarrow 'a\ list$
where
$\quad$ *semantics-term* e - $(Var\ x) = e\ x$ |
$\quad$ *semantics-term* $e\ f\ (Fn\ i\ l) = f\ i$ (*semantics-list* $e\ f\ l$) |
$\quad$ *semantics-list* - - $[] = []$ |
$\quad$ *semantics-list* $e\ f\ (t \# l) =$ *semantics-term* $e\ f\ t\ \#$ *semantics-list* $e\ f\ l$

primrec
$\quad$ *semantics* :: $(id \Rightarrow 'a) \Rightarrow (id \Rightarrow 'a\ list \Rightarrow 'a) \Rightarrow (id \Rightarrow 'a\ list \Rightarrow bool) \Rightarrow$ *fol fm* $\Rightarrow bool$
where
$\quad$ *semantics* - - - $T = True$ |
$\quad$ *semantics* - - - $F = False$ |
$\quad$ *semantics* $e\ f\ g\ (Atom\ a) = (case\ a\ of\ R\ i\ l \Rightarrow if\ i = STR\ ''='' \wedge length\ l = 2$
$\quad\quad$ *then* (*semantics-term* $e\ f\ (hd\ l) =$ *semantics-term* $e\ f\ (hd\ (tl\ l))$)
$\quad\quad$ *else* $g\ i$ (*semantics-list* $e\ f\ l$)) |
$\quad$ *semantics* $e\ f\ g\ (Imp\ p\ q) =$ (*semantics* $e\ f\ g\ p \longrightarrow$ *semantics* $e\ f\ g\ q$) |
$\quad$ *semantics* $e\ f\ g\ (Iff\ p\ q) =$ (*semantics* $e\ f\ g\ p \longleftrightarrow$ *semantics* $e\ f\ g\ q$) |
$\quad$ *semantics* $e\ f\ g\ (And\ p\ q) =$ (*semantics* $e\ f\ g\ p \wedge$ *semantics* $e\ f\ g\ q$) |
$\quad$ *semantics* $e\ f\ g\ (Or\ p\ q) =$ (*semantics* $e\ f\ g\ p \vee$ *semantics* $e\ f\ g\ q$) |
$\quad$ *semantics* $e\ f\ g\ (Not\ p) = (\neg$*semantics* $e\ f\ g\ p$) |
$\quad$ *semantics* $e\ f\ g\ (Exists\ x\ p) = (\exists\, v.$ *semantics* $(e(x := v))\ f\ g\ p$) |
$\quad$ *semantics* $e\ f\ g\ (Forall\ x\ p) = (\forall\, v.$ *semantics* $(e(x := v))\ f\ g\ p$)

5 Soundness of Axioms

Harrison presents a soundness proof for the proof system. His proof is very high level and leaves a lot of the exercise up to the reader. Furthermore, his proof is about the proof system, not its implementation. Our approach is therefore to develop the proof ourselves, using Isabelle/jEdit to explore proofs and to help us reveal the necessary lemmas. Apply-style helped us explore proofs, but we have replaced all apply-style proofs with Isar-style proofs.

The axioms without preconditions are proven using only the automation of Isabelle/HOL. For instance our proof of *sem-axiom-addimp* is simply by *unfolding* and *simp*, and our proof of *sem-axiom-impiff* is by *unfolding* and *fastforce*.

The axioms with preconditions 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.

5.1 *axiom-impall* and *axiom-existseq*

The first challenge is in the soundness proof of *axiom-impall*. Here we need to prove that if a variable is not free or does not occur in an expression, then we can reassign it in an environment, and the expression will evaluate to the same.

lemma *map′*:
$\neg$*occurs-in* (*Var* x) $u \Longrightarrow$
semantics-term e f u = *semantics-term* ($e(x := v)$) f u
$\neg$*occurs-in-list* (*Var* x) $l \Longrightarrow$
semantics-list e f l = *semantics-list* ($e(x := v)$) f l

lemma *map*:
$\neg$*free-in* (*Var* x) $p \Longrightarrow$ *semantics* e f g p = *semantics* ($e(x := v)$) f g p

We then prove *axiom-impall* sound.

lemma *sem-axiom-impall*:
$\neg$*free-in* (*Var* x) $p \Longrightarrow$ *semantics* e f g (*concl* (*axiom-impall* x p))

Using *map′* we can also prove *axiom-existseq* sound.

lemma *sem-axiom-existseq*:
$\neg$*occurs-in* (*Var* x) $u \Longrightarrow$ *semantics* e f g (*concl* (*axiom-existseq* x u))

5.2 *sem-axiom-funcong*

The next challenge is to prove *sem-axiom-funcong* sound. We now take advantage of the *imp-chain* predicate we introduced earlier, and prove a lemma explaining its semantics. The lemma states that a nested implication is true exactly when either some antecedent is false or the conclusion is true.

lemma *sem-imp-chain*:
semantics e f g (*imp-chain* l p) =
(($\exists\, q \in$ *set* l. $\neg$*semantics* e f g q) $\vee$ *semantics* e f g p)

We then also state a lemma which (partially) explains what the semantics of *imp-chain* (*zip-eq* l r) p are. The lemma states that if l and r do not evaluate to the same, then the semantics hold.

lemma *sem-imp-chain-zip-eq*:
length l = *length* $r \Longrightarrow$ *semantics-list* e f $l \neq$ *semantics-list* e f $r \Longrightarrow$
semantics e f g (*imp-chain* (*zip-eq* l r) p)

We are now ready to prove the soundness of the axiom. We do it, respectively, for the case where *semantics-list* e f l = *semantics-list* e f r holds and where it does not. In the case where it holds soundness follows from *sem-imp-chain*, and in the case where it does not hold soundness follows from *sem-imp-chain-zip-eq*.

lemma *sem-axiom-funcong*:
length l = *length* $r \Longrightarrow$ *semantics* e f g (*concl* (*axiom-funcong* i l r))

5.3 *sem-axiom-predcong*

We also prove *sem-axiom-predcong* sound.

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

The proof is similar to that of *sem-axiom-funcong*. It gets a bit more complicated though because we also need to consider both the cases where the predicate in the conclusion is = and the cases where it is not. In the case where it is = we furthermore must consider when it takes two arguments, and when it does not.

lemma *sem-axiom-predcong*:
  length l = *length r* $\Longrightarrow$ *semantics e f g* (*concl* (*axiom-predcong i l r*))

6 Soundness of the Proof System

We have proven the axioms of the system sound. Our next step is to prove the whole system sound. We therefore first define it as an inductive predicate.

inductive
  OK :: *fol fm* $\Rightarrow$ *bool* ($\vdash$ - *0*)
where
  modusponens:
    $\vdash$ *concl pq* $\Longrightarrow$ $\vdash$ *concl p* $\Longrightarrow$ $\vdash$ *concl* (*modusponens pq p*) |
  gen:
    $\vdash$ *concl p* $\Longrightarrow$ $\vdash$ *concl* (*gen - p*) |
  axiom-addimp:
    $\vdash$ *concl* (*axiom-addimp - -*) |
  axiom-distribimp:
    $\vdash$ *concl* (*axiom-distribimp - - -*) |
  axiom-doubleneg:
    $\vdash$ *concl* (*axiom-doubleneg -*) |
  axiom-allimp:
    $\vdash$ *concl* (*axiom-allimp - - -*) |
  axiom-impall:
    $\vdash$ *concl* (*axiom-impall - -*) |
  axiom-existseq:
    $\vdash$ *concl* (*axiom-existseq - -*) |
  axiom-eqrefl:
    $\vdash$ *concl* (*axiom-eqrefl -*) |
  axiom-funcong:
    $\vdash$ *concl* (*axiom-funcong - - -*) |
  axiom-predcong:
    $\vdash$ *concl* (*axiom-predcong - - -*) |
  axiom-iffimp1:
    $\vdash$ *concl* (*axiom-iffimp1 - -*) |
  axiom-iffimp2:
    $\vdash$ *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* - -)

Then we prove it sound using rule induction. All the cases for the axioms are proven using the lemmas that proved them sound. The rules *gen* and *modusponens* are also proven easily with the help of some automation.

The inductive predicate defines a proof system. Another way to see it is that it formalizes all the theorems that can be built with the functions exposed by the *Proofsystem* signature as implemented in the *Proven* structure. Thus the soundness proof, in some sense, verifies that the values of type *fol_thm* are indeed theorems.

7 Code Generation

We previously manually translated all the components of Harrison's LCF-style prover from OCaml to the Standard ML [9]. We now use Isabelle/HOL's code generation to generate a signature and structure similar to Harrison's *Proofsystem* and *Proven*. We want this kernel to hook into the other components of the prover, and therefore we need to make sure that it uses the same constructors and type for the formulas. This is done by instructing the code generator to use them.

code-printing
  type-constructor *tm* ⇀ (*SML*) *term*
  | **constant** *Var* ⇀ (*SML*) *Var* -
  | **constant** *Fn* ⇀ (*SML*) *Fn* (-, -)

code-printing
  type-constructor *fm* ⇀ (*SML*) - *formula*
  | **constant** *T* ⇀ (*SML*) *True*
  | **constant** *F* ⇀ (*SML*) *False*
  | **constant** *Atom* ⇀ (*SML*) *Atom* -
  | **constant** *Imp* ⇀ (*SML*) *Imp* (-, -)
  | **constant** *Iff* ⇀ (*SML*) *Iff* (-, -)
  | **constant** *And* ⇀ (*SML*) *And* (-, -)
  | **constant** *Or* ⇀ (*SML*) *Or* (-, -)
  | **constant** *Not* ⇀ (*SML*) *Not* -
  | **constant** *Exists* ⇀ (*SML*) *Exists* (-, -)
  | **constant** *Forall* ⇀ (*SML*) *Forall* (-, -)

code-printing
  type-constructor *fol* ⇀ (*SML*) *fol*
  | **constant** *R* ⇀ (*SML*) *R* (-, -)

We then choose the appropriate functions to include in the structure and signature and generate them.

export-code
  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
  in *SML* **module-name** *Proven*

If we look at the generated code in the file `Proven.sml`, we see that it looks as follows:

structure *Proven* : **sig**
  type *nat*
  type *fol_thm*
  val *gen* : *string* −> *fol_thm* −> *fol_thm*
  val *axiom_or* : *fol formula* −> *fol formula* −> *fol_thm*
  val *axiom_and* : *fol formula* −> *fol formula* −> *fol_thm*
  ...
  val *axiom_distribimp*
    : *fol formula* −> *fol formula* −> *fol formula* −> *fol_thm*
end = **struct**
...
end; (∗**struct** *Proven*∗)

We notice that the generated code uses Standard ML's transparent ascription (:). For an LCF-style prover Standard ML's opaque ascription (:>) is preferable because it ensures that the signature of the structure is exactly the specified signature. This directly ensures that values of type *fol_thm* only can be created using the functions specified by the signature, i.e. the axioms and the rules.

However, even though our generated code uses transparent ascription, it still has that property. The reason is that we define *fol_thm* as a new data type inside the structure. Its constructor is not part of the signature, and thus the only way to create a value of this type is by using the axioms and the rules. However, if one made another structure where *fol_thm* was defined as

type *fol_thm* = *fol formula*

and one had used transparent ascription, then one would also have been able to build theorems with the constructors of *fol formula*.

Because of this, we choose to change : to :> by hand. It means that anyone can check that the prover follows the LCF style without having to reason about the structure. The changed code is in the file `Proven-lcf.sml`.

8 Testing

We have collected all the examples about the LCF-style prover and related functions and components from Harrison's book in a single OCaml file which we have manually translated to Standard ML. We then compare the result when using our code-generated kernel, with the result when using the manual translation to SML of the kernel. We also compare the result with Harrison's OCaml version. In all cases we get the same results.

We have also run both the manual translation of the kernel (`Init.thy`) and the code-generated kernel (`Proven-Init.thy`) in Isabelle which supports plain Standard ML via the *SML file* command. Here plain Standard ML shares the Poly/ML run-time system with Isabelle/ML used for proof development. We can export to the Isabelle/ML environment as shown in the following fragment which proves the classical tautology $(p \to q) \vee (q \to p)$:

```
SML-export {*
  val ex =
    let val p = Atom(R("p",[]))
        val q = Atom(R("q",[]))
    in
      concl (lcftaut (Or(Imp(p,q),Imp(q,p))))
    end
*}

ML {* ex *}
```

In the fragment the tautology checker *lcftaut* uses the manual translation of the kernel (if the fragment is in `Init.thy`) or the code-generated kernel (if the fragment is in `Proven-Init.thy`). We also in a few cases import from the Isabelle/ML environment in order to change the plain Standard ML setup such that it is possible to run the examples given the way the files are set up.

9 Timing

We also make some informal measurements of the time it took to run all the examples as mentioned in the previous section:

- OCaml
  - John Harrison's kernel: About 20 seconds
- Moscow ML
  - Code-generated kernel: About 60 seconds
  - Manual translation of John Harrison's kernel: About 15 seconds
- Isabelle (Standard ML)
  - Code-generated kernel: About 8 seconds
  - Manual translation of John Harrison's kernel: About 2 seconds

The measurements in Isabelle are done both with the timing panel and on a real clock. As it can be seen the code-generated kernel is not up to speed with the manual translation of the kernel on the same system.

The code generator generates syntactic equality of first-order formulas in a rather elaborate way. We have made some initial experiments where we generate it as Standard ML's = instead. It seems that this gets it up to the speed of the manual translation of kernel, but more tests and experiments are needed.

10 Related Work

The most similar effort is Harrison's verification of HOL Light which is an LCF-style prover for higher-order logic [4]. This effort was extended by Kumar, Arthan, Myreen and Owens [6]. If we look at proof systems in general, there are several formalizations in Isabelle/HOL and other systems. For overviews we refer to other papers [2, 10]. New efforts not covered there are Peltier's formalization of propositional resolution [8] and Breitner and Lohner's formalization of The Incredible Proof Machine [3].

11 Conclusion and Future Work

We have formalized in Isabelle/HOL the kernel of Harrison's LCF-style prover. The formalization is proven to be sound. From the formalization is code-generated a Standard ML-version which can be used together with a manual translation of the rest of the prover. Thus we not only obtain a verified kernel, but also verified derived rules, a verified tableau prover, verified tactics, and a verified small declarative interactive theorem prover.

The code-generated kernel and the components that depend on it give the same result on the examples of the book as a manual translation of the kernel as well as the original OCaml version by Harrison.

We currently generate Standard ML-code only. We would also like to generate OCaml-code to see if it is possible to replace Harrison's original kernel with a verified version.

Finally we would like to investigate further the consequences of using = for syntactic equality which seems to make the code-generated kernel described in this paper as fast as our manual translation of kernel.

Acknowledgement. Thanks to Jasmin Christian Blanchette for comments on a draft of the paper. Also thanks to Andreas Halkjær From for discussions.

Appendix: Proof System

modus ponens	$\dfrac{p \longrightarrow q \qquad p}{q}$
generalization	$\dfrac{p}{\forall x.\, p}$
axiom addimp	$\dfrac{}{p \longrightarrow q \longrightarrow p}$
axiom distribimp	$\dfrac{}{(p \longrightarrow q \longrightarrow r) \longrightarrow (p \longrightarrow q) \longrightarrow p \longrightarrow r}$
axiom doubleneg	$\dfrac{}{((p \longrightarrow \bot) \longrightarrow \bot) \longrightarrow p}$
axiom allimp	$\dfrac{}{(\forall x.\, p \longrightarrow q) \longrightarrow (\forall x.\, p) \longrightarrow (\forall x.\, q)}$
axiom impall	$\dfrac{\neg\, \text{free\_in}\ \ x\ p}{p \longrightarrow (\forall x.\, p)}$
axiom existseq	$\dfrac{\neg\, \text{occurs\_in}\ \ x\ t}{\exists x.\, x = t}$
axiom eqrefl	$\dfrac{}{t = t}$
axiom funcong	$\dfrac{}{s_1 = t_1 \longrightarrow \cdots \longrightarrow s_n = t_n \longrightarrow f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n)}$
axiom predcong	$\dfrac{}{s_1 = t_1 \longrightarrow \cdots \longrightarrow s_n = t_n \longrightarrow P(s_1, \ldots, s_n) \longrightarrow P(t_1, \ldots, t_n)}$
axiom iffimp1	$\dfrac{}{(p \longleftrightarrow q) \longrightarrow p \longrightarrow q}$
axiom iffimp2	$\dfrac{}{(p \longleftrightarrow q) \longrightarrow q \longrightarrow p}$
axiom impiff	$\dfrac{}{(p \longrightarrow q) \longrightarrow (q \longrightarrow p) \longrightarrow (p \longleftrightarrow q)}$
axiom true	$\dfrac{}{\top \longleftrightarrow (\bot \longrightarrow \bot)}$
axiom not	$\dfrac{}{\neg p \longleftrightarrow (p \longrightarrow \bot)}$
axiom and	$\dfrac{}{(p \wedge q) \longleftrightarrow ((p \longrightarrow q \longrightarrow \bot) \longrightarrow \bot)}$
axiom or	$\dfrac{}{(p \vee q) \longleftrightarrow \neg(\neg p \wedge \neg q)}$
axiom exists	$\dfrac{}{(\exists x.\, p) \longleftrightarrow \neg(\forall x.\, \neg p)}$

References

1. Berghofer, S.: First-order logic according to Fitting. Archive of Formal Proofs (Aug 2007), `http://isa-afp.org/entries/FOL-Fitting.shtml`, Formal proof development
2. Blanchette, J.C., Popescu, A., Traytel, D.: Unified classical logic completeness: A coinductive pearl. In: Demri, S., Kapur, D., Weidenbach, C. (eds.) IJCAR 2014. LNCS, vol. 8562, pp. 46–60. Springer (2014)
3. Breitner, J., Lohner, D.: The meta theory of the incredible proof machine. Archive of Formal Proofs (May 2016), `http://isa-afp.org/entries/Incredible_Proof_Machine.shtml`, Formal proof development
4. Harrison, J.: Towards self-verification of HOL Light. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS, vol. 4130, pp. 177–191. Springer (2006)
5. Harrison, J.: Handbook of Practical Logic and Automated Reasoning. Cambridge University Press (2009)
6. Kumar, R., Arthan, R., Myreen, M.O., Owens, S.: Self-formalisation of higher-order logic: Semantics, soundness, and a verified implementation. Journal of Automated Reasoning 56(3), 221–259 (2016)
7. Monk, J.D.: Mathematical Logic, Graduate Texts in Mathematics, vol. 37. Springer (1976)
8. Peltier, N.: Propositional resolution and prime implicates generation. Archive of Formal Proofs (Mar 2016), `http://isa-afp.org/entries/PropResPI.shtml`, Formal proof development
9. Schlichtkrull, A., Jensen, A.B., Villadsen, J.: SML code for Handbook of Practical Logic and Automated Reasoning - For Isabelle too. `https://github.com/logic-tools/sml-handbook/tree/master/code/SML`
10. Schlichtkrull, A.: Formalization of the resolution calculus for first-order logic. In: Blanchette, J.C., Merz, S. (eds.) ITP 2016. LNCS, vol. 9807. Springer (2016)
11. Tarski, A.: A simplified formalization of predicate logic with identity. Archiv für mathematische Logik und Grundlagenforschung 7, 61–79 (1965)
12. Villadsen, J., Jensen, A.B., Schlichtkrull, A.: NaDeA: A natural deduction assistant with a formalization in Isabelle. `https://nadea.compute.dtu.dk`