Students' Proof Assistant (SPA)

Anders Schlichtkrull
Jørgen Villadsen
Andreas Halkjær From

Technical University of Denmark – DTU Compute

The Students' Proof Assistant (SPA) aims to both teach how to use a proof assistant like Isabelle but also to teach how reliable proof assistants are built. Technically it is a miniature proof assistant inside the Isabelle proof assistant. In addition we conjecture that a good way to teach structured proving is with a concrete prover where the connection between semantics, proof system, and prover is clear. In fact, the proofs in Lamport's TLAPS proof assistant have a very similar structure to those in the declarative prover SPA. To illustrate this we compare a proof of Pelletier's problem 43 in TLAPS, Isabelle/Isar and SPA.

A Verified Miniature Proof Assistant

Students' Proof Assistant (SPA) is an advanced e-learning tool for teaching proof assistants for students in computer science as well as in mathematics and complements our other e-learning tool, NaDeA (A Natural Deduction Assistant with a Formalization in Isabelle), which is available online and has been used by hundreds of BSc and MSc students at DTU in regular courses

We conjecture that a good way to teach structured proving is with a concrete prover where the connection between semantics, proof system, and prover is clear

Even for paper proofs Leslie Lamport recommends writing in a structured style

Starting Point

Programming and Verifying a Declarative First-Order Prover
Alexander Birch Jensen, John Bruntse Larsen, Anders Schlichtkrull & Jørgen Villadsen
Special Issue on Automated Reasoning – AI Communications 281-299 2018

A first-order prover with equality based on John Harrison's Handbook of Practical Logic and Automated Reasoning, Cambridge University Press, 2009

ML code reflection is used such that the entire prover can be executed within Isabelle as a very simple interactive proof assistant

As examples we consider Pelletier's problems 1-46

Isabelle lines of code: 4147

Pelletier's Problem 55

Someone in Dreadsbury Mansion killed Aunt Agatha. Agatha, the butler, and Charles live in Dreadsbury Mansion, and are the only ones to live there. A killer always hates, and is no richer than his victim. Charles hates noone that Agatha hates. Agatha hates everybody except the butler. The butler hates everyone not richer than Aunt Agatha. The butler hates everyone whom Agatha hates. Noone hates everyone. Who killed Agatha?

```
ML_val {* (* Pelletier p55 *)

auto ("lives(agatha) /\\ lives(butler) /\\ lives(charles) /\\ " ^
      "(killed(agatha,agatha) \\/ killed(butler,agatha) \\/ " ^
       "killed(charles,agatha)) /\\ " ^
      "(forall x y. killed(x,y) ==> hates(x,y) /\\ ~richer(x,y)) /\\ " ^
      "(forall x. hates(agatha,x) ==> ~hates(charles,x)) /\\ " ^
      "(hates(agatha,agatha) /\\ hates(agatha,charles)) /\\ " ^
      "(forall x. lives(x) /\\ ~richer(x,agatha) ==> hates(butler,x)) /\\ " ^
      "(forall x. hates(agatha,x) ==> hates(butler,x)) /\\ " ^
      "(forall x. ~hates(x,agatha) \\/ ~hates(x,butler) \\/ ~hates(x,charles)) " ^
      "==> killed(agatha,agatha) /\\ " ^
            "~killed(butler,agatha) /\\ " ^
            "~killed(charles,agatha)")

*}
```

Pelletier's Problem 46

$$(\forall x.\ P(x) \wedge (\forall y.\ P(y) \wedge H(y,x) \longrightarrow G(y)) \longrightarrow G(x)) \wedge$$

$$((\exists x.\ P(x) \wedge \neg G(x)) \longrightarrow$$

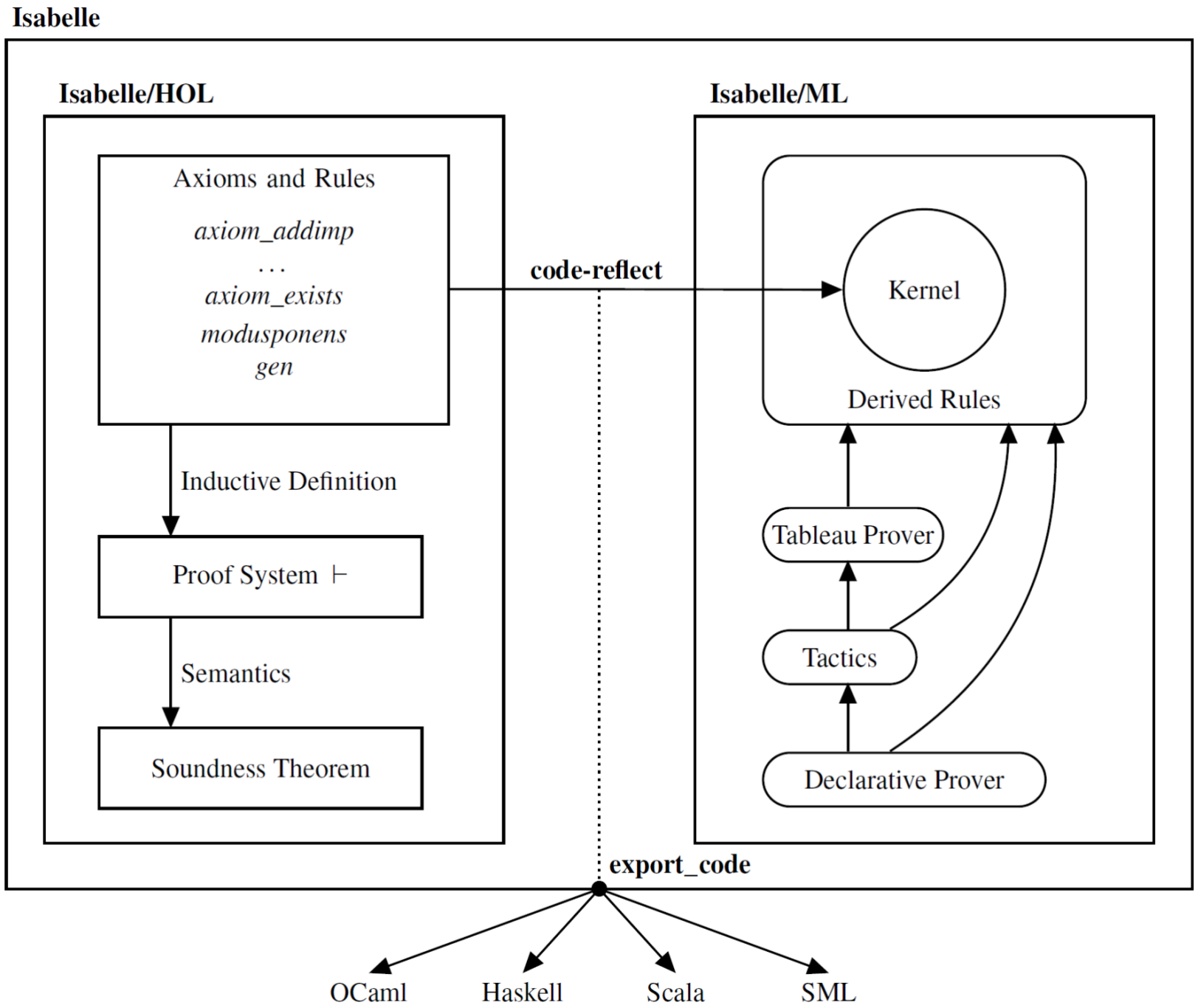
$$(\exists x.\ P(x) \wedge \neg G(x) \wedge (\forall y.\ P(y) \wedge \neg G(y) \longrightarrow J(x,y)))) \wedge$$

$$(\forall xy.\ P(x) \wedge P(y) \wedge H(x,y) \longrightarrow \neg J(y,x)) \longrightarrow$$

$$(\forall x.\ P(x) \longrightarrow G(x))$$

Seventy-Five Problems for Testing Automatic Theorem Provers
Francis Jeffry Pelletier
Journal of Automated Reasoning 191-216 1986

```
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
  ]
```


Isabelle
Isabelle/HOL
Isabelle/ML
Axioms and Rules
axiom_addimp
...
axiom_exists
modusponens
gen
code-reflect
Kernel
Derived Rules
Inductive Definition
Proof System ⊢
Tableau Prover
Semantics
Tactics
Soundness Theorem
Declarative Prover
export_code
OCaml
Haskell
Scala
SML

$$\frac{\vdash p \Rightarrow q \quad \vdash p}{\vdash q} \qquad \frac{\vdash p}{\vdash \forall x.\, p}$$

Handbook of Practical Logic and Automated Reasoning
John Harrison
Cambridge University Press 2009

$$\vdash \quad p \Rightarrow (q \Rightarrow p),$$

$$\vdash \quad (p \Rightarrow q \Rightarrow r) \Rightarrow (p \Rightarrow q) \Rightarrow (p \Rightarrow r),$$

$$\vdash \quad ((p \Rightarrow \bot) \Rightarrow \bot) \Rightarrow p,$$

$$\vdash \quad (\forall x.\, p \Rightarrow q) \Rightarrow (\forall x.\, p) \Rightarrow (\forall x.\, q),$$

$$\vdash \quad p \Rightarrow \forall x.\, p \quad [\textbf{provided } x \notin \mathrm{FV}(p)],$$

$$\vdash \quad (\exists x.\, x = t) \quad [\textbf{provided } x \notin \mathrm{FVT}(t)],$$

$$\vdash \quad t = t,$$

$$\vdash \quad s_1 = t_1 \Rightarrow \cdots \Rightarrow s_n = t_n \Rightarrow f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n),$$

$$\vdash \quad s_1 = t_1 \Rightarrow \cdots \Rightarrow s_n = t_n \Rightarrow P(s_1, \ldots, s_n) \Rightarrow P(t_1, \ldots, t_n).$$

$$\vdash \quad (p \Leftrightarrow q) \Rightarrow p \Rightarrow q,$$

$$\vdash \quad (p \Leftrightarrow q) \Rightarrow q \Rightarrow p,$$

$$\vdash \quad (p \Rightarrow q) \Rightarrow (q \Rightarrow p) \Rightarrow (p \Leftrightarrow q),$$

$$\vdash \quad \top \Leftrightarrow (\bot \Rightarrow \bot),$$

$$\vdash \quad \neg p \Leftrightarrow (p \Rightarrow \bot),$$

$$\vdash \quad p \wedge q \Leftrightarrow (p \Rightarrow q \Rightarrow \bot) \Rightarrow \bot,$$

$$\vdash \quad p \vee q \Leftrightarrow \neg(\neg p \wedge \neg q),$$

$$\vdash \quad (\exists x.\, p) \Leftrightarrow \neg(\forall x.\, \neg p).$$

Pelletier's Problem 43

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

Proof:
We are trying to prove an implication, so we start off by assuming the antecedent, calling it "A" so we can refer to it later in the proof. Since the statement is universally quantified, we arbitrarily fix an x and a y to use in the proof. We then show the biimplication by showing the conjunction of both directions and using the command *at once* which does pure first-order reasoning and can easily handle small steps such as this one.

Consider first the direction $Q(x,y) \longrightarrow Q(y,x)$. This is an implication so we start again by assuming the antecedent. We do not have to name it this time, as we only use it in the proof of the next statement where it can be referenced using *so*. This assumption matches the left-hand side of "A" which we appeal to using *by* and are thus allowed to conclude the right-hand side: That x and y are equivalent with regards to P. The next line swaps the order of the biimplication *at once*, which allows us to then appeal to "A" again, this time in the opposite direction, and conclude the goal $Q(x,y)$. Note that in the final step, the quantified x in "A" is instantiated with our fixed y and y with x.

The proof of the direction $Q(y,x) \longrightarrow Q(x,y)$ is exactly symmetric to the one above.

□

```
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 iprover
          then have "∀z. P(z,y) ⟷ P(z,x)" by iprover
          then show "Q(y,x)" using A by iprover
        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 iprover
          then have "∀z. P(z,x) ⟷ P(z,y)" by iprover
          then show "Q(x,y)" using A by iprover
        qed
      qed
      then show "Q(x,y) ⟷ Q(y,x)" by iprover
    qed
  qed
qed
```

```
ML_val {* (* Pelletier p43 (Harrison has it in a comment but the proof seems not to finish) *)

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
  ]

*}
```

```
(<!"(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
    …
    qed
   ],
   so our thesis at once,
   qed
  ],
  qed
 ],
 qed
]
```

Isabelle Formalization

```
type_synonym id = String.literal

datatype tm = Var id | Fun id ‹tm list›

datatype fm = Truth | Falsity | Pre id ‹tm list› | Imp fm fm | Iff fm fm | Con fm fm | Dis fm fm |
  Neg fm | Exi id fm | Uni id fm
```

Rule = function that takes theorems and returns a theorem

E.g. generalization:

```
definition gen' :: ‹id ⇒ fm ⇒ fm›
  where
    ‹gen' x p ≡ Uni x p›
```

E.g. modus ponens

```
definition ‹fail ≡ Truth›


definition modusponens' :: ‹fm ⇒ fm ⇒ fm›
  where
    ‹modusponens' r p' ≡ case r of Imp p q ⇒ if p = p' then q else fail | _ ⇒ fail›
```

These are actually the only two rules.

Axiom = function that given some arbitrary input returns a theorem

```
definition axiom_addimp' :: ‹fm ⇒ fm ⇒ fm›
 where
  ‹axiom_addimp' p q ≡ Imp p (Imp q p)›


definition axiom_impall' :: ‹id ⇒ fm ⇒ fm›
 where
  ‹axiom_impall' x p ≡ if ¬ free_in x p then (Imp p (Uni x p)) else fail›
```

We encode the rest of the rest of the axioms too of course.

We define the provable formulas

```
inductive OK :: ‹fm ⇒ bool› ("⊢ _" 0)
 where
  ‹⊢ s ⟹ ⊢ s' ⟹ ⊢ modusponens' s s'› |
  ‹⊢ s ⟹ ⊢ gen' _ s› |
  ‹⊢ axiom_addimp' _ _› |
  …
  ‹⊢ axiom_allimp' _ _ _› |
  …
  ‹⊢ axiom_exists' _ _›
```

We define the type of theorems

```
typedef "thm" = ‹{p :: fm. ⊢ p}›
proof -
  have ‹⊢ axiom_addimp' Truth Truth›
    using OK.intros by auto
  then show ?thesis by blast
qed
```

concl :: ‹thm ⇒ fm› = the function that extracts the formula from a theorem

We define semantics of FOL:

```
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 (Fun 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) ⇒ fm ⇒ bool›
 where
  ‹semantics _ _ _ Truth = True› |
  ‹semantics _ _ _ Falsity = False› |
  ‹semantics e f g (Pre i l) = (if i = STR ''='' ∧ length l = 2
   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 (Con p q) = (semantics e f g p ∧ semantics e f g q)› |
  ‹semantics e f g (Dis p q) = (semantics e f g p ∨ semantics e f g q)› |
  ‹semantics e f g (Neg p) = (¬ semantics e f g p)› |
  ‹semantics e f g (Exi x p) = (∃v. semantics (e(x := v)) f g p)› |
  ‹semantics e f g (Uni x p) = (∀v. semantics (e(x := v)) f g p)›
```

We want the axioms and rule to return *thm*s

```
setup_lifting type_definition_thm
```

Remember

```
definition gen' :: ‹id ⇒ fm ⇒ fm›
  where
    ‹gen' x p ≡ Uni x p›
```

We lift it:

```
lift_definition gen :: ‹id ⇒ thm ⇒ thm› is gen'
  using OK.intros(2) .
```

Remember

```
definition axiom_addimp' :: ‹fm ⇒ fm ⇒ fm›
  where
    ‹axiom_addimp' p q ≡ Imp p (Imp q p)›
```

We lift it:

```
lift_definition axiom_addimp :: ‹fm ⇒ fm ⇒ thm› is axiom_addimp'
  using OK.intros(3) .
```

Likewise we lift the rest of the rules and axioms

We prove that the theorems are valid:

theorem soundness: ‹semantics e f g (concl p)›

Code generation

```
code_reflect
  Proven
  datatypes
    tm = Var | Fun
    and
    fm = Truth | Falsity | Pre | Imp | Iff | Con | Dis | Neg | Exi | Uni
  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
```

This generates a module

```
structure Proven:
  sig
    type thm
    val axiom_addimp: fm -> fm -> thm
    …
    val concl: thm -> fm
    datatype tm = Fun of string * tm list | Var of string
    datatype fm = Con of fm * fm | Dis of fm * fm | Exi of string * fm |
                  Falsity | Iff of fm * fm | Imp of fm * fm |
                  Neg of fm | Pre of string * tm list | Truth |
                  Uni of string * fm
  end
```

With an implementation based on the specified functions

Load functions building on Proven

ML_file ‹SPA.ML›

A few examples

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)))" ›

Work in Progress – Now at Version 0.9.9

Logic Tools – NaDeA and other open source software for teaching logic

https://github.com/logic-tools

https://nadea.compute.dtu.dk

Conclusion

We have made an Isabelle/HOL formalization of sound axiomatic system.

We have used code generation to obtain a proof assistant from it.

On top of that is defined:
- **verified derived rules,**
- **verified tableau prover,**
- **verified tactics,**
- **verified declarative ITP.**

We conjecture that a good way to teach structured proving is with this concrete prover: The connection between semantics, proof system and prover is clear.