Code Generation for a Simple First-Order Prover

Villadsen, Jørgen; Schlichtkrull, Anders; Halkjær From, Andreas

Published in:
Proceedings of the Isabelle Workshop 2016

Publication date:
2016

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

Code Generation for a Simple First-Order Prover

Jørgen Villadsen, Anders Schlichtkrull, and Andreas Halkjær From

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

Abstract. We present Standard ML code generation in Isabelle/HOL of a sound and complete prover for first-order logic, taking formalizations by Tom Ridge and others as the starting point. We also define a set of so-called unfolding rules and show how to use these as a simple prover, with the aim of using the approach for teaching logic and verification to computer science students at the bachelor level.

Keywords: Isabelle/HOL, first-order logic, soundness, completeness, code generation, unfolding rules, Standard ML, Isabelle/ML

1 Introduction

We present code generation in Isabelle/HOL of a simple prover for first-order logic. We consider only Standard ML as the target programming language but OCaml, Haskell and Scala are possible too. Our work is based on Tom Ridge's entry in the Archive of Formal Proofs [6] which is building on James Margetson's formalization of work by Wainer and Wallen. Tom Ridge considers a manual translation to OCaml only. The entry also contains a recent manual translation to Standard ML by Jørgen Villadsen.

In addition to code generation we have reworked and simplified the formalization using Isar proofs instead of apply style. In particular the primitive recursive functions have been totally reworked. Our motivation is to use the approach for teaching logic and verification to computer science students at the bachelor level. However, the formalization needs further polishing, in particular where the original proofs are rather complicated — and in a few places the original proofs even have the "FIXME" comment. The formalization with a couple of examples and extra features is about 1800 lines including blank lines but excluding comments. All in all it takes less than 15 seconds to load on a fairly standard computer and is available online [9].

We first briefly describe our formalization of the proof system up to the soundness and completeness of the prover. We then explain the code generation in Standard ML and also the elegant code reflection feature using Isabelle/ML. Finally we define a set of so-called unfolding rules and show how to use these as a simple prover.

2 Proof System

We consider the same proof system as Tom Ridge [6] with a sequent as a list of formulas in negation normal form (nnf) and with no constants or functions:

$$\frac{}{\vdash P(v_{i_1},...,v_{i_k}), \Gamma, \overline{P}(v_{i_1},...,v_{i_k}), \Delta}\ Ax$$

Leaf of the derivation tree.

$$\frac{}{\vdash \overline{P}(v_{i_1},...,v_{i_k}), \Gamma, P(v_{i_1},...,v_{i_k}), \Delta}\ \overline{Ax}$$

Leaf of the derivation tree.

$$\frac{\vdash \Gamma, P(v_{i_1},...,v_{i_k})}{\vdash P(v_{i_1},...,v_{i_k}), \Gamma}\ NoAx$$

$$\frac{\vdash \Gamma, \overline{P}(v_{i_1},...,v_{i_k})}{\vdash \overline{P}(v_{i_1},...,v_{i_k}), \Gamma}\ \overline{NoAx}$$

$$\frac{\vdash \Gamma, A, B}{\vdash A \vee B, \Gamma}\ \vee$$

$$\frac{\vdash \Gamma, A \qquad \vdash \Gamma, B}{\vdash A \wedge B, \Gamma}\ \wedge$$

The only branching rule.

$$\frac{\vdash \Gamma, [v_i/x]A, (\exists x.A)^{i+1}}{\vdash (\exists x.A)^i, \Gamma}\ \exists$$

Superscripts are only relevant for this rule, and allow $[v_i/x]A$ to be instantiated for all i.

$$\frac{\vdash \Gamma, [v_r/x]A}{\vdash \forall x.A, \Gamma}\ \forall$$

v_r is a fresh free variable, chosen as $r = max(S)+1$, where S is the set of subscripts already used for the free variables in A ($r = 0$ if there are no free variables in A).

3 Example

Formulas are in negation normal form (signed predicates and de Bruijn indices):

```
datatype nnf =
  Pre bool id (nat list) |
  Con nnf nnf |
  Dis nnf nnf |
  Uni nnf |
  Exi nnf
```

As a small test example we consider the following classically valid formula:

```
proposition (∀ x. ¬ P x ∧ ¬ Q x) ∨ (∃ x. Q x) ∨ (∃ x. P x)
```

We take the predicate *id* as a type synonym for *nat* and define the test example:

```
abbreviation (input) P-id ≡ 0

abbreviation (input) Q-id ≡ Suc 0

definition
  test ≡ Dis
    (Uni (Con (Pre False P-id [0]) (Pre False Q-id [0])))
    (Dis (Exi (Pre True Q-id [0])) (Exi (Pre True P-id [0])))
```

Using code generation we run the prover on the test example (*True* returned):

```
value check test
```

4 Formalization of Proof System

We define a function *fv* returning the free variables in a formula:

```
primrec extend :: nat list ⇒ nat ⇒ nat list where
  extend l 0 = l |
  extend l (Suc n) = n # l

primrec adjust :: nat list ⇒ nat list where
  adjust [] = [] |
  adjust (h # t) = extend (adjust t) h

primrec fv :: nnf ⇒ nat list where
  fv (Pre - - v) = v |
  fv (Con p q) = fv p @ fv q |
  fv (Dis p q) = fv p @ fv q |
  fv (Uni p) = adjust (fv p) |
  fv (Exi p) = adjust (fv p)
```

We define a function *sv* for substitution in a formula:

```
primrec increase :: (nat ⇒ nat) ⇒ nat ⇒ nat where
  increase - 0 = 0 |
  increase f (Suc n) = Suc (f n)

primrec sv :: (nat ⇒ nat) ⇒ nnf ⇒ nnf where
  sv f (Pre b i v) = Pre b i (map f v) |
  sv f (Con p q) = Con (sv f p) (sv f q) |
  sv f (Dis p q) = Dis (sv f p) (sv f q) |
  sv f (Uni p) = Uni (sv (increase f) p) |
  sv f (Exi p) = Exi (sv (increase f) p)
```

We define a function *inst* for instantiation in a formula:

```
primrec bind :: nat ⇒ nat ⇒ nat where
  bind x 0 = x |
  bind - (Suc n) = n

definition inst :: nnf ⇒ nat ⇒ nnf where
  inst p x ≡ sv (bind x) p
```

We define a function *fresh* returning a new variable given a list of used variables:

```
primrec dec :: nat ⇒ nat where
  dec 0 = 0 |
  dec (Suc n) = n

primrec sub :: nat ⇒ nat ⇒ nat where
  sub x 0 = x |
  sub x (Suc n) = dec (sub x n)

primrec add :: nat ⇒ nat ⇒ nat where
  add x 0 = x |
  add x (Suc n) = Suc (add x n)

primrec fresh :: nat list ⇒ nat where
  fresh [] = 0 |
  fresh (h # t) = Suc (add (sub (dec (fresh t)) h) h)
```

We define the auxiliary functions *stop* and *maps*:

```
primrec stop :: 'a list ⇒ 'b ⇒ 'b list ⇒ 'a list where
  stop c - [] = c |
  stop c p (h # t) = (if p = h then [] else stop c p t)

definition maps :: ('a ⇒ 'b list) ⇒ 'a list ⇒ 'b list where
  maps f l ≡ concat (map f l)
```

We define the main functions *track* and *solve* on sequents:

```
type-synonym sequent = (nat × nnf) list
```

```
primrec track :: sequent ⇒ nat ⇒ nnf ⇒ sequent list where
  track s - (Pre b i v) = stop [s @ [(0,Pre b i v)]] (Pre (¬ b) i v) (map snd s)|
  track s - (Con p q) = [s @ [(0,p)],s @ [(0,q)]] |
  track s - (Dis p q) = [s @ [(0,p),(0,q)]] |
  track s - (Uni p) = [s @ [(0,inst p (fresh (maps fv (Uni p # map snd s))))]]|
  track s n (Exi p) = [s @ [(0,inst p n),(Suc n,Exi p)]]

primrec solve :: sequent ⇒ sequent list where
  solve [] = [[]] |
  solve (h # t) = track t (fst h) (snd h)
```

We separate the prover algorithm in three parts (see function *main* below):

```
type-synonym 'a algorithm = ('a ⇒ bool) ⇒ ('a ⇒ 'a) ⇒ 'a ⇒ bool
```

The function *null* terminates the prover algorithm:

```
primrec null :: 'a list ⇒ bool where
  null [] = True |
  null (- # -) = False

definition main :: sequent list algorithm ⇒ nnf ⇒ bool where
  main a p ≡ a null (maps solve) [[(0,p)]]
```

Note the existential quantifier in the *iterator* function:

```
primrec repeat :: ('a ⇒ 'a) ⇒ 'a ⇒ nat ⇒ 'a where
  repeat - c 0 = c |
  repeat f c (Suc n) = repeat f (f c) n

definition iterator :: 'a algorithm where
  iterator g f c ≡ ∃ n. g (repeat f c n)
```

Finally we obtain the prover itself:

```
definition check :: nnf ⇒ bool where
  check ≡ main iterator
```

5 Soundness & Completeness

For the semantics we consider only a countable universe using unit lists:

```
type-synonym proxy = unit list

type-synonym model = proxy set × (id ⇒ proxy list ⇒ bool)

type-synonym environment = nat ⇒ proxy

definition is-model-environment :: model ⇒ environment ⇒ bool where
  is-model-environment m e ≡ ∀ n. e n ∈ fst m
```

```
primrec semantics :: model ⇒ environment ⇒ nnf ⇒ bool where
  semantics m e (Pre b i v) = (b = snd m i (map e v)) |
  semantics m e (Con p q) = (semantics m e p ∧ semantics m e q) |
  semantics m e (Dis p q) = (semantics m e p ∨ semantics m e q) |
  semantics m e (Uni p) = (∀ z ∈ fst m.
                          semantics m (λx. case x of 0 ⇒ z | Suc n ⇒ e n) p)|
  semantics m e (Exi p) = (∃ z ∈ fst m.
                           semantics m (λx. case x of 0 ⇒ z | Suc n ⇒ e n) p)
```

The omitted soundness and completeness proof is about one thousand lines:

```
theorem check p = (∀ m e. is-model-environment m e ⟶ semantics m e p)
```

6 Code Generation

We must first prove a suitable "code" lemma for the *iterator* function:

```
lemma iterator[code]: iterator g f c = (if g c then True else iterator g f (f c))
```

Then the code generation is possible:

```
proposition check test
```

The simplifier can use the code equations of the underlying program:

```
  by code_simp
```

Testing using the code generator:

```
  by eval
```

Testing using normalization by evaluation (nbe):

```
  by normalization
```

Partially symbolic evaluation is possible with normalization by evaluation:

```
proposition check (Dis (Pre b i v) (Pre (¬ b) i v))
```

Finally we export the program to Standard ML (the name of a file can be added):

```
export-code check test in SML module-name SimPro
```

7 Code Reflection

The program can alternatively be compiled into the system run-time (Isabelle/ML):

```
code-reflect X datatypes nnf = Pre | Con | Dis | Uni | Exi
  and nat = 0::nat | Suc
  functions check
```

Here is the test example again:

ML {*

val true = X.check (
  X.Dis (X.Uni (X.Con (
    X.Pre (false,X.Zero_nat,[X.Zero_nat]),
    X.Pre (false,X.Suc X.Zero_nat,[X.Zero_nat]))),
  X.Dis (
    X.Exi (X.Pre (true,X.Suc X.Zero_nat,[X.Zero_nat])),
    X.Exi (X.Pre (true,X.Zero_nat,[X.Zero_nat])))))

*}

8 Unfolding Rules

In the appendix we list the so-called unfolding rules using the shorthand:

abbreviation (*input*) *PROVER* ≡ *iterator null* (*maps solve*)

The following simplification rules constitute the program (*check*):

theorem *program*:
⋀*p*. *check p* ≡ *PROVER* [[(*0*,*p*)]]
⋀*h t*. *PROVER* (*h* # *t*) ≡ *PROVER* (*maps solve* (*h* # *t*))
PROVER [] ≡ *True*
solve [] ≡ [[]]
⋀*h t*. *solve* (*h* # *t*) ≡ *track t* (*fst h*) (*snd h*)
⋀*s n b i v*. *track s n* (*Pre b i v*) ≡ *stop* [*s* @ [(*0*,*Pre b i v*)]] (*Pre* (¬ *b*) *i v*) (*map snd s*)
⋀*s n p q*. *track s n* (*Con p q*) ≡ [*s* @ [(*0*,*p*)],*s* @ [(*0*,*q*)]]
⋀*s n p q*. *track s n* (*Dis p q*) ≡ [*s* @ [(*0*,*p*),(*0*,*q*)]]
⋀*s n p*. *track s n* (*Uni p*) ≡ [*s* @ [(*0*,*inst p* (*fresh* (*maps fv* (*Uni p* # *map snd s*))))]]
⋀*s n p*. *track s n* (*Exi p*) ≡ [*s* @ [(*0*,*inst p n*),(*Suc n*,*Exi p*)]]
⋀*f l*. *maps f l* ≡ *concat* (*map f l*)
⋀*c p*. *stop c p* [] ≡ *c*
⋀*c p h t*. *stop c p* (*h* # *t*) ≡ (*if p* = *h then* [] *else stop c p t*)
fresh [] ≡ *0*
⋀*h t*. *fresh* (*h* # *t*) ≡ *Suc* (*add* (*sub* (*dec* (*fresh t*)) *h*) *h*)
⋀*x*. *add x 0* ≡ *x*
⋀*x n*. *add x* (*Suc n*) ≡ *Suc* (*add x n*)
⋀*x*. *sub x 0* ≡ *x*
⋀*x n*. *sub x* (*Suc n*) ≡ *dec* (*sub x n*)
dec 0 ≡ *0*
⋀*n*. *dec* (*Suc n*) ≡ *n*

$\bigwedge p\ x.\ inst\ p\ x \equiv sv\ (bind\ x)\ p$
$\bigwedge x.\ bind\ x\ 0 \equiv x$
$\bigwedge x\ n.\ bind\ x\ (Suc\ n) \equiv n$
$\bigwedge f\ b\ i\ v.\ sv\ f\ (Pre\ b\ i\ v) \equiv Pre\ b\ i\ (map\ f\ v)$
$\bigwedge f\ p\ q.\ sv\ f\ (Con\ p\ q) \equiv Con\ (sv\ f\ p)\ (sv\ f\ q)$
$\bigwedge f\ p\ q.\ sv\ f\ (Dis\ p\ q) \equiv Dis\ (sv\ f\ p)\ (sv\ f\ q)$
$\bigwedge f\ p.\ sv\ f\ (Uni\ p) \equiv Uni\ (sv\ (increase\ f)\ p)$
$\bigwedge f\ p.\ sv\ f\ (Exi\ p) \equiv Exi\ (sv\ (increase\ f)\ p)$
$\bigwedge f.\ increase\ f\ 0 \equiv 0$
$\bigwedge f\ n.\ increase\ f\ (Suc\ n) \equiv Suc\ (f\ n)$
$\bigwedge b\ i\ v.\ fv\ (Pre\ b\ i\ v) \equiv v$
$\bigwedge p\ q.\ fv\ (Con\ p\ q) \equiv fv\ p\ @\ fv\ q$
$\bigwedge p\ q.\ fv\ (Dis\ p\ q) \equiv fv\ p\ @\ fv\ q$
$\bigwedge p.\ fv\ (Uni\ p) \equiv adjust\ (fv\ p)$
$\bigwedge p.\ fv\ (Exi\ p) \equiv adjust\ (fv\ p)$
$adjust\ [] \equiv []$
$\bigwedge h\ t.\ adjust\ (h\ \#\ t) \equiv extend\ (adjust\ t)\ h$
$\bigwedge l.\ extend\ l\ 0 \equiv l$
$\bigwedge l\ n.\ extend\ l\ (Suc\ n) \equiv n\ \#\ l$

The following simplification rules depends on the datatype (*nnf*):

theorem *data*:
$\bigwedge b\ i\ v\ p\ q.\ Pre\ b\ i\ v = Con\ p\ q \equiv False$
$\bigwedge b\ i\ v\ p\ q.\ Con\ p\ q = Pre\ b\ i\ v \equiv False$
$\bigwedge b\ i\ v\ p\ q.\ Pre\ b\ i\ v = Dis\ p\ q \equiv False$
$\bigwedge b\ i\ v\ p\ q.\ Dis\ p\ q = Pre\ b\ i\ v \equiv False$
$\bigwedge b\ i\ v\ p.\ Pre\ b\ i\ v = Uni\ p \equiv False$
$\bigwedge b\ i\ v\ p.\ Uni\ p = Pre\ b\ i\ v \equiv False$
$\bigwedge b\ i\ v\ p.\ Pre\ b\ i\ v = Exi\ p \equiv False$
$\bigwedge b\ i\ v\ p.\ Exi\ p = Pre\ b\ i\ v \equiv False$
$\bigwedge p\ q\ p'\ q'.\ Con\ p\ q = Dis\ p'\ q' \equiv False$
$\bigwedge p\ q\ p'\ q'.\ Dis\ p'\ q' = Con\ p\ q \equiv False$
$\bigwedge p\ q\ p'.\ Con\ p\ q = Uni\ p' \equiv False$
$\bigwedge p\ q\ p'.\ Uni\ p' = Con\ p\ q \equiv False$
$\bigwedge p\ q\ p'.\ Con\ p\ q = Exi\ p' \equiv False$
$\bigwedge p\ q\ p'.\ Exi\ p' = Con\ p\ q \equiv False$
$\bigwedge p\ q\ p'.\ Dis\ p\ q = Uni\ p' \equiv False$
$\bigwedge p\ q\ p'.\ Uni\ p' = Dis\ p\ q \equiv False$
$\bigwedge p\ q\ p'.\ Dis\ p\ q = Exi\ p' \equiv False$
$\bigwedge p\ q\ p'.\ Exi\ p' = Dis\ p\ q \equiv False$
$\bigwedge p\ p'.\ Uni\ p = Exi\ p' \equiv False$
$\bigwedge p\ p'.\ Exi\ p' = Uni\ p \equiv False$
$\bigwedge b\ i\ v\ b'\ i'\ v'.\ Pre\ b\ i\ v = Pre\ b'\ i'\ v' \equiv b = b' \wedge i = i' \wedge v = v'$
$\bigwedge p\ q\ p'\ q'.\ Con\ p\ q = Con\ p'\ q' \equiv p = p' \wedge q = q'$
$\bigwedge p\ q\ p'\ q'.\ Dis\ p\ q = Dis\ p'\ q' \equiv p = p' \wedge q = q'$
$\bigwedge p\ p'.\ Uni\ p = Uni\ p' \equiv p = p'$
$\bigwedge p\ p'.\ Exi\ p = Exi\ p' \equiv p = p'$

The following simplification rules provide a functional programming language:

theorem *library*:
$\bigwedge f.\ map\ f\ [] \equiv []$
$\bigwedge f\ h\ t.\ map\ f\ (h\ \#\ t) \equiv f\ h\ \#\ map\ f\ t$
$concat\ [] \equiv []$
$\bigwedge h\ t\ .\ concat\ (h\ \#\ t) \equiv h\ @\ concat\ t$
$\bigwedge l.\ []\ @\ l \equiv l$
$\bigwedge h\ t\ l.\ (h\ \#\ t)\ @\ l \equiv h\ \#\ t\ @\ l$
$\bigwedge x\ y.\ if\ True\ then\ x\ else\ y \equiv x$
$\bigwedge x\ y.\ if\ False\ then\ x\ else\ y \equiv y$
$\neg\ True \equiv False$
$\neg\ False \equiv True$
$\bigwedge b.\ \neg\ \neg\ b \equiv b$
$\bigwedge x\ y.\ fst\ (x,y) \equiv x$
$\bigwedge x\ y.\ snd\ (x,y) \equiv y$
$\bigwedge n.\ 0 = Suc\ n \equiv False$
$\bigwedge n.\ Suc\ n = 0 \equiv False$
$\bigwedge h\ t.\ [] = h\ \#\ t \equiv False$
$\bigwedge h\ t.\ h\ \#\ t = [] \equiv False$
$True = False \equiv False$
$False = True \equiv False$
$0 = 0 \equiv True$
$[] = [] \equiv True$
$True = True \equiv True$
$False = False \equiv True$
$\bigwedge x\ y\ x'\ y'.\ (x,y) = (x',y') \equiv x = x' \wedge y = y'$
$\bigwedge n\ n'.\ Suc\ n = Suc\ n' \equiv n = n'$
$\bigwedge h\ t\ h'\ t'.\ h\ \#\ t = h'\ \#\ t' \equiv h = h' \wedge t = t'$
$\bigwedge b.\ True \wedge b \equiv b$
$\bigwedge b.\ False \wedge b \equiv False$

Let us consider the test example (without the *test* abbreviation):

lemma *check*
(*Dis* (*Uni* (*Con* (*Pre False 0* [*0*]) (*Pre False* (*Suc 0*) [*0*])))
(*Dis* (*Exi* (*Pre True* (*Suc 0*) [*0*])) (*Exi* (*Pre True 0* [*0*])))))

We use just the unfolding rules:

unfolding program data library
by (rule TrueI)

We can even "single-step" it (here with the *test* abbreviation):

proposition *check test*

unfolding test_def
unfolding program(1)
unfolding program(2)
unfolding program(3−) data library
unfolding program(2)
unfolding program(3−) data library
unfolding program(2)
unfolding program(3−) data library
unfolding program(2)
unfolding program(3−) data library
unfolding program(2)
unfolding program(3−) data library
unfolding program(2)
unfolding program(3−) data library
unfolding program(2)
unfolding program(3−) data library
by (rule TrueI)

Here are the results for the main steps (using the *PROVER* abbreviation):

check test

check
(Dis (Uni (Con (Pre False 0 [0]) (Pre False (Suc 0) [0])))
(Dis (Exi (Pre True (Suc 0) [0])) (Exi (Pre True 0 [0])))

PROVER
[[(0, Dis (Uni (Con (Pre False 0 [0]) (Pre False (Suc 0) [0])))
(Dis (Exi (Pre True (Suc 0) [0]))
(Exi (Pre True 0 [0])))]]

PROVER
[[(0, Uni (Con (Pre False 0 [0]) (Pre False (Suc 0) [0]))),
(0, Dis (Exi (Pre True (Suc 0) [0])) (Exi (Pre True 0 [0])))]]

PROVER
[[(0, Dis (Exi (Pre True (Suc 0) [0])) (Exi (Pre True 0 [0]))),
(0, Con (Pre False 0 [0]) (Pre False (Suc 0) [0]))]]

PROVER
[[(0, Con (Pre False 0 [0]) (Pre False (Suc 0) [0])),
(0, Exi (Pre True (Suc 0) [0])), (0, Exi (Pre True 0 [0]))]]

```
PROVER
  [[(0, Exi (Pre True (Suc 0) [0])), (0, Exi (Pre True 0 [0])),
     (0, Pre False 0 [0])],
    [(0, Exi (Pre True (Suc 0) [0])), (0, Exi (Pre True 0 [0])),
     (0, Pre False (Suc 0) [0])]]

PROVER
  [[(0, Exi (Pre True 0 [0])), (0, Pre False 0 [0]),
     (0, Pre True (Suc 0) [0]),
     (Suc 0, Exi (Pre True (Suc 0) [0]))],
    [(0, Exi (Pre True 0 [0])), (0, Pre False (Suc 0) [0]),
     (0, Pre True (Suc 0) [0]),
     (Suc 0, Exi (Pre True (Suc 0) [0]))]]

PROVER
  [[(0, Pre False 0 [0]), (0, Pre True (Suc 0) [0]),
     (Suc 0, Exi (Pre True (Suc 0) [0])), (0, Pre True 0 [0]),
     (Suc 0, Exi (Pre True 0 [0]))],
     [(0, Pre False (Suc 0) [0]), (0, Pre True (Suc 0) [0]),
     (Suc 0, Exi (Pre True (Suc 0) [0])), (0, Pre True 0 [0]),
     (Suc 0, Exi (Pre True 0 [0]))]]

True
```

9 Related Work and Conclusion

There are several formalizations of proof systems in Isabelle/HOL and other systems [1–5, 7, 8]. The present simple prover based on Tom Ridge's entry in the Archive of Formal Proofs [6] seems indeed unique given its overall simplicity, full automation and code generation. Future work include further polishing the proofs and development of a stand-alone tool for illustrating the unfolding rules.

Acknowledgement. Many thanks to Tom Ridge, Alexander Birch Jensen and the Isabelle team at TUM for help and discussions.

Appendix: Simple Prover Unfolding Rules

The "program" rules

check p ≡ PROVER [[(0,p)]]

PROVER (h # t) ≡ PROVER (maps solve (h # t))
PROVER [] ≡ True

solve [] ≡ [[]]
solve (h # t) ≡ track t (fst h) (snd h)

track s n (Pre b i v) ≡ stop [s @ [(0,Pre b i v)]] (Pre (¬ b) i v) (map snd s)
track s n (Con p q) ≡ [s @ [(0,p)],s @ [(0,q)]]
track s n (Dis p q) ≡ [s @ [(0,p),(0,q)]]
track s n (Uni p) ≡ [s @ [(0,inst p (fresh (maps fv (Uni p # map snd s))))]]
track s n (Exi p) ≡ [s @ [(0,inst p n),(Suc n,Exi p)]]

maps f l ≡ concat (map f l)

stop c p [] ≡ c
stop c p (h # t) ≡ (if p = h then [] else stop c p t)

fresh [] ≡ 0
fresh (h # t) ≡ Suc (add (sub (dec (fresh t)) h) h)

add x 0 ≡ x
add x (Suc n) ≡ Suc (add x n)

sub x 0 ≡ x
sub x (Suc n) ≡ dec (sub x n)

dec 0 ≡ 0
dec (Suc n) ≡ n

inst p x ≡ sv (bind x) p

bind x 0 ≡ x
bind x (Suc n) ≡ n

sv f (Pre b i v) ≡ Pre b i (map f v)
sv f (Con p q) ≡ Con (sv f p) (sv f q)
sv f (Dis p q) ≡ Dis (sv f p) (sv f q)
sv f (Uni p) ≡ Uni (sv (increase f) p)
sv f (Exi p) ≡ Exi (sv (increase f) p)

increase f 0 ≡ 0
increase f (Suc n) ≡ Suc (f n)

fv (Pre b i v) ≡ v
fv (Con p q) ≡ fv p @ fv q
fv (Dis p q) ≡ fv p @ fv q
fv (Uni p) ≡ adjust (fv p)
fv (Exi p) ≡ adjust (fv p)

adjust [] ≡ []
adjust (h # t) ≡ extend (adjust t) h

extend l 0 ≡ l
extend l (Suc n) ≡ n # l

The "data" rules

Pre b i v = Con p q ≡ False
Con p q = Pre b i v ≡ False
Pre b i v = Dis p q ≡ False
Dis p q = Pre b i v ≡ False
Pre b i v = Uni p ≡ False
Uni p = Pre b i v ≡ False
Pre b i v = Exi p ≡ False
Exi p = Pre b i v ≡ False
Con p q = Dis p' q' ≡ False
Dis p' q' = Con p q ≡ False
Con p q = Uni p' ≡ False
Uni p' = Con p q ≡ False
Con p q = Exi p' ≡ False
Exi p' = Con p q ≡ False
Dis p q = Uni p' ≡ False
Uni p' = Dis p q ≡ False
Dis p q = Exi p' ≡ False
Exi p' = Dis p q ≡ False
Uni p = Exi p' ≡ False
Exi p' = Uni p ≡ False

Pre b i v = Pre b' i' v' ≡ b = b' ∧ i = i' ∧ v = v'
Con p q = Con p' q' ≡ p = p' ∧ q = q'
Dis p q = Dis p' q' ≡ p = p' ∧ q = q'
Uni p = Uni p' ≡ p = p'
Exi p = Exi p' ≡ p = p'

The "library" rules

$map\ f\ [] \equiv []$
$map\ f\ (h \# t) \equiv f\ h \# map\ f\ t$

$concat\ [] \equiv []$
$concat\ (h \# t) \equiv h\ @\ concat\ t$

$[] @ l \equiv l$
$(h \# t) @ l \equiv h \# t @ l$

$if\ True\ then\ x\ else\ y \equiv x$
$if\ False\ then\ x\ else\ y \equiv y$

$\neg\ True \equiv False$
$\neg\ False \equiv True$
$\neg \neg\ b \equiv b$

$fst\ (x,y) \equiv x$
$snd\ (x,y) \equiv y$

$0 = Suc\ n \equiv False$
$Suc\ n = 0 \equiv False$

$[] = h \# t \equiv False$
$h \# t = [] \equiv False$

$True = False \equiv False$
$False = True \equiv False$

$0 = 0 \equiv True$

$[] = [] \equiv True$

$True = True \equiv True$
$False = False \equiv True$

$(x,y) = (x',y') \equiv x = x' \wedge y = y'$

$Suc\ n = Suc\ n' \equiv n = n'$

$h \# t = h' \# t' \equiv h = h' \wedge t = t'$

$True \wedge b \equiv b$
$False \wedge b \equiv False$

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. 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)
6. Ridge, T.: A mechanically verified, efficient, sound and complete theorem prover for first order logic. Archive of Formal Proofs (Sep 2004), http://isa-afp.org/entries/Verified-Prover.shtml, Formal proof development
7. 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)
8. Villadsen, J., Jensen, A.B., Schlichtkrull, A.: NaDeA: A natural deduction assistant with a formalization in Isabelle. https://nadea.compute.dtu.dk
9. Villadsen, J., Schlichtkrull, A., From, A.H.: SimPro - Simple Prover - With a Formalization in Isabelle. https://github.com/logic-tools/simpro