

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

Tom Ridge

February 6, 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 | 7 |
| 2.4 | Models | 8 |
| 2.5 | Soundness | 9 |
| 2.6 | Contains, Considers | 15 |
| 2.7 | Models 2 | 19 |
| 2.8 | Falsifying Model From Failing Path | 20 |
| 2.9 | Completeness | 21 |
| 2.10 | Sound and Complete | 22 |
| 2.11 | Algorithm | 22 |
| 2.12 | Computation | 24 |
| 3 | Optimisation and Extension | 26 |
| 4 | OCaml Implementation | 26 |

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 $size-subst[simp]$: $\forall m. size\ (subst\ m\ f) = size\ f$
by $(induct\ f)\ (force+)$

definition

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

lemma $size-finst[simp]$: $size\ (finst\ f\ m) = size\ f$
by $(simp\ add:\ finst-def)$

type-synonym $seq = form\ list$

type-synonym $nform = nat * form$

type-synonym $nseq = nform\ list$

definition

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

definition

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

definition

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

lemma $sfv-nil$: $sfv\ [] = []$
by $(force\ simp:\ sfv-def)$

lemma $sfv-cons$: $sfv\ (a\#\ list) = (fv\ a)\ @\ (sfv\ list)$
by $(force\ simp:\ sfv-def)$

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

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

lemma *maxvar*: $\forall v \in set\ vs.\ v \leq maxvar\ vs$
by (*induct vs*) (*auto simp: max-def*)

definition

$newvar :: var\ list \Rightarrow var$ **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*: $newvar\ vs \notin (set\ vs)$
using *length-pos-if-in-set maxvar newvar-def* **by** *force*

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

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

2.2 Derivations

primrec *is-axiom* :: $seq \Rightarrow bool$

where

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

inductive-set

$deriv :: nseq \Rightarrow (nat * nseq)\ set$

for $s :: nseq$

where

$init: (0,s) \in deriv\ s$
 $| step: (n,x) \in deriv\ s \Longrightarrow y \in set\ (subs\ x) \Longrightarrow (Suc\ n,y) \in deriv\ s$
— 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*: $(n, (m, FEx\ f)\#xs) \in deriv(nfs) \implies \neg is-axiom\ (s-of-ns\ ((m, FEx\ f)\#xs)) \implies (Suc\ n, xs@[0, finst\ f\ m], (Suc\ m, FEx\ f]) \in deriv(nfs)$
by (*auto simp: s-of-ns-def*)

lemma *deriv0*[*simp*]: $(0, x) \in deriv\ y \longleftrightarrow (x = y)$
using *deriv.cases by blast*

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 (*cases x*)
case (*Cons ab list*)
show *?thesis*
proof (*cases ab*)
case (*Pair a b*)
with *Cons assms show ?thesis*
by (*cases b; fastforce simp: s-of-ns-def*)
qed
qed (*use assms in auto*)

lemma *deriv-upwards*: $(n, list) \in deriv\ s \implies \neg is-axiom\ (s-of-ns\ (list)) \implies (\exists zs. (Suc\ n, zs) \in deriv\ s \wedge zs \in set\ (subs\ list))$
by (*metis deriv.step deriv-exists list.set-intros(1) subs.simps(1)*)

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 –
obtain *x'* **where** $x': (n, x') \in deriv\ s\ x \in set\ (subs\ x')$
using *Suc-derivE assms by blast*
have *False* **if** *is-axiom (s-of-ns x')*
proof (*cases x'*)
case (*Cons ab list*)

```

show ?thesis
proof (cases ab)
  case (Pair a b)
    with Cons x' that assms show ?thesis
    by (cases b) (auto simp: s-of-ns-def)
  qed
next
  case Nil
  then show ?thesis
    using s-of-ns-def that by auto
  qed
then show ?thesis
  using x' by blast
qed

```

```

lemma deriv-deriv-child: (Suc n,x) ∈ deriv y = (∃ z. z ∈ set (subs y) ∧ ¬ is-axiom
(s-of-ns y) ∧ (n,x) ∈ deriv z)
proof (induction n arbitrary: x y)
  case 0
  then show ?case
    using deriv-downwards by (force elim: Suc-derivE)
next
  case (Suc n)
  then show ?case
    by (meson deriv-downwards deriv.step)
qed

```

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

```

lemma deriv-progress:
  assumes (n, a # list) ∈ deriv s
  and ¬ is-axiom (s-of-ns (a # list))
  shows ∃ zs. (Suc n, list @ zs) ∈ deriv s
  by (metis assms form.exhaust surj-pair not-is-axiom-subs)

```

```

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)})))
  apply (auto simp add: inc-def deriv-deriv-child image-iff split-def)
  apply (metis deriv.cases deriv-deriv-child)
  apply (metis deriv-deriv-child fst-conv old.nat.exhaust snd-conv)
  apply (metis deriv.cases deriv-deriv-child)
  apply (metis deriv.cases deriv-deriv-child fst-conv snd-conv)
  done

```

```

lemma deriv-is-axiom: is-axiom (s-of-ns s) ⇒ deriv s = {(0,s)}

```

using *deriv* by *blast*

lemma *is-axiom-finite-deriv*: *is-axiom (s-of-ns s) \implies finite (deriv s)*
by (*simp add: deriv-is-axiom*)

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)))
by (*metis (mono-tags, lifting) inf deriv0 f failing-path.simps(1) fst-conv is-axiom-finite-deriv snd-conv someI-ex*)

lemma *fSuc*:

assumes *fn*: *f n \in deriv s 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))))*

proof –

have *infinite (UN (deriv ‘ {w. \neg is-axiom (s-of-ns (snd (f n))) \wedge w \in set (subs (snd (f n)))}))*

by (*metis inf deriv finite-imageI finite-insert*)

then obtain *y* **where** *y \in set (subs (snd (f n))) infinite (deriv y)*

by *fastforce*

then have $\exists x. x \in deriv s \wedge fst x = Suc n $\wedge$$

$snd x \in set (subs (snd (failing-path (deriv s) n))) $\wedge$$

$infinite (deriv (snd x)) \wedge \neg is-axiom (s-of-ns (snd x))$

by (*metis deriv.step f fn fst-conv is-axiom-finite-deriv prod.exhaust-sel snd-conv*)

then show *?thesis*

by (*metis (mono-tags, lifting) f failing-path.simps(2) someI-ex*)

qed

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

by (*metis deriv0 f0 split-pairs*)

lemma *is-path-f'*: $f\ n \in \text{deriv } s \wedge \text{fst } (f\ n) = n \wedge \text{infinite } (\text{deriv } (\text{snd } (f\ n))) \wedge \neg \text{is-axiom } (s\text{-of-ns } (\text{snd } (f\ n)))$
by (*induction n*) (*auto simp: f0 fSuc*)

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

end

2.4 Models

typedecl *U*

type-synonym *model* = *U set * (pred \Rightarrow U list \Rightarrow bool)*

type-synonym *env* = *var \Rightarrow U*

primrec *FEval* :: *model \Rightarrow env \Rightarrow form \Rightarrow 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 \neg (IP (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) = ($\forall m \in (\text{fst } MI). \text{FEval } MI (\lambda y. \text{case } y \text{ of } 0 \Rightarrow m \mid \text{Suc } n \Rightarrow e\ n) f$)
| FEval MI e (FEx f) = ($\exists m \in (\text{fst } MI). \text{FEval } MI (\lambda y. \text{case } y \text{ of } 0 \Rightarrow m \mid \text{Suc } n \Rightarrow e\ n) f$)

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

by (*induction A*) (*auto simp: split: nat.splits*)

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

proof (*induction A arbitrary: e1 e2*)

case (*PAtom x1 x2*)

then show *?case*

by (*metis FEval.simps(1) fv.simps(1) map-cong*)

next

case (*NAtom x1 x2*)

then show *?case*

by *simp (metis list.map-cong0)*

next

case (*FConj A1 A2*)

then show *?case*

by *simp blast*

next

```

    case (FDisj A1 A2)
  then show ?case
    by simp blast
next
  case (FAll A)
  then show ?case
    by (metis (no-types, lifting) FEval.simps(5) Nitpick.case-nat-unfold One-nat-def
        Suc-pred fv.simps(5) grOI preSuc)
next
  case (FEx A)
  then show ?case
    by (metis (no-types, lifting) FEval.simps(6) Nitpick.case-nat-unfold One-nat-def
        Suc-pred fv.simps(6) grOI preSuc)
qed

```

```

primrec SEval :: model  $\Rightarrow$  env  $\Rightarrow$  form list  $\Rightarrow$  bool
where
  SEval m e [] = False
| SEval m e (x#xs) = (FEval m e x  $\vee$  SEval m e xs)

```

```

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

```

```

lemma SEval-append: SEval m e (xs@ys)  $\longleftrightarrow$  SEval m e xs  $\vee$  SEval m e ys
  by (induct xs) auto

```

```

lemma SEval-cong: ( $\forall x \in \text{set } (sfv s). e1 x = e2 x$ )  $\implies$  SEval m e1 s = SEval m
e2 s
proof (induction s)
  case Nil
  then show ?case by auto
next
  case (Cons a s)
  then show ?case
    by (metis SEval.simps(2) FEval-cong Un-iff sfv-cons set-append)
qed

```

```

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

```

```

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

```

2.5 Soundness

```

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

```

```

proof –
  have §: (case bump f k of 0 ⇒ m | Suc x ⇒ e x) =
    (case k of 0 ⇒ m | Suc n ⇒ e (f n))
    if m ∈ fst MI for m k e f
    using that by (simp add: bump-def split: nat.splits)
  show ?thesis
  proof (induction A arbitrary: e f)
    case (FAll A)
    with § show ?case by simp
  next
    case (FEx A)
    with § show ?case by simp
  qed (use FEval.simps in auto)
qed

```

```

lemma FEval-finst: FEval mo e (finst A u) = FEval mo (case-nat (e u) e) A
proof –
  have (e ∘ case-nat u (λn. n)) = (case-nat (e u) e)
    by (simp add: fun-eq-iff split: nat.splits)
  then show ?thesis
    by (simp add: FEval-subst finst-def)
qed

```

```

lemma sound-FAll:
  assumes u ∉ set (sfv (FAll f # s))
    and Svalid (s @ [finst f u])
  shows Svalid (FAll f # s)
proof –
  have SEval (M, I) e (FAll f # s)
    if e: is-env (M, I) e for M I e
  proof –
    consider SEval (M, I) e s | FEval (M, I) e (finst f u) ¬ SEval (M, I) e s
    using SEval-append Svalid-def assms e by fastforce
    then show ?thesis
  proof cases
    case 1
    then show ?thesis
      by auto
  next
    case 2
    have FEval (M, I) (case-nat m e) f
      if m ∈ M for m
    proof –
      have FEval (M, I) (case-nat ((e(u := m)) u) (e(u := m))) f (is ?P)
        using assms e ⟨m ∈ M⟩ 2
      apply (simp add: Svalid-def SEval-append is-env-def FEval-finst sfv-cons)
      by (smt (verit, best) SEval-cong fun-upd-apply)
      moreover have ?P = FEval (M, I) (case-nat m e) f

```

```

      using assms
      by (intro FEval-cong strip) (auto simp: sfv-cons split: nat.splits)
    ultimately show ?thesis
      by auto
  qed
  then show ?thesis
    by (auto simp: SEval-append 2)
  qed
  qed
  then show ?thesis
    by (simp add: Svalid-def)
  qed

```

```

lemma sound-FEx: Svalid (s@[fnst f u, FEx f])  $\implies$  Svalid (FEx f#s)
  unfolding Svalid-def
  by (metis FEval.simps(6) FEval-fnst SEval.simps SEval-append is-env-def)

```

```

lemma inj-inc: inj inc
  by (simp add: Prover.inc-def inj-def)

```

```

lemma finite-inc: finite (inc ‘ X) = finite X
  by (metis finite-imageD finite-imageI inj-def inj-inc inj-on-def)

```

```

lemma finite-deriv-deriv: finite (deriv s)  $\implies$  finite (deriv ‘ {w.  $\neg$  is-axiom (s-of-ns s)  $\wedge$  w  $\in$  set (subs s)})
  by simp

```

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
  | PAtom p vs  $\Rightarrow$  False
  | NAtom p vs  $\Rightarrow$  False
  | FConj f g  $\Rightarrow$  False
  | 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)
  by (force simp: is-FEx-def)

```

```

lemma index0:  $\llbracket \text{init } s; (n, u) \in \text{deriv } s; (m, A) \in \text{set } u; \neg \text{is-FEx } A \rrbracket \implies m = 0$ 
proof (induction n arbitrary: u)
  case 0
  then show ?case
    using init-def by auto
next
  case (Suc n)
  then obtain y where  $y: (n, y) \in \text{deriv } s \ u \in \text{set } (\text{subs } y) \neg \text{is-axiom } (s\text{-of-ns } y)$ 
    using deriv-downwards by blast
  have ?case if  $y = \text{Cons } (a,b) \text{ list}$  for a b list
    using that y Cons Suc
  by (fastforce simp: is-FEx-def split: form.splits if-splits)
  then show ?case
    using Suc y
  by (metis empty-iff list.set(1) neq-Nil-conv set-ConsD subs.simps(1) surjective-pairing)
qed

lemma soundness':
  assumes  $\text{init } s \wedge y \ u. (y, u) \in \text{deriv } s \implies y \leq m$ 
  shows  $\llbracket h = m - n; (n, t) \in \text{deriv } s \rrbracket \implies \text{Svalid } (s\text{-of-ns } t)$ 
proof (induction h arbitrary: n t)
  case 0
  show ?case
  proof (cases m=n)
    case True
    show ?thesis
  proof (cases is-axiom (s-of-ns t))
    case True
    then have *: is-axiom (map snd t)
      using s-of-ns-def by force
    have ?thesis if  $t = \text{Cons } u \ v$  for u v
      using * that 0
      by (simp add: Svalid-def SEval-def2 s-of-ns-def (metis FEval.simps(1,2)))
    then show ?thesis
      by (metis True is-axiom.simps(1) list.exhaust list.simps(8) s-of-ns-def)
  next
  case False
  with 0.prem1 True assms show ?thesis
    by (metis deriv-upwards not-less-eq-eq)
  qed
next
  case False
  with 0.prem1 assms show ?thesis by force
qed
next
  case (Suc h)
  show ?case
  proof (cases is-axiom (s-of-ns t))

```

```

case True
have SEval (M, I) e (map snd ((u, v) # list))
  if t = (u, v) # list is-env (M, I) e for u v list M I e
  using that SEval-def2 True s-of-ns-def by fastforce
with True show ?thesis
  unfolding Svalid-def s-of-ns-def
  by (metis Nil-is-map-conv is-axiom.simps(1) list.exhaust surjective-pairing)
next
case False
show ?thesis
proof (cases t)
  case Nil
  with Suc assms show ?thesis
  apply simp
  by (metis Suc-leD deriv.step diff-Suc-Suc diff-diff-cancel diff-le-self
    list.set-intros(1) subs.simps(1))
next
case (Cons u v)
have ?thesis if u = (M, fm) for M fm
  using that
proof (induction fm)
  case (PAtom p vs)
  then have (Suc n, v @ [(0, PAtom p vs)]) ∈ deriv s
    using False Suc.prem local.Cons patom by blast
  with PAtom show ?case
    using Suc.IH [of Suc n v @ [(0, PAtom p vs)]] Suc.prem
    by (fastforce simp: Svalid-def SEval-append Cons s-of-ns-def)
next
case (NAtom p vs)
  then have (Suc n, v @ [(0, NAtom p vs)]) ∈ deriv s
    using False Suc.prem local.Cons natom by blast
  with NAtom show ?case
    using Suc.IH [of Suc n v @ [(0, NAtom p vs)]] Suc.prem
    by (fastforce simp: Svalid-def SEval-append Cons s-of-ns-def)
next
case (FConj fm1 fm2)
  then obtain (Suc n, v @ [(0, fm1)]) ∈ deriv s (Suc n, v @ [(0, fm2)]) ∈
deriv s
    using Suc.prem local.Cons by force
  with FConj show ?case
    using Suc.IH [of Suc n v @ [(0, fm1)]] Suc.prem
    using Suc.IH [of Suc n v @ [(0, fm2)]] assms
    apply (simp add: Cons s-of-ns-def Svalid-def SEval-append)
    by (metis Suc-diff-le diff-Suc-1' diff-Suc-Suc)
next
case (FDisj fm1 fm2)
  then have (Suc n, v @ [(0, fm1), (0, fm2)]) ∈ deriv s
    using Suc.prem local.Cons by force
  with FDisj show ?case

```

```

    using Suc.IH [of Suc n v @ [(0, fm1),(0, fm2)]] Suc.premss assms
    apply (simp add: Cons s-of-ns-def Svalid-def SEval-append)
    by (metis Suc-diff-le diff-Suc-1' diff-Suc-Suc)
  next
  case (Fall fm)
  then have M=0
    using Suc.premss index0 Cons assms by force
  have newvar (sfv (s-of-ns t)) ∉ set (sfv (s-of-ns t))
    by (simp add: newvar)
  with Fall ⟨M=0⟩ show ?case
    using Suc.IH [of Suc n v @ [(0, finst fm (newvar (sfv (s-of-ns t))))]]
Suc.premss
    by (force simp: Cons s-of-ns-def fall sound-Fall)
  next
  case (FEx fm)
  then have (Suc n, v @ [(0, finst fm M), (Suc M, FEx fm)]) ∈ deriv s
    using Suc.premss local.Cons by auto
  with FEx Suc have Svalid (s-of-ns (v@[0, finst fm M], (Suc M, FEx fm)))
    by (metis diff-Suc nat.case(2))
  with FEx show ?case
    by (simp add: local.Cons s-of-ns-def sound-FEx)
  qed
  then show ?thesis
    using surjective-pairing by blast
  qed
  qed
  qed
lemma s-of-ns-inverse[simp]: s-of-ns (ns-of-s s) = s
  by (induct s) (simp-all add: s-of-ns-def ns-of-s-def)

lemma soundness:
  assumes finite (deriv (ns-of-s s)) shows Svalid s
proof -
  obtain x where x: x ∈ fst `deriv (ns-of-s s)
    ∧ y. y ∈ fst `deriv (ns-of-s s) ⇒ y ≤ x
  by (metis assms deriv.init empty-iff finite-imageI image-eqI eq-Max-iff)
  have Svalid (s-of-ns (ns-of-s s))
  proof (intro soundness')
    show init (ns-of-s s)
    by (simp add: init-def ns-of-s-def)
  next
  fix y u
  assume (y, u) ∈ deriv (ns-of-s s)
  with x show y ≤ x
    by fastforce
  qed (use assms x in force)+
  then show ?thesis
    by auto

```

qed

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 $\exists\ zs'.\ snd\ (f\ (Suc\ n)) = list\ @\ zs'$

proof –

have $(snd\ (f\ (Suc\ n))) \in set\ (subs\ (snd\ (f\ n)))$

using *is-path-f* **by** *blast*

then have *?thesis* **if** $a = (M,I)$ **for** $M\ I$

using *assms that*

by (*cases I*) (*auto simp: split: if-splits*)

then show *?thesis*

by *fastforce*

qed

lemma *contains-considers'*:

shows $snd\ (f\ n) = xs@y\#\#ys \implies \exists\ m\ zs'.\ snd\ (f\ (n+m)) = y\#\#zs'$

proof (*induction xs arbitrary: n ys*)

case *Nil*

then show *?case*

by (*metis add.right-neutral append-Nil*)

next

case (*Cons MI v*)

then obtain zs' **where** $snd\ (f\ (Suc\ n)) = (v\ @\ y\ \#\# ys)\ @\ zs'$

using *progress Cons.prem*

by (*metis append-Cons*)

then show *?case*

by (*metis Cons.IH add-Suc-shift append-Cons append-assoc*)

qed

lemma *contains-considers*:

$contains\ f\ n\ y \implies (\exists\ m.\ considers\ f\ (n+m)\ y)$

unfolding *contains-def considers-def*

by (*smt (verit, ccfv-threshold) list.simps(5) FailingPath.contains-considers' FailingPath-axioms*

split-list-first)

```

lemma contains-propagates-patoms:
  contains f n (0, PAtom p vs)  $\implies$  contains f (n+q) (0, PAtom p vs)
proof(induction q)
  case 0
  then show ?case
    by auto
next
  case (Suc q)
  then have §:  $\neg$  is-axiom (s-of-ns (snd (f (n+q))))
    using is-path-f' by blast
  then have infinite (deriv (snd (f (n+q))))
    by (simp add: Suc.premis(1) is-path-f')
  obtain xs ys where *: snd (f (n + q)) = xs @ (0, PAtom p vs) # ys
    (0, PAtom p vs)  $\notin$  set xs
    by (meson Prover.contains-def Suc split-list-first)
  have (0, PAtom p vs)  $\in$  set (snd (f (Suc (n + q))))
proof (cases xs)
  case Nil
  then have (snd (f (Suc (n + q))))  $\in$  set (subs (snd (f (n + q))))
    using Suc.premis(1) is-path-f by blast
  with * Nil show ?thesis
    by (simp split: if-splits)
next
  case (Cons a list)
  with Suc show ?thesis
    by (smt (verit, best) * progress append-Cons append-assoc in-set-conv-decomp)
qed
then show ?case
  by (simp add: contains-def)
qed

```

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)
proof(induction q)
  case 0
  then show ?case
    by auto
next
  case (Suc q)
  then have §:  $\neg$  is-axiom (s-of-ns (snd (f (n+q))))
    using is-path-f' by blast
  then have infinite (deriv (snd (f (n+q))))
    by (simp add: Suc.premis(1) is-path-f')
  obtain xs ys where *: snd (f (n + q)) = xs @ (0, NAtom p vs) # ys
    (0, NAtom p vs)  $\notin$  set xs
    by (meson Prover.contains-def Suc split-list-first)

```

```

have (0, NAtom p vs) ∈ set (snd (f (Suc (n + q))))
proof (cases xs)
  case Nil
  then have (snd (f (Suc (n + q)))) ∈ set (subs (snd (f (n + q))))
    using Suc.prem1 is-path-f by blast
  with * Nil show ?thesis
    by (simp split: if-splits)
next
  case (Cons a list)
  with Suc show ?thesis
    by (smt (verit, best) * progress append-Cons append-assoc in-set-conv-decomp)
qed
then show ?case
  by (simp add: contains-def)
qed

```

```

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 -
  obtain l where l: considers f (n+l) (0, FConj g h)
    using assms contains-considers by blast
  then have *: (snd (f (Suc (n + l)))) ∈ set (subs (snd (f (n + l))))
    using assms(1) is-path-f by blast
  have contains f (n + (Suc l)) (0, g) ∨ contains f (n + (Suc l)) (0, h)
  proof (cases snd (f (n + l)))
    case Nil
    then show ?thesis
      using considers-def l by auto
  next
    case (Cons a list)
    then show ?thesis
      using l * by (auto simp: contains-def considers-def in-set-conv-decomp)
  qed
  then show ?thesis ..
qed

```

```

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 -
  obtain l where l: considers f (n+l) (0, FDisj g h)
    using assms contains-considers by blast
  then obtain a list where *: snd (f (n + l)) = a # list
    by (metis considers-def list.simps(4) neq-Nil-conv)
  have **: snd (f (Suc (n + l))) ∈ set (subs (snd (f (n + l))))
    using assms is-path-f by blast
  show ?thesis
  proof (intro exI conjI)

```

```

  show contains f (n + (Suc l)) (0, g) contains f (n + (Suc l)) (0, h)
  using l *** assms by (auto simp: contains-def considers-def in-set-conv-decomp)
qed

```

```

lemma contains-propagates-fall:
  assumes contains f n (0, FAll g)
  shows  $\exists y. \text{contains } f \text{ (Suc(n+y)) (0, first } g \text{ (newvar (sfv (s-of-ns (snd (f (n+y)))))))))$ 
proof -
  obtain l where l: considers f (n+l) (0, FAll g)
  using assms contains-considers by blast
  then obtain a list where *: snd (f (n + l)) = a # list
  by (metis considers-def list.simps(4) neq-Nil-conv)
  have **: snd (f (Suc (n + l)))  $\in$  set (subs (snd (f (n + l))))
  using assms is-path-f by blast
  show ?thesis
  proof (intro exI conjI)
    show contains f (Suc (n+l)) (0, first g (newvar (sfv (s-of-ns (snd (f (n + l)))))))))
    using l *** assms by (auto simp: contains-def considers-def in-set-conv-decomp)
  qed
qed

```

```

lemma contains-propagates-fex:
  assumes contains f n (m, FEx g)
  shows  $\exists y. \text{contains } f \text{ (n + y) (0, first } g \text{ m) } \wedge \text{contains } f \text{ (n + y) (Suc m, FEx g)}$ 
proof -
  obtain l where l: considers f (n+l) (m, FEx g)
  using assms contains-considers by blast
  then obtain a list where *: snd (f (n + l)) = a # list
  by (metis considers-def list.simps(4) neq-Nil-conv)
  have **: snd (f (Suc (n + l)))  $\in$  set (subs (snd (f (n + l))))
  using assms is-path-f by blast
  show ?thesis
  proof (intro exI conjI)
    show contains f (n + (Suc l)) (0, first g m)
      contains f (n + (Suc l)) (Suc m, FEx g)
    using l *** by (auto simp: contains-def considers-def in-set-conv-decomp)
  qed
qed

```

— also need that if contains one, then contained an original at beginning
 — existentials: show that for exists formulae, if Suc m is marker, then there must have been m
 — show this by showing that nodes are upwardly closed, i.e. if never contains (m,x), then never contains (Suc m, x), by induction on n

lemma *FEx-downward*:

```

assumes init s
shows  $(\text{Suc } m, \text{FEx } g) \in \text{set } (\text{snd } (f \ n)) \implies (\exists n'. (m, \text{FEx } g) \in \text{set } (\text{snd } (f \ n')))$ 
proof (induction n arbitrary: m)
  case 0
    with inf init-def is-path-f-0 <init s>
    show ?case by auto
  next
    case (Suc n)
    note § = Suc assms is-path-f [of n]
    have ?case if  $f \ n = (n, \text{Cons } (a, fm) \ \text{list})$  for a fm list
    proof (cases fm)
      case (FEx x6)
        with § that show ?thesis
        by simp (metis list.set-intros(1) snd-conv)
    qed (use § that in <auto split: if-splits>)
    then show ?case
      by (metis Suc.premis empty-iff is-path-f list.exhaust list.set(1) set-ConsD
        subs.simps(1) split-pairs)
  qed

```

```

lemma FEx0:
  assumes init s
  shows  $\text{contains } f \ n \ (m, \text{FEx } g) \implies (\exists n'. \text{contains } f \ n' \ (0, \text{FEx } g))$ 
  using assms
  by (induction m arbitrary: n) (auto simp: contains-def dest: FEx-downward)

```

```

lemma FEx-upward':
  assumes  $\text{contains } f \ n \ (0, \text{FEx } g)$ 
  shows  $\exists n'. \text{contains } f \ n' \ (m, \text{FEx } g)$ 
  by (induction m; use assms contains-propagates-fex in blast)

```

— FIXME contains and considers aren't buying us much

```

lemma FEx-upward:
  assumes init s
  and  $\text{contains } f \ n \ (m, \text{FEx } g)$ 
  shows  $\exists n'. \text{contains } f \ n' \ (0, \text{fst } g \ m')$ 
proof –
  obtain n' where  $\text{contains } f \ n' \ (m', \text{FEx } g)$ 
  using FEx0 FEx-upward' assms by blast
  then show ?thesis
  using contains-propagates-fex by blast
qed

```

end

2.7 Models 2

```

axiomatization ntou ::  $\text{nat} \Rightarrow U$ 

```

where $ntou$: $inj\ ntou$ — assume universe set is infinite

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

lemma $uton-ntou$: $uton\ (ntou\ x) = x$
by ($simp\ add$: $inv-f-f\ ntou\ uton-def$)

lemma $map-uton-ntou[simp]$: $map\ uton\ (map\ ntou\ xs) = xs$
by ($induct\ xs$, $auto\ simp$: $uton-ntou$)

lemma $ntou-uton$: $x \in range\ ntou \implies ntou\ (uton\ x) = x$
by ($simp\ add$: $f-inv-into-f\ uton-def$)

2.8 Falsifying Model From Failing Path

definition $model :: nseq \Rightarrow model$ **where**

$model\ s \equiv$
 $(range\ ntou,$
 $\lambda\ p\ ms.\ let\ f = failing-path\ (deriv\ s)\ in$
 $\forall\ n\ m.\ \neg\ contains\ f\ n\ (m, PAtom\ p\ (map\ uton\ ms)))$

lemma $is-env-model-ntou$: $is-env\ (model\ s)\ ntou$
by ($simp\ add$: $is-env-def\ model-def$)

lemma (**in** $FailingPath$) [$simp$]:
 $\llbracket init\ s; contains\ f\ n\ (m, A); \neg\ is-FEx\ A \rrbracket \implies m = 0$
by ($metis\ Prover.contains-def\ index0\ is-path-f'\ surjective-pairing$)

lemma (**in** $FailingPath$) $model'$:

assumes $init\ s$
and A : $h = size\ A\ contains\ f\ n\ (m, A)\ FEval\ (model\ s)\ ntou\ A$
shows $\neg\ FEval\ (model\ s)\ ntou\ A$
using A

proof ($induction\ h\ arbitrary$: $A\ m\ n$ $rule$: $less-induct$)

case ($less\ x\ A\ m\ n$)

show $?case$

proof ($cases\ A$)

case ($PAtom\ p\ vs$)

then show $?thesis$

using $f\ less.premis(2)\ map-uton-ntou\ model-def$ **by** $auto$

next

case ($NAtom\ p\ vs$)

with $less.premis$ **obtain** $nN\ mN\ nP\ mP$

where \S : $contains\ f\ nN\ (mN, NAtom\ p\ vs)\ contains\ f\ nP\ (mP, PAtom\ p\ vs)$

using $f\ map-uton-ntou\ model-def$ **by** $auto$

then have $mN=0\ mP=0$

by ($auto\ simp$: $inf\ \langle init\ s \rangle$)

then obtain d **where** d : $considers\ f\ (nN+nP+d)\ (0, PAtom\ p\ vs)$

```

    by (metis §(2) add.commute contains-considers contains-propagates-patoms)
  then have is-axiom (s-of-ns (snd (f (nN+nP+d))))
    using contains-propagates-natoms § ⟨mN = 0⟩ assms
    apply (simp add: s-of-ns-def considers-def image-iff split: list.splits)
    by (metis contains-def form.distinct(1) set-ConsD snd-conv)
  then show ?thesis
    by (simp add: inf is-path-f')
next
case (FConj fm1 fm2)
with less.prem1 inf ⟨init s⟩ have m=0
  by auto
then obtain d where contains f (n+d) (0, fm1) ∨ contains f (n+d) (0, fm2)
  using FConj inf contains-propagates-fconj less.prem2 by blast
with FConj show ?thesis
  using less.IH less.prem1 by force
next
case (FDisj fm1 fm2)
with less.prem1 inf ⟨init s⟩ have m=0
  by auto
then obtain d where contains f (n+d) (0, fm1) ∧ contains f (n+d) (0, fm2)
  using FDisj inf contains-propagates-fdisj less.prem2 by blast
with FDisj show ?thesis
  using less.IH less.prem1 by force
next
case (FAll fm)
with less.prem1 inf ⟨init s⟩ have m=0
  by auto
then obtain d where
  contains f (Suc (n+d)) (0, first fm (newvar (sfv (s-of-ns (snd (f (n+d)))))))
  using FAll inf contains-propagates-fall less.prem2 by blast
with FAll less have ¬ FEval (model s) ntou (first fm (newvar (sfv (s-of-ns
(snd (f (n+d)))))))
  by (metis add-diff-cancel-left' form.size(11) lessI size-first zero-less-diff)
with FAll show ?thesis
  using FEval-first is-env-def is-env-model-ntou by auto
next
case (FEx fm)
then have ∀ m'. ∃ n'. contains f n' (0, first fm m')
  using FEx-upward assms less.prem1 by blast
with FEx less have ∀ m'. ¬ FEval (model s) ntou (first fm m')
  by (metis add.comm-neutral add-Suc-right form.size(12) lessI size-first)
then show ?thesis
  by (simp add: FEval-first FEx model-def)
qed
qed

```

2.9 Completeness

lemma completeness':

assumes *infinite* (*deriv s*) *init s* ($m, A \in \text{set } s$)
shows $\neg \text{FEval} (\text{model } s) \text{ntou } A$
by (*metis contains-def* *assms FailingPath.intro FailingPath.is-path-f-0 FailingPath.model' snd-conv*)

lemma *completeness*:

assumes *infinite* (*deriv (ns-of-s s)*)
shows $\neg \text{Svalid } s$
proof –
have *init (ns-of-s s)*
by(*simp add: init-def ns-of-s-def*)
with *assms* **have** $\bigwedge A. A \in \text{set } s \implies \neg \text{FEval} (\text{model} (\text{ns-of-s } s)) \text{ntou } A$
unfolding *ns-of-s-def* **using** *completeness'* **by** *fastforce*
with *assms* **show** *?thesis*
using *SEval-def2 Svalid-def is-env-model-ntou* **by** *blast*
qed

2.10 Sound and Complete

lemma *Svalid s = finite (deriv (ns-of-s s))*
using *soundness completeness* **by** *blast*

2.11 Algorithm

lemma *ex-iter'*: $(\exists n. R ((g \sim^n) a)) = (R a \vee (\exists n. R ((g \sim^n)(g a))))$
by (*metis (mono-tags, lifting) funpow-0 funpow-Suc-right not0-implies-Suc o-apply*)

— version suitable for computation

lemma *ex-iter*: $(\exists n. R ((g \sim^n) a)) = (\text{if } R a \text{ then True else } (\exists n. R ((g \sim^n)(g a))))$
by (*metis ex-iter'*)

definition

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

lemma *f-upwards*: $f s n = [] \implies f s (n+m) = []$
by (*induction m*) (*auto simp: f-def*)

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

unfolding *f-def*

proof (*induction n arbitrary: x*)

case *0*

then show *?case*

by *auto*

next

case (*Suc n*)

then show *?case*

by (*auto simp: deriv.simps[of Suc n]*)

qed

lemma *deriv-f*: $\text{deriv } s = (\bigcup x. \text{set } (\text{map } (\text{Pair } x) (f [s] x)))$
using *f* **by** (*auto simp: set-eq-iff*)

lemma *finite-deriv*: $\text{finite } (\text{deriv } s) \longleftrightarrow (\exists m. f [s] m = [])$

proof

assume *finite* (*deriv s*)

then obtain *N* **where** $m: N \in \text{fst } ' \text{deriv } s \forall k. k \in \text{fst } ' \text{deriv } s \longrightarrow k \leq N$

by (*metis deriv0 empty-iff finite-imageI image-is-empty eq-Max-iff*)

then have $f [s] (\text{Suc } N) = []$

by (*metis Suc-n-not-le-n f image-eqI list.exhaust list.set-intros(1) split-pairs*)

then show $\exists m. f [s] m = [] \dots$

next

assume $\exists m. f [s] m = []$

then obtain *m* **where** $f [s] m = [] \dots$

then have $\bigwedge d. f [s] (m+d) = []$

using *f-upwards* **by** *blast*

then show *finite* (*deriv s*)

by (*metis empty-iff f list.set(1) FailingPath.is-path-f FailingPath-def surjective-pairing*)

qed

definition *prove'* :: $nseq \text{ list} \Rightarrow \text{bool}$ **where**

$\text{prove}' s \longleftrightarrow (\exists m. ((\text{concat} \circ \text{map subs}) \overset{\sim}{\sim} m) s = [])$

lemma *prove'*:

$\text{prove}' l \longleftrightarrow l = [] \vee \text{prove}' (\text{concat } (\text{map subs } l))$

proof (*cases* $\langle l = [] \rangle$)

case *True*

then show *?thesis*

by (*simp add: prove'-def exI [of - 0]*)

next

have $*$: $\langle ((\text{concat} \circ \text{map subs}) \overset{\sim}{\sim} m) (\text{concat } (\text{map subs } l)) = ((\text{concat} \circ \text{map subs}) \overset{\sim}{\sim} \text{Suc } m) l \rangle$

for *m*

by (*simp only: funpow-Suc-right*) *simp*

case *False*

then have $\langle \text{prove}' l \longleftrightarrow \text{prove}' (\text{concat } (\text{map subs } l)) \rangle$

apply (*simp add: prove'-def*)

apply (*simp only: **)

apply (*metis funpow-0 nat.collapse*)

done

with *False* **show** *?thesis*

by *simp*

qed

definition *prove* :: $nseq \Rightarrow \text{bool}$

where $\text{prove } s = \text{prove}' ([s])$

lemma *finite-deriv-prove: finite (deriv s) = prove s*
by (*simp add: finite-deriv prove-def prove'-def f-def comp-def*)

2.12 Computation

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

— convert to our form

lemma $((\exists x. A x \vee B x) \longrightarrow ((\exists x. B x) \vee (\exists x. A x)))$
 $= ((\forall x. \neg A x \wedge \neg B x) \vee ((\exists x. B x) \vee (\exists x. A x)))$
by *blast*

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:*

$x \in \text{set } [] \longleftrightarrow \text{False}$
 $x \in \text{set } (y \# \text{ys}) \longleftrightarrow x = y \vee x \in \text{set } \text{ys}$
by *simp-all*

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)])*

unfolding *my-f-def finite-deriv-prove prove-def prove'-Nil prove'-Cons ss*
by (*simp add: prove'-Nil*)

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 using soundness completeness finite-deriv-prove by auto*

ML \langle

```

fun max x y = if x > y then x else y;

fun concat [] = []
  | concat (a::list) = a @ (concat list);

type pred = int;

type var = int;

datatype 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;

fun preSuc [] = []
  | preSuc (a::list) = if a = 0 then preSuc list else (a-1)::(preSuc list);

fun fv (PAtom (-,vs)) = vs
  | fv (NAtom (-,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);

fun 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 (fn 0 => 0 | v => (r (v-1))+1) f)
  | subst r (FEx f) = FEx (subst (fn 0 => 0 | v => (r (v-1))+1) f);

fun fst body w = subst (fn 0 => w | v => v-1) body;

fun s-of-ns ns = map (fn (-,y) => y) ns;

fun ns-of-s s = map (fn y => (0,y)) s;

fun sfv s = concat (map fv s);

fun maxvar [] = 0
  | maxvar (a::list) = max a (maxvar list);

fun newvar vs = if vs = [] then 0 else (maxvar vs)+1;

fun test [] - = false
  | test ((-,y)::list) z = if y = z then true else test list z;

```

```

fun subs [] = [[]]
  | subs (x::xs) = let val (n,f') = x in case f' of
      PAtom (p,vs) => if test xs (NAtom (p,vs)) then [] else [xs @ [(0,PAtom
(p,vs))]]
      | NAtom (p,vs) => if test xs (PAtom (p,vs)) 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 n),(n+1,f')]]
    end;

fun step s = concat (map subs s);

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

fun prove s = prove' [s];

fun check f = (prove o ns-of-s) [f];

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

check my-f;

>

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.