

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

Tom Ridge

December 12, 2009

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	10
2.7	Models 2	11
2.8	Falsifying Model From Failing Path	12
2.9	Completeness	12
2.10	Sound and Complete	13
2.11	Algorithm	13
2.12	Computation	14
3	Optimisation and Extension	15
4	OCaml Implementation	15

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 Infinite-Set begin
```

2.1 Formulas

```
types pred = nat
```

```
types 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 => nat list
```

```
where
```

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

```
primrec fv :: form => 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**
 $bump\ phi\ y = (case\ y\ of\ 0 \Rightarrow 0 \mid Suc\ n \Rightarrow Suc\ (phi\ n))$

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

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

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

types $seq = form\ list$

types $nform = nat * form$

types $nseq = nform\ list$

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**
 $ns\text{-of}\text{-}s\ s = map\ (\% x. (0,x))\ s$

primrec $flatten :: 'a\ list\ list \Rightarrow 'a\ list$

where

$flatten\ [] = []$
 $| flatten\ (a\#\ list) = (a\@ (flatten\ list))$

definition

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

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* => *var*

where

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

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

<proof>

definition

newvar :: *var list* => *var* **where**

newvar *vs* = *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 (*set vs*)

<proof>

primrec *subs* :: *nseq* => *nseq list*

where

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)]*
)
)

2.2 Derivations

primrec *is-axiom* :: *seq* => *bool*

where

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*))

inductive-set

deriv :: *nseq* => (*nat * nseq*) *set*

for *s* :: *nseq*

where

init: (*0,s*) \in *deriv s*

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

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

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

lemmas[*simp*] = *init*

lemmas [*intro*] = *init step*

lemma *deriv0[simp]*: $(0, x) \in \text{deriv } y = (x = y)$
 ⟨*proof*⟩

lemma *deriv-upwards*: $(n, \text{list}) \in \text{deriv } s \implies \sim \text{is-axiom } (s\text{-of-ns } (\text{list})) \implies (\exists zs. (\text{Suc } n, zs) \in \text{deriv } s \ \& \ zs : \text{set } (\text{subs } \text{list}))$
 ⟨*proof*⟩

lemma *deriv-downwards* : $(\text{Suc } n, x) \in \text{deriv } s \implies \exists y. (n, y) \in \text{deriv } s \ \& \ x : \text{set } (\text{subs } y) \ \& \ \sim \text{is-axiom } (s\text{-of-ns } y)$
 ⟨*proof*⟩

lemma *deriv-deriv-child[rule-format]*: $\forall x \ y. (\text{Suc } n, x) \in \text{deriv } y = (\exists z. z : \text{set } (\text{subs } y) \ \& \ \sim \text{is-axiom } (s\text{-of-ns } y) \ \& \ (n, x) \in \text{deriv } z)$
 ⟨*proof*⟩

lemma *deriv-progress*: $(n, a \# \text{list}) \in \text{deriv } s \implies \sim \text{is-axiom } (s\text{-of-ns } (a \# \text{list})) \implies (\exists zs. (\text{Suc } n, \text{list}@zs) \in \text{deriv } s)$
 ⟨*proof*⟩

definition

$inc :: nat * nseq \Rightarrow nat * nseq$ **where**
 $inc = (\%_0(n,fs). (Suc\ n, fs))$

lemma *inj-inc[simp]*: $inj\ inc$
 $\langle proof \rangle$

lemma *deriv*: $deriv\ y = insert\ (0,y)\ (inc\ ' (Union\ (deriv\ ' \{w.\ \sim is-axiom\ (s-of-ns\ y)\ \&\ w : set\ (subs\ y)\})))$
 $\langle proof \rangle$

lemma *deriv-is-axiom*: $is-axiom\ (s-of-ns\ s) \implies deriv\ s = \{(0,s)\}$
 $\langle proof \rangle$

lemma *is-axiom-finite-deriv*: $is-axiom\ (s-of-ns\ s) \implies finite\ (deriv\ s)$
 $\langle proof \rangle$

2.3 Failing path

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

where

$failing-path\ ns\ 0 = (SOME\ x.\ x \in ns\ \&\ fst\ x = 0\ \&\ infinite\ (deriv\ (snd\ x))\ \&\ \sim is-axiom\ (s-of-ns\ (snd\ x)))$
 $| failing-path\ ns\ (Suc\ n) = (let\ fn = failing-path\ ns\ n\ in$
 $(SOME\ fsucn.\ fsucn \in ns\ \&\ fst\ fsucn = Suc\ n\ \&\ (snd\ fsucn) : set\ (subs\ (snd\ fn))\ \&\ infinite\ (deriv\ (snd\ fsucn))\ \&\ \sim is-axiom\ (s-of-ns\ (snd\ fsucn))))$

locale *loc1* =
fixes s **and** f
assumes $f : f = failing-path\ (deriv\ s)$

lemma (**in** *loc1*) *f0*: $infinite\ (deriv\ s) \implies f\ 0 \in (deriv\ s)\ \&\ fst\ (f\ 0) = 0\ \&\ infinite\ (deriv\ (snd\ (f\ 0)))\ \&\ \sim is-axiom\ (s-of-ns\ (snd\ (f\ 0)))$
 $\langle proof \rangle$

lemma *infinite-union*: $finite\ Y \implies infinite\ (Union\ (f\ ' Y)) \implies \exists y.\ y \in Y\ \&\ infinite\ (f\ y)$
 $\langle proof \rangle$

lemma *infinite-inj-infinite-image*: $inj-on\ f\ Z \implies infinite\ (f\ ' Z) = infinite\ Z$
 $\langle proof \rangle$

lemma *inj-inj-on*: $inj\ f \implies inj-on\ f\ A$
 $\langle proof \rangle$

lemma *collect-disj*: $\{x.\ P\ x\ | Q\ x\} = \{x.\ P\ x\} \cup \{x.\ Q\ x\}$ $\langle proof \rangle$

lemma *t*: $\text{finite } \{w. P w\} \implies \text{finite } \{w. Q w \ \& \ P w\}$
 ⟨proof⟩

lemma *finite-subs*: $\text{finite } \{w. \sim \text{is-axiom } (s\text{-of-ns } y) \ \& \ w : \text{set } (subs \ y)\}$
 ⟨proof⟩

lemma (*in loc1*) *fSuc*:

shows []
 $f \ n \in \text{deriv } s \ \& \ \text{fst } (f \ n) = n \ \& \ \text{infinite } (\text{deriv } (\text{snd } (f \ n))) \ \& \ \sim \text{is-axiom } (s\text{-of-ns } (\text{snd } (f \ n)))$
 [] $\implies f \ (\text{Suc } n) \in \text{deriv } s \ \& \ \text{fst } (f \ (\text{Suc } n)) = \text{Suc } n \ \& \ (\text{snd } (f \ (\text{Suc } n))) : \text{set } (subs \ (\text{snd } (f \ n))) \ \& \ \text{infinite } (\text{deriv } (\text{snd } (f \ (\text{Suc } n)))) \ \& \ \sim \text{is-axiom } (s\text{-of-ns } (\text{snd } (f \ (\text{Suc } n))))$
 ⟨proof⟩

lemma (*in loc1*) *is-path-f-0*: $\text{infinite } (\text{deriv } s) \implies f \ 0 = (0, s)$
 ⟨proof⟩

lemma (*in loc1*) *is-path-f'*: $\text{infinite } (\text{deriv } s) \implies \forall n. f \ n \in \text{deriv } s \ \& \ \text{fst } (f \ n) = n \ \& \ \text{infinite } (\text{deriv } (\text{snd } (f \ n))) \ \& \ \sim \text{is-axiom } (s\text{-of-ns } (\text{snd } (f \ n)))$
 ⟨proof⟩

lemma (*in loc1*) *is-path-f*: $\text{infinite } (\text{deriv } s) \implies \forall n. f \ n \in \text{deriv } s \ \& \ \text{fst } (f \ n) = n \ \& \ (\text{snd } (f \ (\text{Suc } n))) : \text{set } (subs \ (\text{snd } (f \ n))) \ \& \ \text{infinite } (\text{deriv } (\text{snd } (f \ n)))$
 ⟨proof⟩

2.4 Models

typedecl *U*

types *model* = *U set* * (*pred* => *U list* => *bool*)

types *env* = *var* => *U*

primrec *FEval* :: *model* => *env* => *form* => *bool*

where

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

lemma *and-lem*: $(a=c) \implies (b=d) \implies (a \ \& \ b) = (c \ \& \ d)$ ⟨proof⟩

lemma *or-lem*: $(a=c) \implies (b=d) \implies (a \ | \ b) = (c \ | \ d)$ ⟨proof⟩

lemma *ball-eq-ball*: $\forall x \in m. P x = Q x \implies (\forall x \in m. P x) = (\forall x \in m. Q x)$
 $\langle proof \rangle$

lemma *bex-eq-bex*: $\forall x \in m. P x = Q x \implies (\exists x \in m. P x) = (\exists x \in m. Q x)$
 $\langle proof \rangle$

lemma *preSuc[simp]*: $\forall n. Suc n \in set A = (n \in set (preSuc A))$ $\langle proof \rangle$

lemma *FEval-cong*: $\forall e1 e2. (\forall x. x \in set (fv A) \longrightarrow e1 x = e2 x) \longrightarrow FEval MI e1 A = FEval MI e2 A$
 $\langle proof \rangle$

primrec *SEval* :: *model* => *env* => *form list* => *bool*

where

$SEval m e [] = False$
 $| SEval m e (x\#xs) = (FEval m e x | SEval m e xs)$

lemma *SEval-def2*: $SEval m e s = (\exists f. f \in set s \ \& \ FEval m e f)$
 $\langle proof \rangle$

lemma *SEval-append*: $SEval m e (xs@ys) = ((SEval m e xs) | (SEval m e ys))$
 $\langle proof \rangle$

lemma *all-eq-all*: $\forall x. P x = Q x \implies (\forall x. P x) = (\forall x. Q x)$ $\langle proof \rangle$

lemma *all-conj*: $(\forall x. A x \ \& \ B x) = ((\forall x. A x) \ \& \ (\forall x. B x))$ $\langle proof \rangle$

lemma *SEval-cong*: $(\forall x. x \in set (sfv s) \longrightarrow e1 x = e2 x) \longrightarrow SEval m e1 s = SEval m e2 s$
 $\langle proof \rangle$

definition

is-env :: *model* => *env* => *bool* **where**
 $is-env MI e = (\forall x. e x \in (fst MI))$

definition

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

2.5 Soundness

lemma *fold-compose1*: $(\% x. f (g x)) = (f o g)$
 $\langle proof \rangle$

lemma *FEval-subst*: $\forall e f. (FEval MI e (subst f A)) = (FEval MI (e o f) A)$
 $\langle proof \rangle$

lemma *FEval-finst*: $FEval\ mo\ e\ (finst\ A\ u) = FEval\ mo\ (nat\text{-}case\ (e\ u)\ e)\ A$
 ⟨proof⟩

lemma *ball-maxscope*: $(\forall x \in m. P\ x \mid Q) \implies (\forall x \in m. P\ x) \mid Q$ ⟨proof⟩

lemma *sound-FAll*: $u \notin set\ (sfv\ (FAll\ f\ \#s)) \implies Svalid\ (s@[finst\ f\ u]) \implies Svalid\ (FAll\ f\ \#s)$
 ⟨proof⟩

lemma *sound-FAll'*: $u \notin set\ (sfv\ (FAll\ f\ \#s)) \implies Svalid\ (s@[finst\ f\ u]) \implies Svalid\ (FAll\ f\ \#s)$
 ⟨proof⟩

lemma *sound-FEx*: $Svalid\ (s@[finst\ f\ u, FEx\ f]) \implies Svalid\ (FEx\ f\ \#s)$
 ⟨proof⟩

lemma *max-exists*: $finite\ (X::nat\ set) \implies X \neq \{\} \dashrightarrow (\exists x. x \in X \ \&\ (\forall y. y \in X \dashrightarrow y \leq x))$
 ⟨proof⟩

lemma *inj-finite-image-eq-finite*: $inj\text{-}on\ f\ Z \implies finite\ (f\ ' Z) = finite\ Z$
 ⟨proof⟩

lemma *finite-inc*: $finite\ (inc\ ' X) = finite\ X$
 ⟨proof⟩

lemma *finite-deriv-deriv*: $finite\ (deriv\ s) \implies finite\ (deriv\ ' \{w. \sim is\text{-}axiom\ (s\text{-}of\text{-}ns\ s) \ \&\ w : set\ (subs\ s)\})$
 ⟨proof⟩

definition

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

definition

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

lemma *is-FEx[simp]*: $\sim is\text{-}FEx\ (PAtom\ p\ vs)$

$\& \sim is-FEx (NAtom\ p\ vs)$
 $\& \sim is-FEx (FConj\ f\ g)$
 $\& \sim is-FEx (FDisj\ f\ g)$
 $\& \sim is-FEx (FAll\ f)$
 $\langle proof \rangle$

lemma *index0*: $init\ s \implies \forall u\ m\ A. (n, u) \in deriv\ s \dashrightarrow (m, A) \in (set\ u) \dashrightarrow$
 $(\sim is-FEx\ A) \dashrightarrow m = 0$
 $\langle proof \rangle$

lemma *soundness'*: $init\ s \implies finite\ (deriv\ s) \implies m \in (fst\ ' (deriv\ s)) \implies$
 $\forall y\ u. (y, u) \in (deriv\ s) \dashrightarrow y \leq m \implies \forall n\ t. h = m - n \ \& \ (n, t) \in deriv\ s$
 $\dashrightarrow Svalid\ (s-of-ns\ t)$
 $\langle proof \rangle$

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

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

2.6 Contains, Considers

definition

$contains :: (nat \implies (nat*nseq)) \implies nat \implies nform \implies bool$ **where**
 $contains\ f\ n\ nf = (nf \in set\ (snd\ (f\ n)))$

definition

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

lemma (*in loc1*) *progress*: $infinite\ (deriv\ s) \implies snd\ (f\ n) = a\#\!list \dashrightarrow (\exists\ zs'.$
 $snd\ (f\ (Suc\ n)) = list@\!zs')$
 $\langle proof \rangle$

lemma (*in loc1*) *contains-considers'*: $infinite\ (deriv\ s) \implies \forall n\ y\ ys. snd\ (f\ n)$
 $= xs@\!y\#\!ys \dashrightarrow (\exists\ m\ zs'. snd\ (f\ (n+m)) = y\#\!zs')$
 $\langle proof \rangle$

lemma *list-decomp*[*rule-format*]: $v \in set\ p \dashrightarrow (\exists\ xs\ ys. p = xs@\!(v\#\!ys) \wedge v \notin$
 $set\ xs)$
 $\langle proof \rangle$

lemma (*in loc1*) *contains-considers*: $infinite\ (deriv\ s) \implies contains\ f\ n\ y \implies$
 $(\exists\ m. considers\ f\ (n+m)\ y)$
 $\langle proof \rangle$

lemma (*in loc1*) *contains-propagates-atoms*[*rule-format*]: $infinite\ (deriv\ s) \implies$

contains f n (0, PAtom p vs) --> contains f (n+q) (0, PAtom p vs)
 ⟨proof⟩

lemma (in *loc1*) *contains-propagates-natoms*[rule-format]: *infinite (deriv s) ==> contains f n (0, NAtom p vs) --> contains f (n+q) (0, NAtom p vs)*
 ⟨proof⟩

lemma (in *loc1*) *contains-propagates-fconj*: *infinite (deriv s) ==> contains f n (0, FConj g h) ==> (∃ y. contains f (n+y) (0,g) | contains f (n+y) (0,h))*
 ⟨proof⟩

lemma (in *loc1*) *contains-propagates-fdisj*: *infinite (deriv s) ==> contains f n (0, FDisj g h) ==> (∃ y. contains f (n+y) (0,g) & contains f (n+y) (0,h))*
 ⟨proof⟩

lemma (in *loc1*) *contains-propagates-fall*: *infinite (deriv s) ==> contains f n (0, FAll g)*
 ==> (∃ y. contains f (Suc(n+y)) (0,fnst g (newvar (sfv (s-of-ns (snd (f (n+y))))))))
 — may need constraint on fv
 ⟨proof⟩

lemma (in *loc1*) *contains-propagates-fex*: *infinite (deriv s) ==> contains f n (m, FEx g)*
 ==> (∃ y.
 (contains f (n+y) (0,fnst g m))
 & (contains f (n+y) (Suc m, FEx g)))
 ⟨proof⟩

lemma (in *loc1*) *FEx-downward*: *infinite (deriv s) ==> init s ==> ∀ m. (Suc m, FEx g) ∈ set (snd (f n)) --> (∃ n'. (m, FEx g) ∈ set (snd (f n')))*
 ⟨proof⟩

lemma (in *loc1*) *FEx0*: *infinite (deriv s) ==> init s ==> ∀ n. contains f n (m, FEx g) --> (∃ n'. contains f n' (0, FEx g))*
 ⟨proof⟩

lemma (in *loc1*) *FEx-upward'*: *infinite (deriv s) ==> init s ==> ∀ n. contains f n (0, FEx g) --> (∃ n'. contains f n' (m, FEx g))*
 ⟨proof⟩

lemma (in *loc1*) *FEx-upward*: *infinite (deriv s) ==> init s ==> contains f n (m, FEx g) ==> (∀ m'. ∃ n'. contains f n' (0, fnst g m'))*
 ⟨proof⟩

2.7 Models 2

axiomatization *ntou* :: *nat => U*

where *ntou*: *inj ntou* — assume universe set is infinite

definition *uton* :: $U \Rightarrow \text{nat}$ **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*: $x \in \text{range } \textit{ntou} \implies \textit{ntou} (\textit{uton } x) = x$
<proof>

2.8 Falsifying Model From Failing Path

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

locale *loc2* = *loc1* +
fixes *mo*
assumes *mo*: *mo* = *model s*

lemma *is-env-model-ntou*: *is-env* (*model s*) *ntou*
<proof>

lemma (**in** *loc1*) [*simp*]: *infinite* (*deriv s*) $\implies \textit{init } s \implies (\textit{contains } f n (m, A))$
 $\implies \sim \textit{is-FEx } A \implies m = 0$
<proof>

lemma (**in** *loc2*) *model'*:
notes [*simp*] = *FEval-subst*
notes [*simp del*] = *is-axiom.simps*
shows *infinite* (*deriv s*) $\implies \textit{init } s \implies \forall A. \textit{size } A = h \dashrightarrow (\forall m n. \textit{contains } f n (m, A) \dashrightarrow \sim (\textit{FEval } mo \textit{ntou } A))$

<proof>

lemma (**in** *loc2*) *model*: *infinite* (*deriv s*) $\implies \textit{init } s \implies (\forall A m n. \textit{contains } f n (m, A) \dashrightarrow \sim (\textit{FEval } mo \textit{ntou } A))$
<proof>

2.9 Completeness

lemma (**in** *loc2*) *completeness'*: *infinite* (*deriv s*) $\implies \textit{init } s \implies \forall mA \in \textit{set } s. \sim \textit{FEval } mo \textit{ntou} (\textit{snd } mA)$ — FIXME tidy *deriv s* so that *s* consists of formulae only?
<proof>

thm *loc2.completeness'*[*simplified loc2-def loc2-axioms-def loc1-def*]

lemma *completeness'*: $\text{infinite } (\text{deriv } s) \implies \text{init } s \implies \forall mA \in \text{set } s. \sim \text{FEval } (\text{model } s) \text{ ntou } (\text{snd } mA)$
 ⟨proof⟩

lemma *completeness''*: $\text{infinite } (\text{deriv } (\text{ns-of-} s)) \implies \text{init } (\text{ns-of-} s) \implies \forall A. A \in \text{set } s \dashrightarrow \sim \text{FEval } (\text{model } (\text{ns-of-} s)) \text{ ntou } A$
 ⟨proof⟩

lemma *completeness*: $\text{infinite } (\text{deriv } (\text{ns-of-} s)) \implies \sim \text{Svalid } s$
 ⟨proof⟩

2.10 Sound and Complete

lemma $\text{Svalid } s = \text{finite } (\text{deriv } (\text{ns-of-} s))$
 ⟨proof⟩

2.11 Algorithm

primrec *iter* :: $('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow \text{nat} \Rightarrow 'a$ — fold for nats
where

$\text{iter } g \ a \ 0 = a$
 | $\text{iter } g \ a \ (\text{Suc } n) = g \ (\text{iter } g \ a \ n)$

lemma *iter*: $\forall a. (\text{iter } g \ (g \ a) \ n) = (g \ (\text{iter } g \ a \ n))$
 ⟨proof⟩

lemma *ex-iter'*: $(\exists n. R \ (\text{iter } g \ a \ n)) = (R \ a \ | \ (\exists n. R \ (\text{iter } g \ (g \ a) \ n)))$
 ⟨proof⟩

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

definition

$f :: \text{nseq list} \Rightarrow \text{nat} \Rightarrow \text{nseq list}$ **where**
 $f \ s \ n = \text{iter } (\% x. \text{flatten } (\text{map } \text{subs } x)) \ s \ n$

lemma *f-upwards*: $f \ s \ n = [] \implies f \ s \ (n+m) = []$
 ⟨proof⟩

lemma *flatten-append*: $\text{flatten } (xs@ys) = ((\text{flatten } xs) @ (\text{flatten } ys))$
 ⟨proof⟩

lemma *set-flatten*: $\text{set } (\text{flatten } xs) = \text{Union } (\text{set } ' \ \text{set } xs)$
 ⟨proof⟩

lemma *f*: $\forall x. ((n,x) \in \text{deriv } s) = (x \in \text{set } (f \ [s] \ n))$
 ⟨proof⟩

lemma *deriv-f*: $\text{deriv } s = \text{UNION UNIV } (\% x. \text{set } (\text{map } (\% y. (x,y)) \ (f \ [s] \ x)))$
 ⟨proof⟩

lemma *finite-f*: *finite* (set (f s x))

⟨proof⟩

lemma *finite-deriv*: *finite* (deriv s) = (∃ m. f [s] m = [])

⟨proof⟩

lemma *ex-iter-fSucn*: (∃ m. iter (% x. flat (map subs x)) l m = []) = (if l = [] then True else (∃ n. (iter (% x. flat (map subs x)) ((% x. flat (map subs x)) l) n) = []))

⟨proof⟩

definition *prove'* :: *nseq list => bool where*

prove' s = (∃ m. iter (% x. flatten (map subs x)) s m = [])

lemma *prove'*: *prove' l* = (if l = [] then True else *prove'* ((% x. flatten (map subs x)) l))

⟨proof⟩

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

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

⟨proof⟩

2.12 Computation

— a sample formula to prove

lemma (∃ x. A x | B x) --> ((∃ x. B x) | (∃ x. A x)) ⟨proof⟩

lemma ((∃ x. A x | B x) --> ((∃ x. B x) | (∃ x. A x)))
= ((∀ x. ~ A x & ~ B x) | ((∃ x. B x) | (∃ 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

lemmas *ss* = *list.simps if-True if-False flatten.simps map.simps bump-def sfv-def filter.simps is-axiom.simps fst-conv snd-conv form.simps collect-disj inc-def finst-def ns-of-s-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 mem-iff [symmetric] List.member.simps*

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

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

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

<proof>

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,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 rec prove' l = (if l = [] then true else prove' ((fun x -> flatten (map subs x)) l));;

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.