

A Mechanically Verified, Efficient, Sound and Complete Theorem Prover For First Order Logic

Tom Ridge

March 8, 2026

Abstract

Building on work by Wainer and Wallen, formalised by James Margetson, we present soundness and completeness proofs for a system of first order logic. The completeness proofs naturally suggest an algorithm to derive proofs. This algorithm can be implemented in a tail recursive manner. We provide the formalisation in Isabelle/HOL. The algorithm can be executed via the rewriting tactics of Isabelle. Alternatively, we transport the definitions to OCaml, to give a directly executable program.

Contents

1	Introduction	2
2	Formalisation	2
2.1	Formulas	2
2.2	Derivations	4
2.3	Failing path	6
2.4	Models	7
2.5	Soundness	8
2.6	Contains, Considers	9
2.7	Models 2	11
2.8	Falsifying Model From Failing Path	11
2.9	Completeness	11
2.10	Sound and Complete	12
2.11	Algorithm	12
2.12	Computation	12
3	Optimisation and Extension	13
4	OCaml Implementation	14

1 Introduction

Wainer and Wallen gave soundness and completeness proofs for first order logic in [3]. This material was later formalised by James Margetson [1]. We ported this to the current version of Isabelle in [2]. Drawing on some of the proofs in previous versions, especially the proof of soundness for the $\forall I$ rule, we formalise modified proofs, for a related system. Implicit in [3], and noted by Margetson in [1], is that the proofs of completeness suggest a constructive algorithm. We derive this algorithm, which turns out to be tail recursive, and this is the origin of our claim for efficiency. The algorithm can be executed in Isabelle using the rewriting engine. Alternatively, we provide an implementation in Ocaml.

2 Formalisation

```
theory Prover
imports Main
begin
```

2.1 Formulas

```
type-synonym pred = nat
```

```
type-synonym var = nat
```

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

```
primrec preSuc :: nat list  $\Rightarrow$  nat list
```

```
where
```

```
  preSuc [] = []
| preSuc (a#list) = (case a of 0  $\Rightarrow$  preSuc list | Suc n  $\Rightarrow$  n#(preSuc list))
```

```
primrec fv :: form  $\Rightarrow$  var list — shouldn't need to be more constructive than this
```

```
where
```

```
  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)
```

definition

$bump :: (var \Rightarrow var) \Rightarrow (var \Rightarrow var)$ — substitute a different var for 0 **where**
 $bump \varphi y = (case\ y\ of\ 0 \Rightarrow 0 \mid Suc\ n \Rightarrow Suc\ (\varphi\ n))$

primrec $subst :: (nat \Rightarrow nat) \Rightarrow form \Rightarrow form$

where

$subst\ r\ (PAtom\ p\ vs) = (PAtom\ p\ (map\ r\ vs))$
 $| subst\ r\ (NAtom\ p\ vs) = (NAtom\ p\ (map\ r\ vs))$
 $| subst\ r\ (FConj\ f\ g) = FConj\ (subst\ r\ f)\ (subst\ r\ g)$
 $| subst\ r\ (FDisj\ f\ g) = FDisj\ (subst\ r\ f)\ (subst\ r\ g)$
 $| subst\ r\ (FAll\ f) = FAll\ (subst\ (bump\ r)\ f)$
 $| subst\ r\ (FEx\ f) = FEx\ (subst\ (bump\ r)\ f)$

lemma $size\text{-}subst[simp]: \forall m. size\ (subst\ m\ f) = size\ f$
 $\langle proof \rangle$

definition

$fst :: form \Rightarrow var \Rightarrow form$ **where**
 $fst\ body\ w = (subst\ (\lambda v. case\ v\ of\ 0 \Rightarrow w \mid Suc\ n \Rightarrow n)\ body)$

lemma $size\text{-}fst[simp]: size\ (fst\ f\ m) = size\ f$
 $\langle proof \rangle$

type-synonym $seq = form\ list$

type-synonym $nform = nat * form$

type-synonym $nseq = nform\ list$

definition

$s\text{-of}\text{-}ns :: nseq \Rightarrow seq$ **where**
 $s\text{-of}\text{-}ns\ ns = map\ snd\ ns$

definition

$ns\text{-of}\text{-}s :: seq \Rightarrow nseq$ **where**
 $ns\text{-of}\text{-}s\ s = map\ (\lambda x. (0, x))\ s$

definition

$sfv :: seq \Rightarrow var\ list$ **where**
 $sfv\ s = concat\ (map\ fsv\ s)$

lemma $sfv\text{-}nil: sfv\ [] = []$
 $\langle proof \rangle$

lemma $sfv\text{-}cons: sfv\ (a\#\ list) = (fv\ a) @ (sfv\ list)$
 $\langle proof \rangle$

primrec $maxvar :: var\ list \Rightarrow var$
where

$maxvar \ [] = 0$
 $| \ maxvar \ (a\#list) = \max a \ (maxvar \ list)$

lemma *maxvar*: $\forall v \in \text{set } vs. v \leq maxvar \ vs$
 $\langle \text{proof} \rangle$

definition

$newvar :: \text{var } list \Rightarrow \text{var } \mathbf{where}$
 $newvar \ vs = (\text{if } vs = [] \ \text{then } 0 \ \text{else } Suc \ (maxvar \ vs))$
 — note that for newvar to be constructive, need an operation to get a different var from a given set

lemma *newvar*: $newvar \ vs \notin (\text{set } vs)$
 $\langle \text{proof} \rangle$

primrec *subs* :: $nseq \Rightarrow nseq \ list$

where

$subs \ [] = [[]]$
 $| \ subs \ (x\#xs) =$
 $(\text{let } (m,f) = x \ \text{in}$
 $\quad \text{case } f \ \text{of}$
 $\quad \quad PAtom \ p \ vs \Rightarrow \text{if } NAtom \ p \ vs \in \text{set } (\text{map } snd \ xs) \ \text{then } [] \ \text{else}$
 $\quad \quad [xs@[(0, PAtom \ p \ vs)]]$
 $\quad \quad | \ NAtom \ p \ vs \Rightarrow \text{if } PAtom \ p \ vs \in \text{set } (\text{map } snd \ xs) \ \text{then } [] \ \text{else}$
 $\quad \quad [xs@[(0, NAtom \ p \ vs)]]$
 $\quad \quad | \ FConj \ f \ g \Rightarrow [xs@[(0,f)], xs@[(0,g)]]$
 $\quad \quad | \ FDisj \ f \ g \Rightarrow [xs@[(0,f)], (0,g)]]$
 $\quad \quad | \ FAll \ f \Rightarrow [xs@[(0, \text{finst } f \ (newvar \ (sfv \ (s-of-ns \ (x\#xs)))))]]$
 $\quad \quad | \ FEx \ f \Rightarrow [xs@[(0, \text{finst } f \ m), (Suc \ m, FEx \ f)]]$
 $\quad)$

2.2 Derivations

primrec *is-axiom* :: $seq \Rightarrow \text{bool}$

where

$is-axiom \ [] = \text{False}$
 $| \ is-axiom \ (a\#list) = ((\exists p \ vs. a = PAtom \ p \ vs \wedge NAtom \ p \ vs \in \text{set } list) \vee (\exists p \ vs. a = NAtom \ p \ vs \wedge PAtom \ p \ vs \in \text{set } list))$

inductive-set

$deriv :: nseq \Rightarrow (\text{nat} * nseq) \ \text{set}$

for $s :: nseq$

where

$init: (0, s) \in deriv \ s$
 $| \ step: (n, x) \in deriv \ s \Longrightarrow y \in \text{set } (subs \ x) \Longrightarrow (Suc \ n, y) \in deriv \ s$
 — the closure of the branch at isaxiom

inductive-cases *Suc-derivE*: $(Suc \ n, x) \in deriv \ s$

declare *init* [*simp, intro*]
declare *step* [*intro*]

lemma *patom*: $(n, (m, PAtom\ p\ vs) \# xs) \in deriv(nfs) \implies \neg is-axiom\ (s-of-ns\ ((m, PAtom\ p\ vs) \# xs)) \implies (Suc\ n, xs@[0, PAtom\ p\ vs]) \in deriv(nfs)$
and *natom*: $(n, (m, NAtom\ p\ vs) \# xs) \in deriv(nfs) \implies \neg is-axiom\ (s-of-ns\ ((m, NAtom\ p\ vs) \# xs)) \implies (Suc\ n, xs@[0, NAtom\ p\ vs]) \in deriv(nfs)$
and *fconj1*: $(n, (m, FConj\ f\ g) \# xs) \in deriv(nfs) \implies \neg is-axiom\ (s-of-ns\ ((m, FConj\ f\ g) \# xs)) \implies (Suc\ n, xs@[0, f]) \in deriv(nfs)$
and *fconj2*: $(n, (m, FConj\ f\ g) \# xs) \in deriv(nfs) \implies \neg is-axiom\ (s-of-ns\ ((m, FConj\ f\ g) \# xs)) \implies (Suc\ n, xs@[0, g]) \in deriv(nfs)$
and *fdisj*: $(n, (m, FDisj\ f\ g) \# xs) \in deriv(nfs) \implies \neg is-axiom\ (s-of-ns\ ((m, FDisj\ f\ g) \# xs)) \implies (Suc\ n, xs@[0, f], (0, g]) \in deriv(nfs)$
and *fall*: $(n, (m, Fall\ f) \# xs) \in deriv(nfs) \implies \neg is-axiom\ (s-of-ns\ ((m, Fall\ f) \# xs)) \implies (Suc\ n, xs@[0, finst\ f\ (newvar\ (sfv\ (s-of-ns\ ((m, Fall\ f) \# xs))))]) \in deriv(nfs)$
and *fex*: $(n, (m, FEx\ f) \# xs) \in deriv(nfs) \implies \neg is-axiom\ (s-of-ns\ ((m, FEx\ f) \# xs)) \implies (Suc\ n, xs@[0, finst\ f\ m], (Suc\ m, FEx\ f]) \in deriv(nfs)$
<proof>

lemma *deriv0*[*simp*]: $(0, x) \in deriv\ y \longleftrightarrow (x = y)$
<proof>

lemma *deriv-exists*:
assumes $(n, x) \in deriv\ s\ x \neq [] \neg is-axiom\ (s-of-ns\ x)$
shows $\exists y. (Suc\ n, y) \in deriv\ s \wedge y \in set\ (subs\ x)$
<proof>

lemma *deriv-upwards*: $(n, list) \in deriv\ s \implies \neg is-axiom\ (s-of-ns\ (list)) \implies (\exists zs. (Suc\ n, zs) \in deriv\ s \wedge zs \in set\ (subs\ list))$
<proof>

lemma *deriv-downwards*:
assumes $(Suc\ n, x) \in deriv\ s$
shows $\exists y. (n, y) \in deriv\ s \wedge x \in set\ (subs\ y) \wedge \neg is-axiom\ (s-of-ns\ y)$
<proof>

lemma *deriv-deriv-child*: $(Suc\ n, x) \in deriv\ y = (\exists z. z \in set\ (subs\ y) \wedge \neg is-axiom\ (s-of-ns\ y) \wedge (n, x) \in deriv\ z)$
<proof>

lemmas *not-is-axiom-subst* = *patom natom fconj1 fconj2 fdisj fall fex*

lemma *deriv-progress*:
assumes $(n, a \# list) \in deriv\ s$
and $\neg is-axiom\ (s-of-ns\ (a \# list))$
shows $\exists zs. (Suc\ n, list\ @\ zs) \in deriv\ s$
<proof>

definition

$inc :: nat * nseq \Rightarrow nat * nseq$ **where**

$inc = (\lambda(n,fs). (Suc\ n, fs))$

lemma *deriv*: $deriv\ y = insert\ (0,y)\ (inc\ ' (Union\ (deriv\ ' \{w.\ \neg\ is_axiom\ (s_of_ns\ y)\ \wedge\ w \in\ set\ (subs\ y)\})))$

$\langle proof \rangle$

lemma *deriv-is-axiom*: $is_axiom\ (s_of_ns\ s) \Longrightarrow deriv\ s = \{(0,s)\}$

$\langle proof \rangle$

lemma *is-axiom-finite-deriv*: $is_axiom\ (s_of_ns\ s) \Longrightarrow finite\ (deriv\ s)$

$\langle proof \rangle$

2.3 Failing path

primrec *failing-path* :: $(nat * nseq)\ set \Rightarrow nat \Rightarrow (nat * nseq)$

where

$failing_path\ ns\ 0 = (SOME\ x.\ x \in ns \wedge fst\ x = 0 \wedge infinite\ (deriv\ (snd\ x)) \wedge \neg is_axiom\ (s_of_ns\ (snd\ x)))$

$| failing_path\ ns\ (Suc\ n) = (let\ fn = failing_path\ ns\ n\ in$

$(SOME\ fsucn.\ fsucn \in ns \wedge fst\ fsucn = Suc\ n \wedge (snd\ fsucn) \in set\ (subs\ (snd\ fn)) \wedge infinite\ (deriv\ (snd\ fsucn)) \wedge \neg is_axiom\ (s_of_ns\ (snd\ fsucn))))$

locale *FailingPath* =

fixes s **and** f

assumes inf : $infinite\ (deriv\ s)$

assumes f : $f = failing_path\ (deriv\ s)$

begin

lemma $f0$: $f\ 0 \in (deriv\ s) \wedge fst\ (f\ 0) = 0 \wedge$

$infinite\ (deriv\ (snd\ (f\ 0))) \wedge \neg is_axiom\ (s_of_ns\ (snd\ (f\ 0)))$

$\langle proof \rangle$

lemma $fSuc$:

assumes fn : $f\ n \in deriv\ s \wedge fst\ (f\ n) = n$

and inf : $infinite\ (deriv\ (snd\ (f\ n)))$

and $\neg is_axiom\ (s_of_ns\ (snd\ (f\ n)))$

shows $f\ (Suc\ n) \in deriv\ s \wedge fst\ (f\ (Suc\ n)) = Suc\ n \wedge snd\ (f\ (Suc\ n)) \in set\ (subs\ (snd\ (f\ n))) \wedge infinite\ (deriv\ (snd\ (f\ (Suc\ n)))) \wedge \neg is_axiom\ (s_of_ns\ (snd\ (f\ (Suc\ n))))$

$\langle proof \rangle$

lemma *is-path-f-0*: $f\ 0 = (0,s)$

$\langle proof \rangle$

lemma *is-path-f'*: $f\ n \in deriv\ s \wedge fst\ (f\ n) = n \wedge infinite\ (deriv\ (snd\ (f\ n))) \wedge \neg is_axiom\ (s_of_ns\ (snd\ (f\ n)))$

<proof>

lemma *is-path-f*: $f\ n \in \text{deriv } s \wedge \text{fst } (f\ n) = n \wedge (\text{snd } (f\ (\text{Suc } n))) \in \text{set } (\text{subs } (\text{snd } (f\ n))) \wedge \text{infinite } (\text{deriv } (\text{snd } (f\ n)))$

<proof>

end

2.4 Models

typedecl *U*

type-synonym *model* = $U\ \text{set} * (\text{pred} \Rightarrow U\ \text{list} \Rightarrow \text{bool})$

type-synonym *env* = $\text{var} \Rightarrow U$

primrec *FEval* :: $\text{model} \Rightarrow \text{env} \Rightarrow \text{form} \Rightarrow \text{bool}$

where

$FEval\ MI\ e\ (PAtom\ P\ vs) = (\text{let } IP = (\text{snd } MI)\ P\ \text{in } IP\ (\text{map } e\ vs))$
| $FEval\ MI\ e\ (NAtom\ P\ vs) = (\text{let } IP = (\text{snd } MI)\ P\ \text{in } \neg (IP\ (\text{map } e\ vs)))$
| $FEval\ MI\ e\ (FConj\ f\ g) = ((FEval\ MI\ e\ f) \wedge (FEval\ MI\ e\ g))$
| $FEval\ MI\ e\ (FDisj\ f\ g) = ((FEval\ MI\ e\ f) \vee (FEval\ MI\ e\ g))$
| $FEval\ MI\ e\ (FAll\ f) = (\forall m \in (\text{fst } MI). FEval\ MI\ (\lambda y. \text{case } y\ \text{of } 0 \Rightarrow m \mid \text{Suc } n \Rightarrow e\ n)\ f)$
| $FEval\ MI\ e\ (FEx\ f) = (\exists m \in (\text{fst } MI). FEval\ MI\ (\lambda y. \text{case } y\ \text{of } 0 \Rightarrow m \mid \text{Suc } n \Rightarrow e\ n)\ f)$

lemma *preSuc[simp]*: $\text{Suc } n \in \text{set } A = (n \in \text{set } (\text{preSuc } A))$

<proof>

lemma *FEval-cong*: $(\forall x \in \text{set } (fv\ A). e1\ x = e2\ x) \Longrightarrow FEval\ MI\ e1\ A = FEval\ MI\ e2\ A$

<proof>

primrec *SEval* :: $\text{model} \Rightarrow \text{env} \Rightarrow \text{form list} \Rightarrow \text{bool}$

where

$SEval\ m\ e\ [] = \text{False}$
| $SEval\ m\ e\ (x\#\!xs) = (FEval\ m\ e\ x \vee SEval\ m\ e\ xs)$

lemma *SEval-def2*: $SEval\ m\ e\ s = (\exists f. f \in \text{set } s \wedge FEval\ m\ e\ f)$

<proof>

lemma *SEval-append*: $SEval\ m\ e\ (xs\@\!ys) \longleftrightarrow SEval\ m\ e\ xs \vee SEval\ m\ e\ ys$

<proof>

lemma *SEval-cong*: $(\forall x \in \text{set } (sfv\ s). e1\ x = e2\ x) \Longrightarrow SEval\ m\ e1\ s = SEval\ m\ e2\ s$

<proof>

definition

$is-env :: model \Rightarrow env \Rightarrow bool$
where $is-env MI e \equiv (\forall x. e x \in (fst MI))$

definition

$Svalid :: form list \Rightarrow bool$
where $Svalid s \equiv (\forall MI e. is-env MI e \longrightarrow SEval MI e s)$

2.5 Soundness

lemma $FEval-subst: (FEval MI e (subst f A)) = (FEval MI (e \circ f) A)$
 $\langle proof \rangle$

lemma $FEval-finst: FEval mo e (finst A u) = FEval mo (case-nat (e u) e) A$
 $\langle proof \rangle$

lemma $sound-FAll:$

assumes $u \notin set (sfv (FAll f \# s))$
and $Svalid (s @ [finst f u])$
shows $Svalid (FAll f \# s)$
 $\langle proof \rangle$

lemma $sound-FEx: Svalid (s@[finst f u, FEx f]) \Longrightarrow Svalid (FEx f \# s)$
 $\langle proof \rangle$

lemma $inj-inc: inj inc$
 $\langle proof \rangle$

lemma $finite-inc: finite (inc ` X) = finite X$
 $\langle proof \rangle$

lemma $finite-deriv-deriv: finite (deriv s) \Longrightarrow finite (deriv ` \{w. \neg is-axiom (s-of-ns s) \wedge w \in set (subs s)\})$
 $\langle proof \rangle$

definition

$init :: nseq \Rightarrow bool$ **where**
 $init s = (\forall x \in (set s). fst x = 0)$

definition

$is-FEx :: form \Rightarrow bool$ **where**
 $is-FEx f = (case f of$
 $\quad PAtom p vs \Rightarrow False$
 $\quad | NAtom p vs \Rightarrow False$
 $\quad | FConj f g \Rightarrow False$
 $\quad | FDisj f g \Rightarrow False$

| $FAll\ f \Rightarrow False$
| $FEx\ f \Rightarrow True$

lemma *is-FEx[simp]*: $\neg is-FEx\ (PAtom\ p\ vs)$
 $\wedge \neg is-FEx\ (NAtom\ p\ vs)$
 $\wedge \neg is-FEx\ (FConj\ f\ g)$
 $\wedge \neg is-FEx\ (FDisj\ f\ g)$
 $\wedge \neg is-FEx\ (FAll\ f)$
 $\langle proof \rangle$

lemma *index0*: $\llbracket init\ s; (n, u) \in deriv\ s; (m, A) \in set\ u; \neg is-FEx\ A \rrbracket \Longrightarrow m = 0$
 $\langle proof \rangle$

lemma *soundness'*:
assumes $init\ s \wedge y\ u. (y, u) \in deriv\ s \Longrightarrow y \leq m$
shows $\llbracket h = m - n; (n, t) \in deriv\ s \rrbracket \Longrightarrow Svalid\ (s-of-ns\ t)$
 $\langle proof \rangle$

lemma *s-of-ns-inverse[simp]*: $s-of-ns\ (ns-of-s\ s) = s$
 $\langle proof \rangle$

lemma *soundness*:
assumes $finite\ (deriv\ (ns-of-s\ s))$ **shows** $Svalid\ s$
 $\langle proof \rangle$

2.6 Contains, Considers

definition
 $contains :: (nat \Rightarrow (nat*nseq)) \Rightarrow nat \Rightarrow nform \Rightarrow bool$ **where**
 $contains\ f\ n\ nf \equiv (nf \in set\ (snd\ (f\ n)))$

definition
 $considers :: (nat \Rightarrow (nat*nseq)) \Rightarrow nat \Rightarrow nform \Rightarrow bool$ **where**
 $considers\ f\ n\ nf \equiv (case\ snd\ (f\ n)\ of\ [] \Rightarrow False \mid (x\#\#xs) \Rightarrow x = nf)$

context *FailingPath*
begin

lemma *progress*:
assumes $snd\ (f\ n) = a\ \#\ list$
shows $\exists\ zs'.\ snd\ (f\ (Suc\ n)) = list\ @\ zs'$
 $\langle proof \rangle$

lemma *contains-considers'*:
shows $snd\ (f\ n) = xs@y\#\#ys \Longrightarrow \exists\ m\ zs'.\ snd\ (f\ (n+m)) = y\#\#zs'$
 $\langle proof \rangle$

lemma *contains-considers*:
 $contains\ f\ n\ y \Longrightarrow (\exists\ m.\ considers\ f\ (n+m)\ y)$

$\langle proof \rangle$

lemma *contains-propagates-patoms:*

contains f n (0, PAtom p vs) \implies contains f (n+q) (0, PAtom p vs)
 $\langle proof \rangle$

The same proof as above

lemma *contains-propagates-natoms:*

contains f n (0, NAtom p vs) \implies contains f (n+q) (0, NAtom p vs)
 $\langle proof \rangle$

lemma *contains-propagates-fconj:*

assumes *contains f n (0, FConj g h)*
shows $\exists y. \text{contains } f (n + y) (0, g) \vee \text{contains } f (n + y) (0, h)$
 $\langle proof \rangle$

lemma *contains-propagates-fdisj:*

assumes *contains f n (0, FDisj g h)*
shows $\exists y. \text{contains } f (n + y) (0, g) \wedge \text{contains } f (n + y) (0, h)$
 $\langle proof \rangle$

lemma *contains-propagates-fall:*

assumes *contains f n (0, Fall g)*
shows $\exists y. \text{contains } f (Suc(n+y)) (0, \text{fst } g (\text{newvar } (sfv (s\text{-of-ns } (snd (f (n+y)))))))$
 $\langle proof \rangle$

lemma *contains-propagates-fex:*

assumes *contains f n (m, FEx g)*
shows $\exists y. \text{contains } f (n + y) (0, \text{fst } g m) \wedge \text{contains } f (n + y) (Suc m, FEx g)$
 $\langle proof \rangle$

lemma *FEx-downward:*

assumes *init s*
shows $(Suc m, FEx g) \in \text{set } (snd (f n)) \implies (\exists n'. (m, FEx g) \in \text{set } (snd (f n')))$
 $\langle proof \rangle$

lemma *FEx0:*

assumes *init s*
shows *contains f n (m, FEx g) \implies ($\exists n'. \text{contains } f n' (0, FEx g)$)*
 $\langle proof \rangle$

lemma *FEx-upward':*

assumes *contains f n (0, FEx g)*
shows $\exists n'. \text{contains } f n' (m, FEx g)$
 $\langle proof \rangle$

lemma *FEx-upward:*

assumes *init s*
and *contains f n (m, FEx g)*
shows $\exists n'. \text{contains } f \ n' \ (0, \text{finst } g \ m')$
 ⟨*proof*⟩

end

2.7 Models 2

axiomatization *ntou :: nat \Rightarrow U*
where *ntou: inj ntou* — assume universe set is infinite

definition *uton :: U \Rightarrow nat* **where** *uton = inv ntou*

lemma *uton-ntou: uton (ntou x) = x*
 ⟨*proof*⟩

lemma *map-uton-ntou[simp]: map uton (map ntou xs) = xs*
 ⟨*proof*⟩

lemma *ntou-uton: x \in range ntou \implies ntou (uton x) = x*
 ⟨*proof*⟩

2.8 Falsifying Model From Failing Path

definition *model :: nseq \Rightarrow model* **where**
model s \equiv
(range ntou,
$\lambda p \ ms. \text{let } f = \text{failing-path } (\text{deriv } s) \text{ in}$
$\forall n \ m. \neg \text{contains } f \ n \ (m, \text{PAtom } p \ (\text{map } \text{uton } ms))$)

lemma *is-env-model-ntou: is-env (model s) ntou*
 ⟨*proof*⟩

lemma **(in FailingPath)** *[simp]:*
[[init s; contains f n (m,A); \neg is-FEx A]] \implies m = 0
 ⟨*proof*⟩

lemma **(in FailingPath)** *model':*
assumes *init s*
and *A: h = size A contains f n (m, A) FEval (model s) ntou A*
shows $\neg \text{FEval } (\text{model } s) \ \text{ntou } A$
 ⟨*proof*⟩

2.9 Completeness

lemma *completeness':*
assumes *infinite (deriv s) init s (m,A) \in set s*
shows $\neg \text{FEval } (\text{model } s) \ \text{ntou } A$
 ⟨*proof*⟩

lemma completeness:
assumes *infinite* (*deriv* (*ns-of-s* *s*))
shows \neg *Svalid* *s*
 \langle *proof* \rangle

2.10 Sound and Complete

lemma *Svalid* *s* = *finite* (*deriv* (*ns-of-s* *s*))
 \langle *proof* \rangle

2.11 Algorithm

lemma *ex-iter'*: $(\exists n. R ((g \hat{\sim} n) a)) = (R a \vee (\exists n. R ((g \hat{\sim} n)(g a))))$
 \langle *proof* \rangle

lemma *ex-iter*: $(\exists n. R ((g \hat{\sim} n) a)) = (\text{if } R a \text{ then True else } (\exists n. R ((g \hat{\sim} n)(g a))))$
 \langle *proof* \rangle

definition

f :: *nseq list* \Rightarrow *nat* \Rightarrow *nseq list* **where**
f *s* *n* $\equiv ((\lambda x. \text{concat} (\text{map} \text{subs } x)) \hat{\sim} n) s$

lemma *f-upwards*: *f* *s* *n* = [] \implies *f* *s* (*n+m*) = []
 \langle *proof* \rangle

lemma *f*: $((n, x) \in \text{deriv } s) = (x \in \text{set} (f [s] n))$
 \langle *proof* \rangle

lemma *deriv-f*: *deriv* *s* = $(\bigcup x. \text{set} (\text{map} (\text{Pair } x) (f [s] x)))$
 \langle *proof* \rangle

lemma *finite-deriv*: *finite* (*deriv* *s*) $\longleftrightarrow (\exists m. f [s] m = [])$
 \langle *proof* \rangle

definition *prove'* :: *nseq list* \Rightarrow *bool* **where**
prove' *s* $\longleftrightarrow (\exists m. ((\text{concat} \circ \text{map} \text{subs}) \hat{\sim} m) s = [])$

lemma *prove'*:
prove' *l* $\longleftrightarrow l = [] \vee \text{prove}' (\text{concat} (\text{map} \text{subs } l))$
 \langle *proof* \rangle

definition *prove* :: *nseq* \Rightarrow *bool*
where *prove* *s* = *prove'* ([*s*])

lemma *finite-deriv-prove*: *finite* (*deriv* *s*) = *prove* *s*
 \langle *proof* \rangle

2.12 Computation

lemma $(\exists x. A x \vee B x) \longrightarrow ((\exists x. B x) \vee (\exists x. A x))$

<proof>
lemma $((\exists x. A x \vee B x) \longrightarrow ((\exists x. B x) \vee (\exists x. A x)))$
 $= ((\forall x. \neg A x \wedge \neg B x) \vee ((\exists x. B x) \vee (\exists x. A x)))$
<proof>

definition *my-f* :: *form where*

my-f = *FDisj*
(FAll (FConj (NAtom 0 [0]) (NAtom 1 [0])))
(FDisj (FEx (PAtom 1 [0])) (FEx (PAtom 0 [0])))

— we compute by rewriting

lemma *membership-simps*:

$x \in \text{set } [] \longleftrightarrow \text{False}$
 $x \in \text{set } (y \# ys) \longleftrightarrow x = y \vee x \in \text{set } ys$
<proof>

lemmas *ss = list.inject if-True if-False concat.simps list.map*
sfv-def filter.simps snd-conv form.simps first-def s-of-ns-def
Let-def newvar-def subs.simps split-beta append-Nil append-Cons
subst.simps nat.simps fv.simps maxvar.simps preSuc.simps simp-thms
membership-simps

lemmas *prove'-Nil = prove' [of [], simplified]*

lemmas *prove'-Cons = prove' [of x#l, simplified] for x l*

lemma *search: finite (deriv [(0,my-f)])*
<proof>

abbreviation *Sprove* :: *form list* \Rightarrow *bool where* *Sprove* \equiv *prove* \circ *ns-of-s*

abbreviation *check* :: *form* \Rightarrow *bool where* *check formula* \equiv *Sprove [formula]*

abbreviation *valid* :: *form* \Rightarrow *bool where* *valid formula* \equiv *Svalid [formula]*

theorem *check = valid* *<proof>*

<ML>

end

3 Optimisation and Extension

There are plenty of obvious optimisations. The first medium level optimisation is to avoid the recomputation of newvars by incorporating the maxvar into a sequent. At a low level, most of the list operations are just moving a pointer along a list: only FConj requires duplicating a list. Reporting “not

provable” on obviously non-provable goals would be useful, as would a more efficient choice of witnessing terms for existentials.

In terms of extensions, the obvious targets are function terms and equality.

4 OCaml Implementation

```
open List;;

type pred = int;;

type var = int;;

type form =
  PAtom of (pred*(var list))
  | NAtom of (pred*(var list))
  | FConj of form * form
  | FDisj of form * form
  | FAll of form
  | FEx of form
;;

let rec preSuc t = match t with
  [] -> []
  | (a::list) -> (match a with 0 -> preSuc list | sucn -> (sucn-1::preSuc list));;

let rec fv t = match t with
  PAtom (p,vs) -> vs
  | NAtom (p,vs) -> vs
  | FConj (f,g) -> (fv f)@(fv g)
  | FDisj (f,g) -> (fv f)@(fv g)
  | FAll f -> preSuc (fv f)
  | FEx f -> preSuc (fv f);;

let suc x = x+1;;

let bump phi y = match y with 0 -> 0 | sucn -> suc (phi (sucn-1));;

let rec subst r f = match f with
  PAtom (p,vs) -> PAtom (p,map r vs)
  | NAtom (p,vs) -> NAtom (p,map r vs)
  | FConj (f,g) -> FConj (subst r f, subst r g)
  | FDisj (f,g) -> FDisj (subst r f, subst r g)
```

```

    | FALL f -> FALL (subst (bump r) f)
    | FEx f -> FEx (subst (bump r) f);;

let finst body w = subst (fun v -> match v with 0 -> w | sucn -> (sucn-
1)) body;;

let s_of_ns ns = map snd ns;;

let sfv s = flatten (map fv s);;

let rec maxvar t = match t with
  [] -> 0
  | (a::list) -> max a (maxvar list);;

let newvar vs = suc (maxvar vs);;

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,P
      | NAtom (p,vs) -> if mem (PAtom (p,vs)) (map snd xs) then [] else [xs@[ (0,N
      | 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 rec prove' l = (if l = [] then true else prove' ((fun x -> flatten (map subs x)

let prove s = prove' [s];;

let my_f = FDisj (
  (FALL (FConj ((NAtom (0,[0])), (NAtom (1,[0])))),
  (FDisj ((FEx ((PAtom (1,[0]))),(FEx (PAtom (0,[0]))))))));;

prove [(0,my_f)];;

```

References

- [1] J. Margetson. Completeness of the first order predicate calculus. 1999.
- [2] J. Margetson and T. Ridge. Completeness of the first order predicate calculus. *Archive of Formal Proofs*, 2004.

- [3] 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.