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

Tom Ridge

March 17, 2025

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

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* ⇒ *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* ⇒ *var*) ⇒ (*var* ⇒ *var*) — substitute a different var for 0 **where**
bump φ *y* = (*case y of 0* ⇒ *0* | *Suc n* ⇒ *Suc* (φ *n*))

primrec *subst* :: (*nat* ⇒ *nat*) ⇒ *form* ⇒ *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-subst*[*simp*]: ∀ *m*. *size* (*subst m f*) = *size f*
⟨*proof*⟩

definition
finst :: *form* ⇒ *var* ⇒ *form* **where**
finst body w = (*subst* (λ *v*. *case v of 0* ⇒ *w* | *Suc n* ⇒ *n*) *body*)

lemma *size-finst*[*simp*]: *size* (*finst f m*) = *size f*
⟨*proof*⟩

type-synonym *seq* = *form list*

type-synonym *nform* = *nat* ∗ *form*

type-synonym *nseq* = *nform list*

definition
s-of-ns :: *nseq* ⇒ *seq* **where**
s-of-ns ns = *map snd ns*

definition
ns-of-s :: *seq* ⇒ *nseq* **where**
ns-of-s s = *map* (λ *x*. (*0*,*x*)) *s*

definition
sfv :: *seq* ⇒ *var list* **where**
sfv s = *concat* (*map fv s*)

lemma *sfv-nil*: *sfv* [] = []
⟨*proof*⟩

lemma *sfv-cons*: *sfv* (*a*#*list*) = (*fv a*) @ (*sfv list*)
⟨*proof*⟩

primrec *maxvar* :: *var list* ⇒ *var*
where

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

lemma *maxvar*: ∀ *v* ∈ *set vs*. *v* ≤ *maxvar vs*
⟨*proof*⟩

definition
newvar :: *var list* ⇒ *var* **where**
newvar vs = (*if vs* = [] *then 0 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* ∉ (*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*) ∈ *deriv s*
| *step*: (*n*,*x*) ∈ *deriv s* ⟹ *y* ∈ *set* (*subs x*) ⟹ (*Suc n*,*y*) ∈ *deriv s*
— the closure of the branch at isaxiom

inductive-cases *Suc-derivE*: (*Suc n*, *x*) ∈ *deriv s*

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

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

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

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

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

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

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

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

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

definition
inc :: *nat* ∗ *nseq* ⇒ *nat* ∗ *nseq* **where**
inc = (λ(*n*,*fs*). (*Suc n*, *fs*))

lemma *deriv*: *deriv y* = *insert* (*0*,*y*) (*inc* ‘ (*Union* (*deriv* ‘ {*w*. ¬ *is-axiom* (*s-of-ns y*) ∧ *w* ∈ *set* (*subs y*)})))
⟨*proof*⟩

lemma *deriv-is-axiom*: *is-axiom* (*s-of-ns s*) ⟹ *deriv s* = {(*0*,*s*)}
⟨*proof*⟩

lemma *is-axiom-finite-deriv*: *is-axiom* (*s-of-ns s*) ⟹ *finite* (*deriv s*)
⟨*proof*⟩

2.3 Failing path

primrec *failing-path* :: (*nat* ∗ *nseq*) *set* ⇒ *nat* ⇒ (*nat* ∗ *nseq*)
where
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*))))

locale *FailingPath* =
fixes *s* **and** *f*
assumes *inf*: *infinite* (*deriv s*)
assumes *f*: *f* = *failing-path* (*deriv s*)

begin

lemma *f0*: *f 0* ∈ (*deriv s*) ∧ *fst* (*f 0*) = *0* ∧
infinite (*deriv* (*snd* (*f 0*))) ∧ ¬ *is-axiom* (*s-of-ns* (*snd* (*f 0*)))
⟨*proof*⟩

lemma *fSuc*:
assumes *fn*: *f n* ∈ *deriv s fst* (*f n*) = *n*
and *inf*: *infinite* (*deriv* (*snd* (*f n*)))
and ¬ *is-axiom* (*s-of-ns* (*snd* (*f n*)))
shows *f* (*Suc n*) ∈ *deriv s* ∧ *fst* (*f* (*Suc n*)) = *Suc n* ∧ *snd* (*f* (*Suc n*)) ∈ *set* (*subs* (*snd* (*f n*))) ∧ *infinite* (*deriv* (*snd* (*f* (*Suc n*)))) ∧ ¬ *is-axiom* (*s-of-ns* (*snd* (*f* (*Suc n*))))
⟨*proof*⟩

lemma *is-path-f-0*: *f 0* = (*0*,*s*)
⟨*proof*⟩

lemma *is-path-f′*: *f n* ∈ *deriv s* ∧ *fst* (*f n*) = *n* ∧ *infinite* (*deriv* (*snd* (*f n*))) ∧ ¬ *is-axiom* (*s-of-ns* (*snd* (*f n*)))

⟨*proof*⟩

lemma *is-path-f*: $f\ n \in deriv\ s \wedge fst\ (f\ n) = n \wedge (snd\ (f\ (Suc\ n))) \in set\ (subs\ (snd\ (f\ n))) \wedge infinite\ (deriv\ (snd\ (f\ n)))$
⟨*proof*⟩

end

2.4 Models

typedecl *U*

type-synonym *model* = *U set* ∗ (*pred* ⇒ *U list* ⇒ *bool*)

type-synonym *env* = *var* ⇒ *U*

primrec *FEval* :: *model* ⇒ *env* ⇒ *form* ⇒ *bool*
where
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*)

lemma *preSuc*[*simp*]: *Suc n* ∈ *set A* = (*n* ∈ *set* (*preSuc A*))
⟨*proof*⟩

lemma *FEval-cong*: (∀ *x* ∈ *set* (*fv A*). *e1 x* = *e2 x*) ⟹ *FEval MI e1 A* = *FEval MI e2 A*
⟨*proof*⟩

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* = (∃ *f*. *f* ∈ *set s* ∧ *FEval m e f*)
⟨*proof*⟩

lemma *SEval-append*: *SEval m e* (*xs*@*ys*) ⟷ *SEval m e xs* ∨ *SEval m e ys*
⟨*proof*⟩

lemma *SEval-cong*: (∀ *x* ∈ *set* (*sfv s*). *e1 x* = *e2 x*) ⟹ *SEval m e1 s* = *SEval m e2 s*
⟨*proof*⟩

definition
is-env :: *model* ⇒ *env* ⇒ *bool*
where *is-env MI e* ≡ (∀ *x*. *e x* ∈ (*fst MI*))

definition
Svalid :: *form list* ⇒ *bool*
where *Svalid s* ≡ (∀ *MI e*. *is-env MI e* ⟶ *SEval MI e s*)

2.5 Soundness

lemma *FEval-subst*: (*FEval MI e* (*subst f A*)) = (*FEval MI* (*e* ∘ *f*) *A*)
⟨*proof*⟩

lemma *FEval-finst*: *FEval mo e* (*finst A u*) = *FEval mo* (*case-nat* (*e u*) *e*) *A*
⟨*proof*⟩

lemma *sound-FAll*:
assumes *u* ∉ *set* (*sfv* (*FAll f* # *s*))
and *Svalid* (*s* @ [*finst f u*])
shows *Svalid* (*FAll f* # *s*)
⟨*proof*⟩

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

lemma *inj-inc*: *inj inc*
⟨*proof*⟩

lemma *finite-inc*: *finite* (*inc* ‘ *X*) = *finite X*
⟨*proof*⟩

lemma *finite-deriv-deriv*: *finite* (*deriv s*) ⟹ *finite* (*deriv* ‘ {*w*. ¬ *is-axiom* (*s-of-ns s*) ∧ *w* ∈ *set* (*subs s*)})
⟨*proof*⟩

definition
init :: *nseq* ⇒ *bool* **where**
init s = (∀ *x* ∈ (*set s*). *fst x* = *0*)

definition
is-FEx :: *form* ⇒ *bool* **where**
is-FEx f = (*case f of*
PAtom p vs ⇒ *False*
| *NAtom p vs* ⇒ *False*
| *FConj f g* ⇒ *False*
| *FDisj f g* ⇒ *False*

| *FAll f* ⇒ *False*
| *FEx f* ⇒ *True*)

lemma *is-FEx*[*simp*]: ¬ *is-FEx* (*PAtom p vs*)
∧ ¬ *is-FEx* (*NAtom p vs*)
∧ ¬ *is-FEx* (*FConj f g*)
∧ ¬ *is-FEx* (*FDisj f g*)
∧ ¬ *is-FEx* (*FAll f*)
⟨*proof*⟩

lemma *index0*: ⟦*init s*; (*n*, *u*) ∈ *deriv s*; (*m*, *A*) ∈ *set u*; ¬ *is-FEx A*⟧ ⟹ *m* = *0*
⟨*proof*⟩

lemma *soundness′*:
assumes *init s* ⋀*y u*. (*y*, *u*) ∈ *deriv s* ⟹ *y* ≤ *m*
shows ⟦*h* = *m*−*n*; (*n*, *t*) ∈ *deriv s*⟧ ⟹ *Svalid* (*s-of-ns t*)
⟨*proof*⟩

lemma *s-of-ns-inverse*[*simp*]: *s-of-ns* (*ns-of-s s*) = *s*
⟨*proof*⟩

lemma *soundness*:
assumes *finite* (*deriv* (*ns-of-s s*)) **shows** *Svalid s*
⟨*proof*⟩

2.6 Contains, Considers

definition
contains :: (*nat* ⇒ (*nat*∗*nseq*)) ⇒ *nat* ⇒ *nform* ⇒ *bool* **where**
contains f n nf ≡ (*nf* ∈ *set* (*snd* (*f n*)))

definition
considers :: (*nat* ⇒ (*nat*∗*nseq*)) ⇒ *nat* ⇒ *nform* ⇒ *bool* **where**
considers f n nf ≡ (*case snd* (*f n*) *of* [] ⇒ *False* | (*x*#*xs*) ⇒ *x* = *nf*)

context *FailingPath*
begin

lemma *progress*:
assumes *snd* (*f n*) = *a* # *list*
shows ∃ *zs′*. *snd* (*f* (*Suc n*)) = *list* @ *zs′*
⟨*proof*⟩

lemma *contains-considers′*:
shows *snd* (*f n*) = *xs*@*y*#*ys* ⟹ ∃ *m zs′*. *snd* (*f* (*n*+*m*)) = *y*#*zs′*
⟨*proof*⟩

lemma *contains-considers*:
contains f n y ⟹ (∃ *m*. *considers f* (*n*+*m*) *y*)

⟨*proof*⟩

lemma *contains-propagates-patoms*:
contains f n (*0*, *PAtom p vs*) ⟹ *contains f* (*n*+*q*) (*0*, *PAtom p vs*)
⟨*proof*⟩

The same proof as above

lemma *contains-propagates-natoms*:
contains f n (*0*, *NAtom p vs*) ⟹ *contains f* (*n*+*q*) (*0*, *NAtom p vs*)
⟨*proof*⟩

lemma *contains-propagates-fconj*:
assumes *contains f n* (*0*, *FConj g h*)
shows ∃ *y*. *contains f* (*n* + *y*) (*0*, *g*) ∨ *contains f* (*n* + *y*) (*0*, *h*)
⟨*proof*⟩

lemma *contains-propagates-fdisj*:
assumes *contains f n* (*0*, *FDisj g h*)
shows ∃ *y*. *contains f* (*n* + *y*) (*0*, *g*) ∧ *contains f* (*n* + *y*) (*0*, *h*)
⟨*proof*⟩

lemma *contains-propagates-fall*:
assumes *contains f n* (*0*, *FAll g*)
shows ∃ *y*. *contains f* (*Suc*(*n*+*y*)) (*0*,*finst g* (*newvar* (*sfv* (*s-of-ns* (*snd* (*f* (*n*+*y*)))))))
⟨*proof*⟩

lemma *contains-propagates-fex*:
assumes *contains f n* (*m*, *FEx g*)
shows ∃ *y*. *contains f* (*n* + *y*) (*0*, *finst g m*) ∧ *contains f* (*n* + *y*) (*Suc m*, *FEx g*)
⟨*proof*⟩

lemma *FEx-downward*:
assumes *init s*
shows (*Suc m*, *FEx g*) ∈ *set* (*snd* (*f n*)) ⟹ (∃ *n*′. (*m*, *FEx g*) ∈ *set* (*snd* (*f n*′)))
⟨*proof*⟩

lemma *FEx0*:
assumes *init s*
shows *contains f n* (*m*, *FEx g*) ⟹ (∃ *n*′. *contains f n*′ (*0*, *FEx g*))
⟨*proof*⟩

lemma *FEx-upward*′:
assumes *contains f n* (*0*, *FEx g*)
shows ∃ *n*′. *contains f n*′ (*m*, *FEx g*)
⟨*proof*⟩

lemma *FEx-upward*:

assumes *init s*
and *contains f n* (*m*, *FEx g*)
shows ∃ *n′*. *contains f n′* (*0*, *finst g m′*)
⟨*proof*⟩

end

2.7 Models 2

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

definition *uton* :: *U* ⇒ *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* ∈ *range ntou* ⟹ *ntou* (*uton x*) = *x*
⟨*proof*⟩

2.8 Falsifying Model From Failing Path

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

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

lemma (**in** *FailingPath*) [*simp*]:
⟦*init s*; *contains f n* (*m*,*A*); ¬ *is-FEx A*⟧ ⟹ *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 ¬ *FEval* (*model s*) *ntou A*
⟨*proof*⟩

2.9 Completeness

lemma *completeness′*:
assumes *infinite* (*deriv s*) *init s* (*m*,*A*) ∈ *set s*
shows ¬ *FEval* (*model s*) *ntou A*
⟨*proof*⟩

lemma *completeness*:
  assumes *infinite* (*deriv* (*ns-of-s s*))
  shows ¬ *Svalid s*
⟨*proof*⟩

2.10 Sound and Complete

lemma *Svalid s* = *finite* (*deriv* (*ns-of-s s*))
  ⟨*proof*⟩

2.11 Algorithm

lemma *ex-iter'*: (∃ *n*. *R* ((*g*^^*n*)*a*)) = (*R a* ∨ (∃ *n*. *R* ((*g*^^*n*)(*g a*))))
  ⟨*proof*⟩
lemma *ex-iter*: (∃ *n*. *R* ((*g*^^*n*)*a*)) = (*if R a then True else* (∃ *n*. *R* ((*g*^^*n*)(*g a*))))
  ⟨*proof*⟩

definition
  f :: *nseq list* ⇒ *nat* ⇒ *nseq list* **where**
  f s n ≡ ((λ *x*. *concat* (*map subs x*))^^*n*) *s*

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

lemma *f*: ((*n*,*x*) ∈ *deriv s*) = (*x* ∈ *set* (*f* [*s*] *n*))
⟨*proof*⟩

lemma *deriv-f*: *deriv s* = (⋃ *x*. *set* (*map* (*Pair x*) (*f* [*s*] *x*)))
  ⟨*proof*⟩

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

definition *prove'* :: *nseq list* ⇒ *bool* **where**
  prove' s = (∃ *m*. ((λ *x*. *concat* (*map subs x*))^^*m*) *s* = [])

lemma *prove'*: *prove' l* = (*if l* = [] *then True else prove'* ((λ *x*. *concat* (*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

lemma (∃ *x*. *A x* ∨ *B x*) ⟶ ((∃ *x*. *B x*) ∨ (∃ *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*:
$x \in$ *set* [] $\longleftrightarrow$ *False*
$x \in$ *set* (y # *ys*) $\longleftrightarrow x = y \vee x \in$ *set ys*
⟨*proof*⟩

lemmas *ss* = *list.inject if-True if-False concat.simps list.map*
sfv-def filter.simps snd-conv form.simps finst-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* ⇒ *bool* **where** *Sprove* ≡ *prove* ∘ *ns-of-s*

abbreviation *check* :: *form* ⇒ *bool* **where** *check formula* ≡ *Sprove* [*formula*]

abbreviation *valid* :: *form* ⇒ *bool* **where** *valid formula* ≡ *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,PA
        | NAtom (p,vs) -> if mem (PAtom (p,vs)) (map snd xs) then [] else [xs@[(0,NA
        | 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.