# A Mechanically Verified, Sound and Complete Theorem Prover for First Order Logic

Tom Ridge[1] and James Margetson

[1]LFCS, Informatics, Edinburgh University, Scotland, UK

**Abstract.** We present a system of first order logic, together with soundness and completeness proofs wrt. standard first order semantics. Proofs are mechanised in Isabelle/HOL. Our definitions are computable, allowing us to derive an algorithm to test for first order validity. This algorithm may be executed in Isabelle/HOL using the rewrite engine. Alternatively the algorithm has been ported to OCaML.

## 1 Introduction

In this work we mechanise a system of first order logic, and show soundness and completeness for this system. We also derive an algorithm which tests a sequent $s$ for first order validity: $s$ is true in all models iff our algorithm terminates with the answer *True*. All results are mechanised in Isabelle/HOL, and the theorem prover can be executed inside Isabelle/HOL using the rewrite engine. Alternatively, the definitions have been ported to OCaML to give a directly executable theorem prover.

This work is interesting for a number of reasons. Soundness is a prerequisite for a logical system. Completeness of a logical system means that any sequent true in all models is provable in the system. This signals a step change in confidence in the system: when attempting a proof of a true statement, we have gone from knowing that we will never err, to knowing that we will eventually succeed. Soundness and completeness for first order logic are the first significant results in metamathematics, so that the mathematical content of this work is interesting.

Our main contribution is to take this process one step further, and provide a mechanically verified algorithm that will actually prove every valid sequent. This is the first mechanically verified, sound and complete theorem prover for FOL. Others have presented mechanisations of completeness proofs for propositional logic, and occasionally predicate logic, but none have aimed to make the definitions executable. Completeness of a theorem prover is useful from a user's point of view: one wants to know that the failure of a theorem prover to prove a sequent arises from the unprovability of the sequent, and not from a deficiency in the theorem prover.

Our work is also interesting because of the range of possible applications.

> The mechanization of metamathematics itself has important implications for automated reasoning since metatheorems can be applied as labor-saving devices to simplify proof construction.[Sha94]

J. Hurd and T.F. Melham (Eds.): TPHOLs 2005, LNCS 3603, pp. 294–309, 2005.

For instance, reflection [Har95] is a mechanism whereby, having verified a piece of code correct, one can incorporate it in a trusted way in the kernel of a theorem prover or proof checker. Since we have verified a theorem prover for first order logic, we could conceivably incorporate this code into the kernel of a theorem prover or proof checker. The advantage of reflection is that we can safely extend our systems in non-trivial ways.

Proofs involving the semantics of logical systems are subtle, and can be hard to construct correctly, and to understand correctly, because one tends to bring a significant amount of intuition to the process, which may not be justified. As an example of these problems, free variables have long been felt to be problematic [ZTA$^+$], so much so that some formalisations of first order logic go to great lengths to avoid them all together [Qui62]. We feel that this work has pedagogic advantages in this area, and have attempted to illustrate this with a completely formal proof of the soundness of the $\forall R/\forall I$ rule[1].

We feel the mechanisation is also a contribution.

- We polish the proofs substantially: we were not afraid to change the definition of the logical system to make the proofs much more pleasant.
- The mechanisation is small, consisting of around 1000 lines of definitions and proofs, which makes comprehending and extending the work hopefully as simple as possible.
- We highlight dependencies between sections of the proof: soundness, for instance, does not require the universe of the models to be infinite.
- For metamathematical reasons, we aim to make the proofs as weak as possible. We remove uses of wellfounded induction in favour of natural number induction, which is the strongest principle we use, save for one application of König's lemma. Consequently the proofs could be carried out in relatively weak systems, certainly much weaker than HOL.

This work is also interesting from an aesthetic standpoint, in that it nicely combines the areas of mathematics, metamathematics, logic, and algorithms. In the following sections, we give *all* the definitions, and outline the main lemmas, of the mechanised proofs.

## 2 Proof Outline

The rules of our logical system are given in Fig. 1. Terms are simply variables $x_i$. Theoretically this is no restriction, since the usual first order terms may be simulated. We occasionally use $x, y, z$ to stand for variables. Parameter $a$ is a variable $x_i$. $a$ is used in preference to emphasise the eigenvariable status: $a$ does not appear free in $\forall x.A, \Gamma$. Atomic predicates are positive atoms $P, Q, \ldots$ and negative atoms $\overline{P}, \overline{Q}, \ldots$. Literals are atomic predicates applied to a tuple of terms, $P(x_{i_1}, \ldots, x_{i_n}), \overline{P}(x_{i_1}, \ldots, x_{i_n}), \ldots$. The intent is that $P(x_{i_1}, \ldots, x_{i_n})$ is true in a model iff $\overline{P}(x_{i_1}, \ldots, x_{i_n})$ is false. Note that there is no relation between

[1] Which is simply the $\forall$ rule here, since we work in a one sided system.

$$\frac{}{\vdash P(x_{i_1},\ldots,x_{i_k}),\Gamma,\overline{P}(x_{i_1},\ldots,x_{i_k}),\Delta}\ Ax \qquad \frac{\vdash \Gamma,P(x_{i_1},\ldots,x_{i_k})}{\vdash P(x_{i_1},\ldots,x_{i_k}),\Gamma}\ NoAx$$

$$\frac{}{\vdash \overline{P}(x_{i_1},\ldots,x_{i_k}),\Gamma,P(x_{i_1},\ldots,x_{i_k}),\Delta}\ \overline{Ax} \qquad \frac{\vdash \Gamma,\overline{P}(x_{i_1},\ldots,x_{i_k})}{\vdash \overline{P}(x_{i_1},\ldots,x_{i_k}),\Gamma}\ \overline{NoAx}$$

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

$$\frac{\vdash \Gamma,[a/x]A}{\vdash \forall x.A,\Gamma}\ \forall \qquad \frac{\vdash \Gamma,[x_n/x]A,(\exists x.A)^{n+1}}{\vdash (\exists x.A)^n,\Gamma}\ \exists$$

$NoAx$: $\overline{P}(x_{i_1},\ldots,x_{i_k})$ does not appear in $\Gamma$
$\overline{NoAx}$: $P(x_{i_1},\ldots,x_{i_k})$ does not appear in $\Gamma$
$\forall$: $a$ does not appear free in $\forall x.A,\Gamma$

**Fig. 1.** Deterministic Variant of Wainer and Wallen's System [WW92]

an atomic predicate applied to two tuples of different arity. For instance, the value of $P(x,y)$ in a model is independent of the value of $P(x,y,z)$. Formulas $A,B,\ldots$ are inductively defined as the least set containing literals, and closed under applications of $\wedge,\vee,\forall,\exists$. The omission of $\neg,\rightarrow$ is no restriction, since they can be defined as abbreviations as usual. Numbered formulae are pairs of a formula and a number. We say that a formula $A$ is tagged with $n$, and write $A^n$, when talking about the numbered formula $(A,n)$. Sequents $\vdash \Gamma$ are lists $\Gamma$ of numbered formulae. Initially every formula in a sequent is tagged with 0. Any formulae which arise as a result of applying rules $\wedge,\vee,\forall,\exists$ get tagged with 0. Except when the formula is quantified by an $\exists$, the tag is irrelevant and is not displayed.

A derivation in this system is simply a *finite* tree constructed using the rules. If we read these rules backwards, they give an algorithm for taking apart a sequent. The algorithm is deterministic[2] because exactly one rule applies to any given sequent.

We can connect syntax and semantics in the standard way by giving an interpretation of primitive propositions as propositions in a model and extending this to formulas and sequents such that the extension respects standard Tarski semantics. For example, $A\wedge B$ is true in a model iff $A$ is true, and $B$ is true.

Looking at the rules, it is quite easy to convince ourselves that they are *sound* in the sense that, if the premises of the rule are true in all models, with respect to all interpretations of free variables, then so too is the conclusion. For the axioms, this is just the recognition that we are working in a classical metalogic. Actually, soundness for rule $\forall$ is not so obvious, and we give a full proof later.

[2] In rule $\forall$, we choose $a$ to be $x_{i+1}$ where $i$ is the maximum index of the free variables occurring in the conclusion of the rule.

More interesting is the question of whether the rules are *complete*, that is, do they suffice to prove every true proposition? We wish to show that, if a sequent $s$ is true in all models, then we can prove it using our system of rules. Put another way, if we fail to prove $s$ then it had better be false in at least one model. Given that we fail to prove $s$, how can we exhibit a model where $s$ is false? When we attempt to prove $s$ using our system of rules, if we eventually close every branch then we have a proof of $s$. So if we fail to prove $s$, there must be at least one branch which goes on forever. We call this branch a failing path, and denote it $f$. We can then define a model by taking $\mathbb{N}$ as the domain, interpreting each variable $x_i$ as the number $i$, and interpreting an atomic predicate $P(x_{i_1}, \ldots, x_{i_n})$ as true iff $P(x_{i_1}, \ldots, x_{i_n})$ does not appear anywhere on the failing path $f$. By induction on the size of the formula, we can see that every formula appearing on $f$ must get false. Since $f$ starts at $s$, and all formulae in $s$ get false, then $s$ must also get false in this model. Let us consider the $\wedge$ case in the inductive argument. We must show that if $P \wedge Q$ appears on $f$, then $P \wedge Q$ gets false in the model. Assume $P \wedge Q$ appears on $f$. We must show that $P \wedge Q$ gets false in the model. Since $P \wedge Q$ appears in a sequent on $f$, eventually $P \wedge Q$ gets to the head of the sequent, and rule $\wedge$ is applied. At this point, either $P$ appears on $f$ or $Q$ appears on $f$. By induction hypothesis, (at least one of) $P$ or $Q$ gets false in the model, so $P \wedge Q$ gets false in the model.

Having shown soundness and completeness for our system, we can derive an algorithm for first order validity: the algorithm simply takes the initial sequent $s$, and applies the rules of the logical system, keeping track of those sequents that appear at each stage. If at any stage all branches have been closed and there are no more sequents to consider, the algorithm will terminate with the answer *True*: if the algorithm terminates with *True*, then certainly we have a finite derivation and $s$ is valid. Conversely, if $s$ is valid, then by completeness there is a finite derivation, and we can argue that at stage $n$ our algorithm has considered all potential derivations of depth less than or equal to $n$. In this way, we can see that the algorithm will terminate with *True* iff $s$ is valid. We also note that by the undecidability result for first order logic, if the sequent would lead to an infinite derivation, the algorithm cannot always terminate with *False*.

## 3 Formalisation

### 3.1 Notation

We work in Isabelle/HOL, a variant of classical, simply typed, higher order logic. Isabelle/HOL has a meta-logic. The symbol $\Longrightarrow$ stands for the implication of the meta-logic. Nothing substantial is lost by considering this synonymous with the object level implication $\longrightarrow$. Nested meta-logical implication $A \Longrightarrow B \Longrightarrow C$ can be written $[\![ A; B ]\!] \Longrightarrow C$. The symbol $\equiv$ stands for the equality of the meta-logic. Again, nothing substantial is lost by considering this synonymous with the object level equality $=$. Type aliases are syntactic shorthand for the underlying type, and have no logical meaning. On the other hand, new types

may be introduced axiomatically, or defined in relation to an already existing type. The type $\mathbb{N}$ of natural numbers is written *nat*. Type constructors are functions mapping type lists to types. Application of a type constructor is typically written postfix. For example, the type of sets over an underlying type $'a$ is $'a\ set$. Application of a function $f$ to an argument $a$ is written simply $f\ a$. The function which takes an argument $x$ and produces a result $f\ x$ is denoted by $\lambda x.\ f\ x$. The function which is exactly the same as $f$, except that $x$ is mapped to $y$, is written $f(x := y)$. The type of a function with domain $'a$ and codomain $'b$ is $'a \Rightarrow 'b$. ML style datatypes are present, as is definition by primitive and wellfounded recursion. Lists are a particularly important datatype. The type of lists over a base type $'a$ is formed by applying the *list* type constructor, viz. $'a\ list$. The empty list is written $[]$, whilst the list $xs$ with an additional $x$ on the front is written $x \#\ xs$. The list containing $1, 2, 3$ is written $[1, 2, 3]$. The functions to take the head, *hd*, and tail, *tl*, of a list are as usual, such that $hd\ (x \#\ xs) = x$ and $tl\ (x \#\ xs) = xs$. The concatenation of two lists is written $xs\ @\ ys$. The function *set* takes an $'a\ list$ to an $'a\ set$ in the obvious way. The type of pairs over base types $'a$, $'b$ is written $'a \times 'b$, and has the standard projections *fst*, *snd*. Datatypes are accompanied by destructors in the form of case statements. For example, the case distinction on natural numbers may be written $case\ x\ of\ 0 \Rightarrow baseCase \mid Suc\ n \Rightarrow stepCase\ n$. This has the eta-contracted form $natcase\ baseCase\ stepCase$.

### 3.2 Formulas

Predicates $P_i$ are identified by their index $i \in \mathbb{N}$. Similarly variables $x_i$.

```
types pred = nat
```

```
types var = nat
```

Terms are variables. Formulas are literals (positive and negative atomic predicates applied to tuples of variables, here represented as lists), conjunctions, disjunctions, foralls and exists. Variables and binding are handled by De Bruijn's nameless representation: a bound variable is a natural number indicating the number of enclosing quantifiers one must traverse to find the binding quantifier. This represents a deep embedding of the logic [WN04]. Free variables, substitution, and instantiation are defined as usual.

```
datatype form =
  PAtom pred (var list)
  | NAtom pred (var list)
  | FConj form form
  | FDisj form form
  | FAll form
  | FEx form
```

```
consts fv :: form ⇒ var list
primrec
  fv (PAtom p vs) = vs
  fv (NAtom p vs) = vs
  fv (FConj f g) = (fv f) @ (fv g)
  fv (FDisj f g) = (fv f) @ (fv g)
  fv (FAll f) = preSuc (fv f)
  fv (FEx f) = preSuc (fv f)
```

```
consts preSuc :: nat list ⇒ nat list
primrec
  preSuc [] = []
  preSuc (a#list) = (case a of 0 ⇒ preSuc list | Suc n ⇒ n#(preSuc list))
```

```
consts subst :: (nat ⇒ nat) ⇒ form ⇒ form
primrec
```

```
  subst r (PAtom p vs) = (PAtom p (map r vs))
  subst r (NAtom p vs) = (NAtom p (map r vs))
  subst r (FConj f g) = FConj (subst r f) (subst r g)
  subst r (FDisj f g) = FDisj (subst r f) (subst r g)
  subst r (FAll f) = FAll (subst (λ y. case y of 0 ⇒ 0 | Suc n ⇒ Suc (r n)) f)
  subst r (FEx f) = FEx (subst (λ y. case y of 0 ⇒ 0 | Suc n ⇒ Suc (r n)) f)


constdefs finst :: form ⇒ var ⇒ form
  finst body w ≡ subst (λ v. case v of 0 ⇒ w | Suc n ⇒ n) body
```

Sequents are formula lists. A numbered formula is a pair of a natural number and a formula. Numbered sequents are numbered formula lists. We define mappings between sequents and numbered sequents.

```
types seq = form list

types nform = nat × form

types nseq = nform list

constdefs s-of-ns :: nseq ⇒ seq
  s-of-ns ns ≡ map snd ns

constdefs ns-of-s :: seq ⇒ nseq
  ns-of-s s ≡ map (λ x. (0,x)) s
```

The free variables of a sequent, the maximum of a list of free variables, and a new variable are defined.

```
constdefs sfv :: seq ⇒ var list
  sfv s ≡ flatten (map fv s)

consts maxvar :: var list ⇒ var
primrec
  maxvar [] = 0
  maxvar (a#list) = max a (maxvar list)

constdefs newvar :: var list ⇒ var
  newvar vs ≡ Suc (maxvar vs)
```

### 3.3 Derivations

In addition to the rules in Fig. 1, we add the following rule to deal with the degenerate case of the empty sequent. We note that we could simply terminate the proof at this point, but that this rule ensures that the proofs in the rest of the mechanisation are uniform.

$$\frac{\vdash}{\vdash} \mathit{Nil}$$

Then to represent the rules, we use a function mapping the conclusion of a rule to a list of premises.

```
consts subs :: nseq ⇒ nseq list
primrec
  subs [] = [[]]
  subs (x#xs) =
  (let (m,f) = x in case f of
  PAtom p vs ⇒ if NAtom p vs ∈ set (map snd xs) then [] else [xs@[(0,PAtom p vs)]]
  | NAtom p vs ⇒ if (PAtom p vs) ∈ set (map snd xs) then [] else [xs@[(0,NAtom p vs)]]
  | FConj f g ⇒ [xs@[(0,f)],xs@[(0,g)]]
  | FDisj f g ⇒ [xs@[(0,f),(0,g)]]
  | FAll f ⇒ [xs@[(0,finst f (newvar (sfv (s-of-ns (x#xs)))))]]
  | FEx f ⇒ [xs@[(0,finst f m),(Suc m,FEx f)]])
```

Derivations are defined inductively wrt. this function. The additional natural number indicates the depth of a node in the derivation tree, and aids inductive

arguments. We also make an abstracting definition of a predicate to recognise a terminal sequent, that is, one that can be closed by an application of the rules $Ax, \overline{Ax}$.

```
consts deriv :: nseq ⇒ (nat × nseq) set
inductive deriv s
  intros
  init: (0,s) ∈ deriv s
  step: (n,x) ∈ deriv s ⟹ y ∈ set (subs x) ⟹ (Suc n,y) ∈ deriv s

consts is-axiom :: seq ⇒ bool
primrec
  is-axiom [] = False
  is-axiom (a#list) = ((∃ p vs. a = PAtom p vs ∧ NAtom p vs ∈ set list)
    ∨ (∃ p vs. a = NAtom p vs ∧ PAtom p vs ∈ set list))
```

Our first task is to show that these derivations are sound wrt. first order models.

### 3.4 Models

A first order model $(M, I)$ is a set of elements $M$ and an interpretation $I$ of the syntactic predicates $P_j$ as predicates $\mathcal{P}_j$ over tuples (represented as lists) of elements of the model. Which type should the elements of a model be drawn from? We assert the existence of a universal type[3].

```
typedecl U                    types model = U set × (pred ⇒ U list ⇒ bool)
```

An alternative would be to quantify over all models at all types. However, type quantification is currently not supported in HOL. We would like to echo Harrison [Har98] who notes the utility of type quantification in HOL [T.F92] in a similar context.

The third semantic notion is that of an environment, which is an assignment of elements in the model to free variables.

```
types env = var ⇒ U

constdefs is-env :: model ⇒ env ⇒ bool
  is-env MI e ≡ ∀ x. e x ∈ (fst MI)
```

Given a model and an environment, we can evaluate the truth of a formula, using standard Tarski semantics.

```
consts feval :: model ⇒ env ⇒ form ⇒ bool
primrec
  feval MI e (PAtom P vs) = (let IP = (snd MI) P in IP (map e vs))
  feval MI e (NAtom P vs) = (let IP = (snd MI) P in ¬ (IP (map e vs)))
  feval MI e (FConj f g) = ((feval MI e f) ∧ (feval MI e g))
  feval MI e (FDisj f g) = ((feval MI e f) ∨ (feval MI e g))
  feval MI e (FAll f) = (∀ m ∈ (fst MI). feval MI (λ y. case y of 0 ⇒ m | Suc n ⇒ e n) f)
  feval MI e (FEx f) = (∃ m ∈ (fst MI). feval MI (λ y. case y of 0 ⇒ m | Suc n ⇒ e n) f)
```

This extends to sequents, and finally we can say what it means for a sequent to be valid.

[3] This is one of only two places where we make axiomatic assertions. Both could be avoided by using existing type *ind* instead of declaring $U$.

```
consts seval :: model ⇒ env ⇒ seq ⇒ bool
primrec
  seval MI e [] = False
  seval MI e (x#xs) = (feval MI e x ∨ seval MI e xs)

constdefs svalid :: form list ⇒ bool
  svalid s ≡ ∀ MI e. is-env MI e ⟶ seval MI e s
```

### 3.5 Soundness

We prove that the rules are sound, that is, that any sequent at the root of a derivation is true in all models. Conceptually this is done by induction on the finite derivation, from leaf to root.

```
lemma soundness: finite (deriv (ns-of-s s)) ⟹ svalid s
```

For each rule, we need a lemma stating that if the premises are true in all models, then so too is the conclusion. We treat the most interesting case of the $\forall$ rule, which we prove for abitrary fresh $u$. This case depends on the following lemma, which states roughly that evaluating $[u/x]f$ in environment $e$ is equivalent to evaluating $f$ in an environment $e(x := e\ u)$, i.e. in the same environment except that the free variable $x$ gets mapped to whatever $u$ was mapped to by the original environment $e$.

```
lemma feval-finst: feval MI e (finst A u) = feval MI (nat-case (e u) e) A
```

The statement of the main lemma is as follows.

```
lemma sound-FAll: u ∉ set (sfv (FAll f#s)) ⟹ svalid (s@[finst f u]) ⟹ svalid (FAll f#s)
```

Suppose we wish to show that $\forall x.f$ is true in all models wrt. all environments $e$, and we assume that for $u$ fresh wrt. $f$, $[u/x]f$ is true in all models wrt. all environments $e'$. To show $\forall x.f$ is true wrt. environment $e$, we must show that for all $m$, $f$ is true wrt. environment $e(x := m)$. From the assumption, choosing $e'$ to be $e(u := m)$, we get that $[u/x]f$ is true wrt. environment $e(u := m)$. By lemma *feval-finst*, this is equivalent to $f$ being true wrt. environment $e(u := m)(x := (e(u := m)\ u))$, and this environment is simply $e(u := m)(x := m)$. Since $u$ is fresh wrt. $f$, we actually have that $f$ is true wrt. environment $e(x := m)$, which is what we had to show. This is the essential idea behind the proof of the main lemma. An Isar proof of this lemma is included in the development, but omitted here for space reasons.

### 3.6 Failing Path

We wish to show completeness of our rule system wrt. validity, i.e. if some sequent $s$ is true in all models, then it is provable. Alternatively, if $s$ is not provable, we must exhibit a model where $s$ is false. If $s$ is not provable, then when we attempt to prove it using the rules of our system, we will not end up with a finite derivation. If the derivation tree is infinite then, since it is finitely branching, we can use König's lemma to find an infinite path in the tree. We call this infinite path a failing path. We define a failing path as a function from a

derivation tree to a path through the derivation tree. Paths are simply functions with domain *nat*.

```
consts failing-path :: (nat × nseq) set ⇒ nat ⇒ (nat × nseq)
primrec
  failing-path ns 0 = (SOME x. x ∈ ns ∧ fst x = 0 ∧ infinite (deriv (snd x))
    ∧ ¬ is-axiom (s-of-ns (snd x)))
  failing-path ns (Suc n) = (let fn = failing-path ns n in SOME fsucn. fsucn ∈ ns
    ∧ fst fsucn = Suc n ∧ (snd fsucn) ∈ set (subs (snd fn)) ∧ infinite (deriv (snd fsucn))
    ∧ ¬ is-axiom (s-of-ns (snd fsucn)))
```

If $f$ is the failing path for *deriv s* then the essential property of $f$ is given in the following.

```
lemma (in loc1) is-path-f: infinite (deriv ns) ⟹ ∀ n. f n ∈ deriv ns ∧ fst (f n) = n
  ∧ (snd (f (Suc n))) ∈ set (subs (snd (f n))) ∧ infinite (deriv (snd (f n)))
```

Note that HOL is sufficiently powerful that we can simply define a failing path through a tree using the choice operator, avoiding an explicit invocation of König's lemma.

### 3.7 Contains, Considers

We now wish to talk about when a path $f$ contains a numbered formula *nf* at position $n$. We also introduce the notion of when a formula is considered at a point $n$ in a path, which is when the formula is at the head of the sequent at position $n$.

```
constdefs contains :: (nat ⇒ (nat × nseq)) ⇒ nat ⇒ nform ⇒ bool
  contains f n nf ≡ nf ∈ set (snd (f n))

constdefs considers :: (nat ⇒ (nat × nseq)) ⇒ nat ⇒ nform ⇒ bool
  considers f n nf ≡ case snd (f n) of [] ⇒ False | (x#xs) ⇒ x = nf
```

### 3.8 Models 2

A falsifying model will in general consist of at least countably many elements. So far, we have said nothing about the size of our universe type. We require that it is infinite so that it can contain an infinite falsifying model. We assert the existence of an injective function from *nat* to $U$[4].

```
consts ntou :: nat ⇒ U

axioms ntou: inj ntou
```

```
constdefs uton :: U ⇒ nat
  uton ≡ inv ntou
```

### 3.9 Falsifying Model from Failing Path

We are now in a position to define a falsifying model, given an infinite derivation.

```
constdefs model :: nseq ⇒ model
  model ns ≡ (range ntou, λ p ms. (let f = failing-path (deriv ns) in
  (∀ n m. ¬ contains f n (m,PAtom p (map uton ms)))))
```

[4] If we had taken the existing HOL type *ind* instead of declaring $U$, then we could avoid asserting this axiom, and our development would be conservative.

The point of the model is that any formula contained in a sequent on the failing path $f$ gets false in the model. This is proved by induction on the size of the formula.

**lemma** ⟦ *f = failing-path (deriv (ns-of-s s)); infinite (deriv (ns-of-s s)); contains f n (m,A)* ⟧ ⟹ ¬ *feval (model (ns-of-s s)) ntou A*

Let us treat the case that $(\exists x.P\ x)^n$ appears on $f$. Then we know that $(\exists x.P\ x)^0$ appears on $f$. Then $(\exists x.P\ x)^0$ eventually gets considered, and $P\ x_0$ and $(\exists x.P\ x)^1$ appears on $f$. Then $(\exists x.P\ x)^1$ eventually gets considered, and $P\ x_1$ and $(\exists x.P\ x)^2$ appears on $f$. Continuing in this way, we see that for all $n$, $P\ x_n$ appears on $f$. Applying the induction hypothesis to each of these, we see that for all $n$, the interpretation of $P\ x_n$, i.e. $\mathcal{P}\ n$, is false in the model. So "there exists an $n$ such that $\mathcal{P}\ n$" is false in the model, and so $\exists x.P\ x$ gets false in the model.

### 3.10 Completeness

Since the sequent $s$ which gave rise to the infinite derivation appears on the failing path at position 0, and all formulas in $s$ get false, it too must get false in the model. We have thus found our falsifying model, and $s$ could never have been proved using any sound system of rules. The completeness lemma is as follows.

**lemma** *completeness*: *infinite (deriv (ns-of-s s))* ⟹ ¬ *svalid s*

### 3.11 Soundness and Completeness

We can combine our soundness and completeness results to get a lemma which connects validity and provability.

**lemma** *soundComplete*: *svalid s = finite (deriv (ns-of-s s))*

### 3.12 Algorithm and Computation

The rules of our system are deterministic. We therefore want to turn the rules into a deterministic algorithm. This algorithm checks to see if the derivation is finite by repeatedly applying the rules to all sequents at depth $n$, to obtain the list of sequents at depth $n+1$. If there are no sequents at a given depth, then all branches have been closed and we have found our finite derivation.

We define a global version of the algorithm as a function that takes an initial sequent $s$, and a number $n$, and gives back the list of sequents at depth $n$ in the derivation rooted at $s$. We need a step function that takes a list of sequents at depth $n$ in the derivation, and gives back a list of sequents at depth $n+1$.

**constdefs** *step* :: *nseq list* ⇒ *nseq list*
*step* ≡ λ *s*. *flatten* (*map subs s*)

We are going to iterate this step function repeatedly, so we need an iteration function.

**consts** *iter* :: $('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow nat \Rightarrow 'a$ — fold for nats
**primrec**
*iter g a 0* = *a*
*iter g a* (*Suc n*) = *g* (*iter g a n*)

So iterating the step function $n$ times on an initial sequent $s$ gives the sequents at depth $n$ in the derivation.

**lemma** $\forall\ x.\ ((n,x) \in deriv\ s) = (x \in set\ (iter\ step\ [s]\ n))$

Now we know that the derivation from $s$ is finite iff there is some $n$ such that there are no sequents at depth $n$ in the derivation, in which case the $n$th iteration of the step function on $s$ will be empty.

**lemma** *finite-deriv*: *finite* (*deriv s*) = ($\exists$ *m. iter step* [*s*] *m* = [])

Now the following is the definition in OCaML of a function that searches the natural numbers to see if there is an $n$ such that the $n$th iteration of the *subs* function on an initial sequent $s$ is empty. By lemma *finite-deriv*, if the derivation is finite, then there is some $n$ such that the $n$th iteration is empty, and the algorithm will terminate. However, if the derivation is not finite, then our algorithm will fail to terminate, searching ever increasing $n$.

```
let rec prove' s n = (if iter step s n = [] then true else prove' s (n+1));;
let prove s = prove' [ns_of_s s] 0;;
```

There is an obvious source of inefficiency in that, having computed the $n$th iteration, we throw the result away and compute the $n + 1$ th iteration from scratch. The following is an equivalent, but much more efficient implementation.

```
let rec prove' s = (if s = [] then true else prove' (step s));;
let prove s = prove' [ns_of_s s];;
```

When we come to replicate this in HOL we run into a problem. HOL is a logic of total functions, whilst the general recursive OCaML definitions just given are clearly partial. However, we can improve our confidence in the OCaML definitions as follows. Firstly, we define our function *prove′* in a non-constructive way.

**constdefs** *prove′* :: *nseq list* ⇒ *bool*
*prove′ s* ≡ ∃ *m. iter step s m* = []

**constdefs** *prove* :: *seq* ⇒ *bool*
*prove s* ≡ *prove′* [*ns-of-s s*]

From lemma *finite-deriv*, *prove s* corresponds to the finiteness of the derivation from $s$.

**lemma** *finite-deriv-prove*: *finite* (*deriv* (*ns-of-s s*)) = *prove s*

Note that together with lemma *soundComplete* we have that validity of $s$ is equivalent to *prove s*. We can even show that *prove'* satisfies the relation implied by the general recursive OCaML definition.

**lemma** *prove′*: *prove′ s* = (*if s* = [] *then True else prove′* (*step s*))

At this point, we believe that if the OCaML function terminates with *true*, then the derivation is finite. Conversely, we believe that if the derivation is finite, there is some $n$ such that the $n$th iteration is empty, and our algorithm will discover this at stage $n$ and terminate with *true*. On the other hand, we also believe that if the derivation is not finite, then the algorithm will fail to terminate. We have shown formally in HOL that any function that claimed to do this must satisfy the relation implied in lemma *prove'*. However, there are many functions that satisfy this recursive equation, such as the constant function of one argument that returns *true*, so that our beliefs have not been fully formalised. To go further would require some explicit notion of termination, and some way of handling general recursive definitions formally in HOL. We discuss this further in Sect. 5.

To execute the prover in Isabelle, we can rewrite the *prove* function to *prove'*, then use the lemma *prove'* to rewrite *prove'*. Further rewriting can occur. Examining the definitions, one sees that all functions involved are computable by rewriting. Of course, this rewriting will not terminate on some inputs, but if it does terminate with the output *True*, then the sequent is valid, and this will have been formally proved inside Isabelle. In terms of performance, one can comfortably run the verified prover inside Isabelle on small examples using the rewriting engine[5]. Alternatively, we have transported the definitions to OCaML, to give a directly executable program. For instance, the last few clauses of the OCaML program are as follows.

```
let subs t = match t with
    [] -> [[]]
  | (x::xs) -> let (m,f) = x in match f with
  PAtom (p,vs) -> if mem (NAtom (p,vs)) (map snd xs) then [] else [xs@[(0,PAtom (p,vs))]]
| NAtom (p,vs) -> if mem (PAtom (p,vs)) (map snd xs) then [] else [xs@[(0,NAtom (p,vs))]]
| FConj (f,g)  -> [xs@[(0,f)];xs@[(0,g)]]
| FDisj (f,g) -> [xs@[(0,f);(0,g)]]
| FAll f -> [xs@[(0,finst f (newvar (sfv (s_of_ns (x::xs)))))]]
| FEx f -> [xs@[(0,finst f m);(suc m,FEx f)]];;

let step = fun s -> flatten (map subs s);;

let rec prove' s = (if s = [] then true else prove' (step s));;

let prove s = prove' [ns_of_s s];;
```

## 4 Related Work

Completeness for first order logic has been mechanised several times, but without focusing on executability. In HOL Light, Harrison develops basic model theory for first order predicate logic in [Har98], mechanising a textbook by Kreisel and Krivine, but without addressing executability considerations. In Isabelle/HOL, Berghofer has also tackled classical first order predicate logic [Ber02], mechanising a textbook by Melvin and Fitting, again without focusing on executability.

In ALF, Persson has mechanised a constructive proof of intuitionistic predicate logic [Per96], but we were unable to trace the thesis. From comments in

[5] This applies to the newest version: the original was somewhat inefficient.

other work, it appears that Persson formalised the semantics of FOL formulae wrt. formal topology, so that the development is presumably substantially different from that here.

Completeness for propositional logic has also been tackled several times, although these results tend to be substantially easier to mechanise. There is a line of work by Underwood et al. in the NuPRL system. Primarily this involves mechanised proofs of completeness for intuitionistic propositional logic. The final paper in this line appears to be [Cal99], which usefully references much of the previous work. Included in this work is a paper [Und95] that discusses computational aspects of classical reasoning, applied to completeness results for intuitionistic logic, although these results were not mechanised. In Coq there has been much work on mechanising propositional logic. For instance, Weich tackles intuitionistic propositional logic in [Wei01].

The basic idea for this type of completeness proof is found in the work of Henkin [Hen49], where the author introduces the then radical idea of utilising the terms of the logical system as the elements of the model. Such a model is usually called a Henkin-model. The proof proceeds by successively extending the language and the term model until a maximally consistent and term complete model is formed. It is a relatively short step from here to defining a model directly from a failing derivation, as we have done here. However, we are unable to trace the paper in which this step appears for the first time. A succinct presentation of this approach is [WW92]. This work was originally mechanised in Isabelle/HOL by Margetson [Mar99]. This was then remechanised by Ridge, who modified the logical system, simplified and polished the proofs, and extended the work so that the prover is executable by rewriting inside Isabelle.

Less closely related is work on verifying proof checkers. For instance, Pollack has a verified type checker in [Pol95], and further work with McKinna is reported in [MP99].

More generally, this work is an exercise in mechanising results in metamathematics. A good example of a much more comprehensive mechanisation in this area is the work of Shankar on mechanising Gödel's incompleteness theorem [Sha94].

## 5 Conclusion and Future Work

We have presented a deterministic system of first order logic, proved soundness and completeness, and captured the system as an algorithm that can be directly executed inside Isabelle using the rewrite system. There are many extensions that might be considered.

Most immediately, we claim that if the derivation is finite, then the algorithm will terminate with *true*. To make this claim fully formal would require a treatment of general recursive definitions and explicit non-termination in HOL. One approach would be to work with a formalised semantics for OCaML, or some other suitable language. A more abstract approach would be to apply the techniques of LCF, a logic which explicitly deals with termination and recursive definitions.

Currently, there is a "leap of faith" [Har95] required to bridge the gap between the formalised definitions and those in OCaML. As we have shown, HOL is already a powerful language for expressing functional algorithms. A more radical approach than mechanising a theory of recursive functions inside HOL, is simply to carve out an exectuable subset of HOL itself. This is harder than it appears because one must treat general recursive definitions.

Terms in our system are simply variables. Although theoretically this is no restriction, it would be interesting to extend the mechanisation to deal with full first order terms. We might consider a representation of terms such as the following.

**datatype** *folterm = Var nat | App nat (folterm list)*

It is then not too difficult to enumerate these in an effective way. The next extension is to equality. Again, the lack of equality is theoretically no restriction, but it is usual to treat equality as a special relation, rather than axiomatising its properties.

We can also seek to extend the formalisation to cover other results in proof theory and automatic theorem proving, or as the basis of a more substantial verified theorem prover. Although we have not stated the result explicitly, our system does not use the *Cut* rule, so that the proofs generated are *Cut* free. *Cut* elimination is one of the main results of proof theory. Our proofs represent a semantic proof of *Cut* elimination. It would be interesting to tackle a syntactic proof, such as that by Pfenning [Pfe00]. Indeed, Pfenning cites *Cut* elimination as a challenge problem for formalisation. An approach along our current lines would have advantages over Pfenning's, in that the embedding is deeper, and properties such as coverage could be proven. Resolution is perhaps most easily treated as a variant of proof search in a system such as G3c [Avr93]. If syntactic *Cut* elimination were in place, a proof of completeness for a resolution based system would be relatively straight forward. Alternatively, one could try for a direct semantic proof. Mechanisation of these results could provide benefits to the community, allowing proposed improvements in algorithms to be formally assessed in terms of completeness preservation.

We alluded to the use of the reflection mechanism to incorporate verified code into the kernel of a theorem prover. It would be interesting to port the proofs to a system that supported such a feature, and investigate issues such as performance. Previous versions of this paper used the word "efficient" in the title, which referred to the fact that the algorithm was tail recursive, without backtracking. The algorithm does not use unification to select quantifier instantiations, and so is roughly comparable to Gilmore's procedure in terms of performance. In this sense, it is not competitive with current unification based approaches. It would be interesting to examine the performance of the system when extended to use unification.

The mechanisation described here can be found at the Archive for Formal Proofs [afp], which also includes the related OCaML code. Alternatively, the newest version is maintained at Ridge's homepage [Rid].

Finally, we would like to thank the anonymous reviewers for their extremely close reading which uncovered several inadequacies in a previous version.

## References

[afp] The archive of formal proofs. http://afp.sourceforge.net/.

[Avr93] Arnon Avron. Gentzen-type systems, resolution and tableaux. *Journal of Automated Reasoning*, 10(2):265–281, 1993.

[Ber02] Stefan Berghofer. Formalising first order logic in isabelle, 2002. http://www4.in.tum.de/~streckem/Admin/club2_berghofer_model_theory.pdf.

[Cal99] James Caldwell. Intuitionistic tableau extracted. In *Proceedings of International Conference on Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX'99)*, volume 1617 of *LNAI*, pages 82–96. Springer-Verlag, 1999. http://www.nuprl.org/documents/Caldwell/tableaux99.html.

[Har95] John Harrison. Metatheory and reflection in theorem proving: A survey and critique. Technical Report CRC-053, SRI Cambridge, Millers Yard, Cambridge, UK, 1995. Available on the Web as http://www.cl.cam.ac.uk/users/jrh/papers/reflect.dvi.gz.

[Har98] John Harrison. Formalizing basic first order model theory. In Jim Grundy and Malcolm Newey, editors, *Theorem Proving in Higher Order Logics: 11th International Conference, TPHOLs'98*, volume 1497 of *Lecture Notes in Computer Science*, pages 153–170, Canberra, Australia, 1998. Springer-Verlag.

[Hen49] Leon Henkin. The completeness of the first-order functional calculus. *The Journal of Symbolic Logic*, 14:159–166, 1949.

[Mar99] James Margetson. Completeness of the first order predicate calculus. 1999. Unpublished description of formalisation in Isabelle/HOL of [WW92].

[MP99] J. McKinna and R. Pollack. Some lambda calculus and type theory formalized. *Journal of Automated Reasoning*, 23(3–4), November 1999.

[Per96] Henrik Persson. Constructive completeness of intuitionistic predicate logic. 1996. http://www.cs.chalmers.se/Cs/Research/Logic/publications.mhtml.

[Pfe00] Frank Pfenning. Structural cut elimination I. intuitionistic and classical logic. *Information and Computation*, 157(1/2):84–141, March 2000.

[Pol95] Robert Pollack. A verified typechecker. In M.Dezani-Ciancaglini and G.Plotkin, editors, *Proceedings of the Second International Conference on Typed Lambda Calculi and Applications, TLCA'95, Edinburgh*, volume 902 of *LNCS*. Springer-Verlag, April 1995.

[Qui62] Willard Van Orman Quine. *Mathematical Logic*. Harper and Row, 1962.

[Rid] Tom Ridge. Informatics homepage. http://homepages.inf.ed.ac.uk/s0128214/.

[Sha94] N. Shankar. *Metamathematics, Machines, and Gödel's Proof*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, UK, 1994. http://www.csl.sri.com/users/shankar/goedel-book.html.

[T.F92] T.F. Melham. The HOL logic extended with quantification over type variables. In L.J.M. Claesen and M.J.C. Gordon, editors, *International Workshop on Higher Order Logic Theorem Proving and its Applications*, pages 3–18, Leuven, Belgium, 1992. North-Holland.

[Und95] Judith Underwood. Tableau for intuitionistic predicate logic as metatheory. In Peter Baumgartner, Reiner Hähnle, and Joachim Posegga, editors, *Theorem Proving with Analytic Tableaux and Related Methods*, volume 918 of *Lecture Notes in Artificial Intelligence*. Springer, 1995. http://www.lfcs.inf.ed.ac.uk/reports/95/ECS-LFCS-95-321/.

[Wei01] Klaus Weich. *Improving Proof Search in Intuitionistic Propositional Logic*. Logos-Verlag, 2001. http://www.logos-verlag.de/cgi-bin/engbuchmid?isbn=767&lng=eng&id=.

[WN04] Martin Wildmoser and Tobias Nipkow. Certifying machine code safety: Shallow versus deep embedding. In K. Slind, A. Bunker, and G. Gopalakrishnan, editors, *Theorem Proving in Higher Order Logics (TPHOLs 2004)*, volume 3223, pages 305–320, 2004.

[WW92] S. S. Wainer and L. A. Wallen. Basic proof theory. In S. S. Wainer, P. Aczel, and H. Simmons, editors, *Proof Theory: A Selection of Papers from the Leeds Proof Theory Programme 1990*, pages 3–26. Cambridge University Press, Cambridge, 1992.

[ZTA+] Richard Zach, Neil Tennant, Arnon Avron, Michael Kremer, Charles Parsons, and Timothy Y. Chow. The rule of generalization in fol, and pseudo-theorems. Thread on the FOM mailing list. http://www.cs.nyu.edu/pipermail/fom/2004-September/008513.html.