

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

Tom Ridge

June 16, 2019

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	9
2.5	Soundness	10
2.6	Contains, Considers	18
2.7	Models 2	23
2.8	Falsifying Model From Failing Path	23
2.9	Completeness	25
2.10	Sound and Complete	26
2.11	Algorithm	26
2.12	Computation	29
3	Optimisation and Extension	31
4	OCaml Implementation	32

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
```

```
declare ex-image-cong-iff [simp del]
```

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

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

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

definition

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

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

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

where

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

definition

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

lemma *sfv-nil*: $\text{sfv } [] = []$ **by** (*force simp: sfv-def*)
lemma *sfv-cons*: $\text{sfv } (a\#list) = (\text{fv } a) @ (\text{sfv } list)$ **by** (*force simp: sfv-def*)

primrec *maxvar* :: $\text{var } list \Rightarrow \text{var}$

where

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

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

by (*induct vs*) (*auto simp: max-def*)

definition

newvar :: $\text{var } list \Rightarrow \text{var}$ **where**

$\text{newvar } vs = (\text{if } vs = [] \text{ then } 0 \text{ else } \text{Suc } (\text{maxvar } vs))$

— note that for *newvar* to be constructive, need an operation to get a different var from a given set

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

using *length-pos-if-in-set maxvar newvar-def* **by** *force*

primrec *subs* :: $\text{nseq} \Rightarrow \text{nseq } list$

where

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

2.2 Derivations

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

where

$\text{is-axiom } [] = \text{False}$
 $| \text{is-axiom } (a\#list) = ((? p \text{ vs. } a = \text{PAtom } p \text{ vs} \ \& \ \text{NAtom } p \text{ vs} : \text{set } list) \ | \ (? p \text{ vs. } a = \text{NAtom } p \text{ vs} \ \& \ \text{PAtom } p \text{ vs} : \text{set } list))$

inductive-set

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

for *s* :: nseq

where

init: $(0, s) \in \text{deriv } s$
| *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*: $(n, (m, FEx \ f) \# xs) \in \text{deriv}(nfs) \implies \sim \text{is-axiom } (s\text{-of-ns } ((m, FEx \ f) \# xs)) \implies (\text{Suc } n, xs@[0, \text{finst } f \ m], (\text{Suc } m, FEx \ f]) \in \text{deriv}(nfs)$
apply(*auto intro: step simp add: Let-def s-of-ns-def*)
done

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

lemmas[*simp*] = *init*
lemmas [*intro*] = *init step*

lemma *deriv0*[*simp*]: $(0, x) \in \text{deriv } y = (x = y)$
apply(*rule*)
apply(*erule deriv.cases*) **apply**(*simp*) **apply**(*simp*)
apply(*simp*)
done

lemma *deriv-upwards*: $(n, list) \in \text{deriv } s \implies \sim \text{is-axiom } (s\text{-of-ns } (list)) \implies (\exists zs. (\text{Suc } n, zs) \in \text{deriv } s \ \& \ zs : \text{set } (\text{subs } list))$
apply(*case-tac list*) **apply** *force*
apply(*case-tac a*) **apply**(*case-tac b*)
apply(*simp add: Let-def*) **apply**(*rule*) **apply**(*simp add: s-of-ns-def*) **apply**(*force dest: not-is-axiom-subst*)
apply(*simp add: Let-def*) **apply**(*rule*) **apply**(*simp add: s-of-ns-def*) **apply**(*force dest: not-is-axiom-subst*)
apply(*simp add: Let-def*) **apply**(*force dest: not-is-axiom-subst*)
apply(*simp add: Let-def*) **apply**(*force dest: not-is-axiom-subst*)
apply(*simp add: Let-def*) **apply**(*force dest: not-is-axiom-subst*)

apply(simp add: Let-def) **apply**(force dest: not-is-axiom Subs)
done

lemma deriv-downwards : $(\text{Suc } n, x) \in \text{deriv } s \implies \exists y. (n, y) \in \text{deriv } s \ \& \ x : \text{set } (\text{subs } y) \ \& \ \sim \text{is-axiom } (s\text{-of-ns } y)$
apply(erule deriv.cases)
apply(simp)
apply(simp add: s-of-ns-def Let-def)
apply(rule-tac $x=xa$ in exI) **apply**(simp)
apply(case-tac xa) **apply**(simp)
apply(case-tac a) **apply**(case-tac b)
apply(auto simp add: Let-def)
apply (rename-tac[!] nat lista a)
apply(subgoal-tac NAtom nat lista \in snd ‘ set list) **apply**(simp) **apply**(force)
apply(subgoal-tac PAtom nat lista \in snd ‘ set list) **apply**(simp) **apply**(force)
done

lemma deriv-deriv-child[rule-format]: $\forall x y. (\text{Suc } n, x) \in \text{deriv } y = (\exists z. z : \text{set } (\text{subs } y) \ \& \ \sim \text{is-axiom } (s\text{-of-ns } y) \ \& \ (n, x) \in \text{deriv } z)$
apply(induct n)
apply(rule, rule) **apply**(rule) **apply**(frule-tac deriv-downwards) **apply**(simp)
apply(simp) **apply**(rule step) **apply**(simp) **apply**(simp)
apply(blast dest!: deriv-downwards elim: deriv.cases) — blast needs some help
with the reasoning, hence derivSucE
done

lemma deriv-progress: $(n, a\#list) \in \text{deriv } s \implies \sim \text{is-axiom } (s\text{-of-ns } (a\#list)) \implies (\exists zs. (\text{Suc } n, \text{list}@zs) \in \text{deriv } s)$
apply(subgoal-tac $a\#list \neq []$) **prefer** 2 **apply**(simp)
apply(case-tac a) **apply**(case-tac b)
apply(force dest: not-is-axiom Subs)+
done

definition

$inc :: \text{nat} * \text{nseq} \implies \text{nat} * \text{nseq}$ **where**
 $inc = (\% (n, fs). (\text{Suc } n, fs))$

lemma inj-inc[simp]: inj inc
apply(simp add: inc-def) **apply**(simp add: inj-on-def) **done**

lemma deriv: $\text{deriv } y = \text{insert } (0, y) (inc \text{ ‘ } (\text{Union } (\text{deriv } \text{ ‘ } \{w. \sim \text{is-axiom } (s\text{-of-ns } y) \ \& \ w : \text{set } (\text{subs } y)\}))$)
apply(rule set-eqI)
apply(simp add: split-paired-all)
apply(case-tac a)
apply(force simp: inc-def)
apply(force simp: deriv-deriv-child inc-def)

done

lemma *deriv-is-axiom*: $is-axiom (s-of-ns s) ==> deriv s = \{(0,s)\}$
apply(*rule*)
apply(*rule*)
apply(*case-tac x*) **apply**(*simp*)
apply(*erule-tac deriv.induct*) **apply**(*force*) **apply**(*simp add: s-of-ns-def*) **ap-**
ply(*case-tac s*) **apply**(*simp*) **apply**(*case-tac aa*) **apply**(*case-tac ba*)
apply(*simp-all add: Let-def*)
done

lemma *is-axiom-finite-deriv*: $is-axiom (s-of-ns s) ==> finite (deriv s)$
apply(*simp add: deriv-is-axiom*) **done**

2.3 Failing path

primrec *failing-path* :: $(nat * nseq) set ==> nat ==> (nat * nseq)$

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*: $infinite (deriv \ s) ==> f \ 0 \in (deriv \ s) \ \& \ fst (f \ 0) = 0 \ \& \ infinite (deriv (snd (f \ 0))) \ \& \sim is-axiom (s-of-ns (snd (f \ 0)))$

apply(*simp add: f*) **apply**(*rule someI-ex*) **apply**(*rule-tac x=(0,s)*) **in** *exI*) **ap-**
ply(*force simp add: deriv-is-axiom*) **done**

lemma *infinite-union*: $finite \ Y ==> infinite (Union (f ' Y)) ==> \exists y. y \in Y \ \& \ infinite (f \ y)$

apply(*rule ccontr*) **apply**(*simp*) **done**

lemma *infinite-inj-infinite-image*: $inj-on \ f \ Z ==> infinite (f ' Z) = infinite \ Z$

apply(*rule ccontr*)

apply(*simp*)

apply(*force dest: finite-imageD*)

done

lemma *inj-inj-on*: $inj \ f ==> inj-on \ f \ A$

by(*blast intro: subset-inj-on*)

lemma *collect-disj*: $\{x. P \ x \mid Q \ x\} = \{x. P \ x\} \ Un \ \{x. Q \ x\}$ **by**(*force*)

lemma *t*: $\text{finite } \{w. P w\} \implies \text{finite } \{w. Q w \ \& \ P w\}$
by (*rule finite-subset, auto*)

lemma *finite Subs*: $\text{finite } \{w. \sim \text{is-axiom } (s\text{-of-ns } y) \ \& \ w : \text{set } (subs \ y)\}$
apply(*case-tac y*) **apply**(*simp add: s-of-ns-def*)
apply(*case-tac a*) **apply**(*case-tac b*)
apply(*simp-all only: Subs.simps*)
apply(*auto intro: t simp add: collect-disj*)
done

lemma (*in loc1*) *fSuc*:
shows []
 $f \ n \in \text{deriv } s \ \& \ \text{fst } (f \ n) = n \ \& \ \text{infinite } (\text{deriv } (\text{snd } (f \ n))) \ \& \ \sim \text{is-axiom } (s\text{-of-ns } (\text{snd } (f \ n)))$
[] $\implies f \ (\text{Suc } n) \in \text{deriv } s \ \& \ \text{fst } (f \ (\text{Suc } n)) = \text{Suc } n \ \& \ (\text{snd } (f \ (\text{Suc } n))) : \text{set } (subs \ (\text{snd } (f \ n))) \ \& \ \text{infinite } (\text{deriv } (\text{snd } (f \ (\text{Suc } n)))) \ \& \ \sim \text{is-axiom } (s\text{-of-ns } (\text{snd } (f \ (\text{Suc } n))))$
apply(*simp add: Let-def f*)
apply(*rule-tac someI-ex*)
apply(*simp only: f[symmetric]*)
apply(*drule-tac subst[OF deriv[of snd (f n)]]*)
apply(*simp only: finite-insert*) **apply**(*subgoal-tac infinite* ($\bigcup (\text{deriv } \{w. \sim \text{is-axiom } (s\text{-of-ns } (\text{snd } (f \ n))) \ \& \ w : \text{set } (subs \ (\text{snd } (f \ n)))\})$)))
apply(*drule-tac infinite-union[OF finite-Subs]*) **apply**(*erule exE*) **apply**(*rule-tac*
 $x = (\text{Suc } n, y)$ **in** *exI*)
apply(*clarify*) **apply**(*simp*) **apply**(*case-tac f n*) **apply**(*simp add: step*) **ap-**
ply(*force simp add: is-axiom-finite-deriv*)
apply(*force simp add: infinite-inj-infinite-image inj-inj-on*)
done

lemma (*in loc1*) *is-path-f-0*: $\text{infinite } (\text{deriv } s) \implies f \ 0 = (0, s)$
apply(*subgoal-tac* $f \ 0 \in \text{deriv } s \ \& \ \text{fst } (f \ 0) = 0$)
prefer 2 **apply**(*frule-tac f0*) **apply**(*simp*)
apply(*case-tac f 0*) **apply**(*elim conjE, simp*)
done

lemma (*in loc1*) *is-path-f'*: $\text{infinite } (\text{deriv } s) \implies \forall n. f \ n \in \text{deriv } s \ \& \ \text{fst } (f \ n) = n \ \& \ \text{infinite } (\text{deriv } (\text{snd } (f \ n))) \ \& \ \sim \text{is-axiom } (s\text{-of-ns } (\text{snd } (f \ n)))$
apply(*rule*)
apply(*induct-tac n*)
apply(*simp add: f0*)
apply(*simp add: fSuc*)
done

lemma (*in loc1*) *is-path-f*: $\text{infinite } (\text{deriv } s) \implies \forall n. f \ n \in \text{deriv } s \ \& \ \text{fst } (f \ n) = n \ \& \ (\text{snd } (f \ (\text{Suc } n))) : \text{set } (subs \ (\text{snd } (f \ n))) \ \& \ \text{infinite } (\text{deriv } (\text{snd } (f \ n)))$
apply(*rule*)
apply(*frule-tac is-path-f'*) **apply**(*simp*) **apply**(*simp add: fSuc*)
done

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

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

lemma *and-lem*: $(a=c) \implies (b=d) \implies (a \& b) = (c \& d)$ **by** *simp*

lemma *or-lem*: $(a=c) \implies (b=d) \implies (a | b) = (c | d)$ **by** *simp*

lemma *ball-eq-ball*: $\forall x \in m. P\ x = Q\ x \implies (\forall x \in m. P\ x) = (\forall x \in m. Q\ x)$
by *blast*

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

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

apply(*induct-tac A*) **apply**(*simp*) **apply**(*case-tac a, force, force*) **done**

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

proof (*induction A*)

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) gr0I 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) gr0I preSuc)
qed

```

```

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 = ( $\exists f. f \in \text{set } s \ \& \ \text{FEval } m \ e \ f$ )
  by (induct s) auto

```

```

lemma SEval-append: SEval m e (xs@ys) = ( (SEval m e xs) | (SEval m e ys) )
  by (induct xs) auto

```

```

lemma all-eq-all:  $\forall x. P \ x = Q \ x \implies (\forall x. P \ x) = (\forall x. Q \ x)$  by blast

```

```

lemma all-conj:  $(\forall x. A \ x \ \& \ B \ x) = ((\forall x. A \ x) \ \& \ (\forall x. B \ x))$  by blast

```

```

lemma SEval-cong:  $(\forall x. x \in \text{set } (sfv \ s) \ \longrightarrow \ e1 \ x = e2 \ x) \ \longrightarrow \ SEval \ m \ e1 \ s = SEval \ m \ e2 \ s$ 
  apply(induct s) apply(simp)
  apply(simp) apply(intro all impI)
  apply(rule or-lem) apply(simp add: sfv-cons) apply(simp add: FEval-cong)
  apply(simp add: sfv-cons)
  done

```

```

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 =  $(\forall MI \ e. \ is\text{-env} \ MI \ e \ \longrightarrow \ SEval \ MI \ e \ s)$ 

```

2.5 Soundness

```

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

```

apply(*rule ext*) **apply**(*force*) **done**

lemma *FEval-subst*: $\forall e f. (FEval\ MI\ e\ (subst\ f\ A)) = (FEval\ MI\ (e\ o\ f)\ A)$
apply(*induct A*)
apply(*simp add: Let-def*) **apply**(*simp only: fold-compose1 map-map*) **ap-**
ply(*blast*)
apply(*simp add: Let-def*) **apply**(*simp only: fold-compose1 map-map*) **ap-**
ply(*blast*)
apply(*simp*)
apply(*simp*)
apply(*simp (no-asm-use)*) **apply**(*simp*) **apply**(*rule,rule*) **apply**(*rule ball-eq-ball*)
apply(*rule*) **apply**(*simp add: bump-def*)
apply(*subgoal-tac (%u. case-nat m e (case u of 0 => 0 | Suc n => Suc (f n)))*)
= (*case-nat m (%n. e (f n))*) **apply**(*simp*)
apply(*rule ext*) **apply**(*case-tac u*)
apply(*simp*) **apply**(*simp*)
apply(*simp (no-asm-use)*) **apply**(*simp*) **apply**(*rule,rule*) **apply**(*rule beq-eq-beq*)
apply(*rule*) **apply**(*simp add: bump-def*)
apply(*subgoal-tac (%u. case-nat m e (case u of 0 => 0 | Suc n => Suc (f n)))*)
= (*case-nat m (%n. e (f n))*) **apply**(*simp*)
apply(*rule ext*) **apply**(*case-tac u*)
apply(*simp*) **apply**(*simp*)
done

lemma *FEval-finst*: $FEval\ mo\ e\ (finst\ A\ u) = FEval\ mo\ (case-nat\ (e\ u)\ e)\ A$
apply(*simp add: finst-def*)
apply(*simp add: FEval-subst*)
apply(*subgoal-tac (e o case-nat u (%n. n)) = (case-nat (e u) e)*) **apply**(*simp*)
apply(*rule ext*)
apply(*case-tac x,auto*)
done

lemma *ball-maxscope*: $(\forall x \in m. P\ x \mid Q) ==> (\forall x \in m. P\ x) \mid Q$ **by** *blast*

lemma *sound-FALL*: $u \notin set\ (sfv\ (FALL\ f\ \#s)) ==> Svalid\ (s@[finst\ f\ u]) ==>$
Svalid (FALL f #s)
apply(*simp add: Svalid-def del: SEval.simps*)
apply(*rule allI*)
apply(*rule allI*)
apply(*rename-tac M I*)
apply(*rule allI*) **apply**(*rule*)
apply(*simp*)
apply(*simp add: SEval-append*)
apply(*rule ball-maxscope*)
apply(*rule*)
apply(*simp add: FEval-finst*)

apply(*drule-tac* $x=M$ **in** *spec*, *drule-tac* $x=I$ **in** *spec*) — this is the goal

apply(*drule-tac* $x=e(u:=m)$ **in** *spec*) **apply**(*erule* *impE*) **apply**(*simp* *add: is-env-def*)
apply(*erule* *disjE*)
apply(*rule* *disjI2*)
apply(*subgoal-tac* *SEval* (M,I) ($e(u:=m)$) $s = SEval$ (M,I) e s)
apply(*simp*)
apply(*rule* *SEval-cong*[*rule-format*]) **apply**(*simp* *add: sfv-cons*) **apply**(*force*)

apply(*rule* *disjI1*)
apply(*simp*)
apply(*subgoal-tac* *FEval* (M,I) (*case-nat* m ($e(u:=m)$))) $f = FEval$ (M,I)
(*case-nat* m e) f)
apply(*simp*)
apply(*rule* *FEval-cong*[*rule-format*])

apply(*case-tac* x , *simp*)
apply(*simp*)
apply(*simp* *only: preSuc*[*rule-format*, *symmetric*])
apply(*subgoal-tac* $nat \in set$ (fv (*FAll* f))) **prefer** 2 **apply**(*simp*)

apply(*force* *simp: sfv-cons*)
done

— phew, that really was a bit more difficult than expected
— note that we can avoid maxscoping at the cost of instantiating the hyp twice—
an additional time for M

— different proof, instantiating quantifier twice, avoiding maxscoping

lemma *sound-FAll'*: $u \notin set$ (sfv (*FAll* $f\#s$)) $==>$ *Svalid* ($s@[finst$ f $u]$) $==>$
Svalid (*FAll* $f\#s$)

apply(*simp* *add: Svalid-def* *del: SEval.simps*)
apply(*rule* *allI*)
apply(*rule* *allI*)
apply(*rename-tac* M I)
apply(*rule* *allI*) **apply**(*rule*)
apply(*simp*)
apply(*simp* *add: SEval-append*)
apply(*drule-tac* $x=M$ **in** *spec*, *drule-tac* $x=I$ **in** *spec*)
apply(*frule-tac* $x=e$ **in** *spec*) **apply**(*erule* *impE*) **apply**(*simp* *add: is-env-def*)
apply(*case-tac* *SEval* (M, I) e s) **apply**(*simp*)
apply(*simp*)

— second instantiation
apply(*simp* *add: FEval-finst*)
apply(*rule*)
apply(*drule-tac* $x=e(u:=m)$ **in** *spec*) **apply**(*erule* *impE*) **apply**(*simp* *add: is-env-def*)

apply(*erule* *disjE*)
apply(*subgoal-tac* *SEval* (M,I) ($e(u:=m)$) $s = SEval$ (M,I) e s)
apply(*simp*)

```

apply(rule SEval-cong[rule-format]) apply(simp add: sfv-cons) apply(force)
apply(simp)
apply(subgoal-tac FEval (M,I) (case-nat m (e(u :=m))) f = FEval (M,I)
(case-nat m e) f)
apply(simp)
apply(rule FEval-cong[rule-format])
apply(case-tac x, simp)
apply(simp)
apply(simp only: preSuc[rule-format, symmetric])
apply(subgoal-tac nat ∈ set (fv (FAll f))) prefer 2 apply(simp)
apply(force simp: sfv-cons)
done
— not much better, probably slightly worse

```

```

lemma sound-FEx: Svalid (s@[finst f u, FEx f]) ==> Svalid (FEx f#s)
apply(simp add: Svalid-def del: SEval.simps)
apply(rule allI)
apply(rule allI)
apply(rename-tac ms m)
apply(rule) apply(rule)
apply(simp)
apply(simp add: SEval-append)
apply(simp add: FEval-finst)

```

```

apply(drule-tac x=ms in spec, drule-tac x=m in spec)
apply(drule-tac x=e in spec) apply(erule impE) apply(assumption)
apply(erule disjE)
apply(simp)
apply(erule disjE)
apply(rule disjI1)
apply(rule-tac x=e u in bexI) apply(simp) apply(simp add: is-env-def)
apply(force)
done

```

```

lemma max-exists: finite (X::nat set) ==> X ≠ {} --> (∃x. x ∈ X & (∀y. y
∈ X --> y ≤ x))
apply(erule-tac finite-induct)
apply(force)
apply(rule)
apply(case-tac F = {})
apply(force)
apply(erule impE) apply(force)
apply(elim exE conjE)
apply(rule-tac x=max x xa in exI)
apply(rule) apply(simp add: max-def)
apply(simp add: max-def) apply(force)

```

done
 — poor max lemmas in lib

lemma *inj-finite-image-eq-finite*: $\text{inj-on } f \ Z \implies \text{finite } (f \ ' \ Z) = \text{finite } Z$
by (*blast intro: finite-imageD*)

lemma *finite-inc*: $\text{finite } (\text{inc} \ ' \ X) = \text{finite } X$
apply(*rule inj-finite-image-eq-finite*)
apply(*rule-tac B=UNIV in subset-inj-on*)
apply(*auto*)
done

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

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

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

lemma *is-FEx[simp]*: $\sim \text{is-FEx } (PAtom \ p \ vs)$
 $\& \sim \text{is-FEx } (NAtom \ p \ vs)$
 $\& \sim \text{is-FEx } (FConj \ f \ g)$
 $\& \sim \text{is-FEx } (FDisj \ f \ g)$
 $\& \sim \text{is-FEx } (FAll \ f)$
by(*force simp: is-FEx-def*)

lemma *index0*: $\text{init } s \implies \forall u \ m \ A. (n, u) \in \text{deriv } s \dashrightarrow (m, A) \in (\text{set } u) \dashrightarrow (\sim \text{is-FEx } A) \dashrightarrow m = 0$
apply(*induct-tac n*)
apply(*rule,rule,rule,rule,rule,rule*) **apply**(*simp*) **apply**(*force simp add: init-def*)
apply(*rule,rule,rule,rule,rule,rule*)
 — inversion on $(Suc \ n, u) \in \text{deriv } s$
apply(*drule-tac deriv-downwards*) **apply**(*elim exE conjE*)
apply(*case-tac y*) **apply**(*simp*)
apply(*case-tac a*) **apply**(*case-tac b*)
 apply(*force simp add: Let-def s-of-ns-def*)
 apply(*force simp add: Let-def s-of-ns-def*)
 apply(*force simp add: Let-def s-of-ns-def*)

```

apply(force simp add: Let-def s-of-ns-def)
apply(force simp add: Let-def s-of-ns-def)
apply(force simp add: is-FEx-def Let-def s-of-ns-def)
done

lemma soundness': init s ==> finite (deriv s) ==> m ∈ (fst ‘ (deriv s)) ==>
∀ y u. (y,u) ∈ (deriv s) --> y ≤ m ==> ∀ n t. h = m - n & (n,t) ∈ deriv s
--> Svalid (s-of-ns t)
apply(induct-tac h)
  — base case
apply(simp) apply(rule,rule,rule) apply(elim conjE)
apply(subgoal-tac n=m) prefer 2 apply(force)
apply(simp)
apply(simp add: Svalid-def) apply(rule,rule) apply(rename-tac gs g) ap-
ply(rule) apply(rule) apply(simp add: SEval-def2)
apply(case-tac is-axiom (s-of-ns t))
  — base case, is axiom
apply(simp add: s-of-ns-def) apply(case-tac t) apply(simp) apply(simp)
apply(erule disjE)
  — base case, is axiom, starts with PAtom
apply(elim conjE exE)
apply(subgoal-tac FEval (gs,g) e (PAtom p vs) | FEval (gs,g) e (NAtom p
vs))
apply(erule disjE) apply(force) apply(rule-tac x=NAtom p vs in exI)
apply(force)
apply(simp add: Let-def)
  — base case, is axiom, starts with NAtom
apply(elim conjE exE)
apply(subgoal-tac FEval (gs,g) e (PAtom p vs) | FEval (gs,g) e (NAtom p vs))
apply(erule disjE) apply(rule-tac x=PAtom p vs in exI) apply(force) ap-
ply(force)
apply(simp add: Let-def)

  — base case, not is axiom: if not a satax, then subs holds... but this can't be
apply(drule-tac deriv-upwards) apply(assumption) apply(elim conjE exE) ap-
ply(force)

  — step case, by case analysis

apply(intro allI impI) apply(elim exE conjE)

apply(case-tac is-axiom (s-of-ns t))
  — step case, is axiom
apply(simp add: Svalid-def) apply(rule,rule) apply(rename-tac gs g) apply(rule)
apply(rule) apply(simp add: SEval-def2)
apply(simp add: s-of-ns-def) apply(case-tac t) apply(simp) apply(simp)
apply(erule disjE)
apply(elim conjE exE)
apply(subgoal-tac FEval (gs,g) e (PAtom p vs) | FEval (gs,g) e (NAtom p

```

$vs))$
apply(*erule disjE*) **apply**(*force*) **apply**(*rule-tac x=NAtom p vs in exI*)
apply(*blast*)
apply(*simp add: Let-def*)
apply(*elim conjE exE*)
apply(*subgoal-tac FEval (gs,g) e (PAtom p vs) | FEval (gs,g) e (NAtom p vs)*)
apply(*erule disjE*) **apply**(*rule-tac x=PAtom p vs in exI*) **apply**(*blast*) **ap-**
ply(*simp*) **apply**(*force*)
apply(*simp add: Let-def*)

— we hit FAll/ FEx cases first

apply(*case-tac $\exists (a::nat) f list. t = (a, FAll f) \# list$*)
apply(*elim exE*) **apply**(*simp*)
apply(*subgoal-tac a = 0*)
prefer 2
apply(*rule-tac n=na and u=t and A=FAll f in index0[rule-format]*)
apply(*assumption*) **apply**(*simp*) **apply**(*simp*) **apply**(*simp*)
apply(*frule-tac deriv.step*) **apply**(*simp add: Let-def*) — nice use of simp to
instantiate
apply(*drule-tac x=Suc na in spec, drule-tac x=list @ [(0, first f (newvar (sfv*
(*s-of-ns t*)))] **in spec**) **apply**(*erule impE, simp*)
apply(*subgoal-tac newvar (sfv (s-of-ns t)) \notin set (sfv (s-of-ns t))*)
prefer 2 **apply**(*rule newvar*)
apply(*simp*)
apply(*simp add: s-of-ns-def*)
apply(*frule-tac sound-FAll*) **apply**(*simp*) **apply**(*simp*)

apply(*case-tac $\exists a f list. t = (a, FEx f) \# list$*)
apply(*elim exE*)
apply(*frule-tac deriv.step*) **apply**(*simp add: Let-def*) — nice use of simp to
instantiate
apply(*drule-tac x=Suc na in spec, drule-tac x=list @ [(0, first f a), (Suc a,*
FEx f)] in spec) **apply**(*erule impE, assumption*)
apply(*drule-tac x=Suc na in spec, drule-tac x=list @ [(0, first f a), (Suc a, FEx*
f)] in spec) **apply**(*erule impE*) **apply**(*rule*) **apply**(*arith*) **apply**(*assumption*)
apply(*subgoal-tac Svalid (s-of-ns (list@[0, first f a], (Suc a, FEx f)))*)
apply(*simp add: s-of-ns-def*)
apply(*frule-tac sound-FEx*) **apply**(*simp*) **apply**(*simp*)

— now for other cases

— case empty list

apply(*case-tac t*) **apply**(*simp*)
apply(*frule-tac step*) **apply**(*simp*) **apply**(*simp*) **apply**(*drule-tac x=Suc na in*
spec) **back** **apply**(*drule-tac x=[] in spec*) **apply**(*erule impE*) **apply**(*rule*) **ap-**
ply(*arith*) **apply**(*assumption*) **apply**(*assumption*)

apply(*simp add: Svalid-def*) **apply**(*rule, rule*) **apply**(*rename-tac gs g*) **apply**(*rule*)
apply(*rule*) **apply**(*simp add: SEval-def2*)

— na t in deriv, so too is subs
 — if not a satax, then subs holds...
apply(*case-tac a*)
apply(*case-tac b*)
 apply(*simp del: FEval.simps*) **apply**(*frule-tac patom*) **apply**(*assumption*)
 apply(*rename-tac nat lista*)
 apply(*frule-tac x=Suc na in spec, drule-tac x=list @ [(0, PAtom nat lista)]*)
in spec)
 apply(*erule impE*) **apply**(*arith*)
 apply(*drule-tac x=gs in spec, drule-tac x=g in spec, drule-tac x=e in spec*)
apply(*erule impE*) **apply**(*simp add: is-env-def*)
 apply(*elim exE conjE*) **apply**(*rule-tac x=f in exI*) **apply**(*simp add: s-of-ns-def*) — weirdly, simp succeeds, force and blast fail
 apply(*simp del: FEval.simps*) **apply**(*frule-tac natom*) **apply**(*assumption*)
 apply(*rename-tac nat lista*)
 apply(*frule-tac x=Suc na in spec, drule-tac x=list @ [(0, NAtom nat lista)]*)
in spec)
 apply(*erule impE*) **apply**(*arith*)
 apply(*drule-tac x=gs in spec, drule-tac x=g in spec, drule-tac x=e in spec*)
apply(*erule impE, simp*)
 apply(*elim exE conjE*) **apply**(*rule-tac x=f in exI*) **apply**(*simp add: s-of-ns-def*)
 apply(*simp del: FEval.simps*) **apply**(*frule-tac fconj1*) **apply**(*assumption*)
apply(*frule-tac fconj2*) **apply**(*assumption*)
 apply(*rename-tac form1 form2*)
 apply(*frule-tac x=Suc na in spec, drule-tac x=list @ [(0, form1)] in spec*)
 apply(*erule impE*) **apply**(*arith*)
 apply(*drule-tac x=gs in spec, drule-tac x=g in spec, drule-tac x=e in spec*)
apply(*erule impE, simp*) **apply**(*elim exE conjE*)
 apply(*drule-tac x=Suc na in spec, drule-tac x=list @ [(0, form2)] in spec*)
 apply(*erule impE*) **apply**(*arith*)
 apply(*drule-tac x=gs in spec, drule-tac x=g in spec, drule-tac x=e in spec*)
apply(*erule impE, simp*) **apply**(*elim exE conjE*)
 apply(*simp only: s-of-ns-def*)
 apply(*simp*)
 apply(*elim disjE*)
 apply(*simp*) **apply**(*rule-tac x=FConj form1 form2 in exI*) **apply**(*simp*)
 apply(*simp*) **apply**(*rule-tac x=fa in exI*) **apply**(*simp*)
 apply(*simp*) **apply**(*rule-tac x=f in exI*) **apply**(*simp*)
 apply(*rule-tac x=f in exI, simp*)
 apply(*simp del: FEval.simps*) **apply**(*frule-tac fdisj*) **apply**(*assumption*)
 apply(*rename-tac form1 form2*)
 apply(*frule-tac x=Suc na in spec, drule-tac x=list @ [(0, form1),(0,form2)]*)
in spec)
 apply(*erule impE*) **apply**(*arith*)
 apply(*drule-tac x=gs in spec, drule-tac x=g in spec, drule-tac x=e in spec*)
apply(*erule impE, simp*) **apply**(*elim exE conjE*)
 apply(*simp add: s-of-ns-def*)
 apply(*elim disjE*)
 apply(*simp*) **apply**(*rule-tac x=FDisj form1 form2 in exI*) **apply**(*simp*)

```

apply(simp) apply(rule-tac x=FDIsj form1 form2 in exI) apply(simp)
apply(rule-tac x=f in exI) apply(simp)
  — all case
apply(force)
apply(force)
done

```

```

lemma [simp]: s-of-ns (ns-of-s s) = s
apply(induct s)
apply(simp add: s-of-ns-def ns-of-s-def)
apply(simp add: s-of-ns-def ns-of-s-def)
done

```

```

lemma soundness: finite (deriv (ns-of-s s)) ==> Svalid s
apply(subgoal-tac init (ns-of-s s))
prefer 2 apply(simp add: init-def ns-of-s-def)
apply(subgoal-tac finite (fst ' (deriv (ns-of-s s)))) prefer 2 apply(simp)
apply(frule-tac max-exists) apply(erule impE) apply(simp) apply(subgoal-tac
(0,ns-of-s s) ∈ deriv (ns-of-s s)) apply(force) apply(simp)
apply(elim exE conjE)
apply(frule-tac soundness') apply(assumption) apply(assumption) apply(force)

apply(drule-tac x=0 in spec, drule-tac x=ns-of-s s in spec)
apply(force )
done

```

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)

```

```

lemma (in loc1) progress: infinite (deriv s) ==> snd (f n) = a#list --> (∃ zs'.
snd (f (Suc n)) = list@zs')
apply(subgoal-tac (snd (f (Suc n))) : set (subs (snd (f n)))) defer apply(frule-tac
is-path-f) apply(blast)
apply(case-tac a)
apply(case-tac b)
apply(safe)
apply(simp-all add: Let-def split: if-splits)
apply(erule disjE)
apply(simp-all)
done

```

```

lemma (in loc1) contains-considers': infinite (deriv s) ==> ∀ n y ys. snd (f n)

```

```

= xs@y#ys --> (∃ m zs'. snd (f (n+m)) = y#zs')
  apply(induct-tac xs)
  apply(rule,rule,rule,rule) apply(rule-tac x=0 in exI) apply(rule-tac x=ys in
exI) apply(force)

  apply(rule,rule,rule,rule) apply(simp) apply(frule-tac progress) apply(erule
impE) apply(assumption)
  apply(erule exE) apply(simp)

  apply(drule-tac x=Suc n in spec)
  apply(case-tac y) apply(rename-tac u v)
  apply(drule-tac x=u in spec)
  apply(drule-tac x=v in spec)
  apply(erule impE) apply(force)

  apply(elim exE)
  apply(rule-tac x=Suc m in exI)
  apply(force)
done

```

lemma *list-decomp*[rule-format]: $v \in \text{set } p \longrightarrow (\exists \text{ xs ys. } p = \text{xs}@(\text{v}\#\text{ys}) \wedge v \notin \text{set } \text{xs})$

```

  apply(induct p)
  apply(force)
  apply(case-tac v=a)
  apply(force)
  apply(auto)
  apply(rule-tac x=a#xs in exI)
  apply(auto)
done

```

lemma (in *loc1*) *contains-considers*: $\text{infinite } (\text{deriv } s) \implies \text{contains } f \ n \ y \implies (\exists m. \text{considers } f \ (n+m) \ y)$

```

  apply(simp add: contains-def considers-def)
  apply(frule-tac list-decomp) apply(elim exE conjE)
  apply(frule-tac contains-considers'[rule-format]) apply(assumption) apply(elim
exE)
  apply(rule-tac x=m in exI)
  apply(force)
done

```

lemma (in *loc1*) *contains-propagates-patoms*[rule-format]: $\text{infinite } (\text{deriv } s) \implies \text{contains } f \ n \ (0, \text{PAtom } p \ \text{vs}) \longrightarrow \text{contains } f \ (n+q) \ (0, \text{PAtom } p \ \text{vs})$

```

  apply(induct-tac q) apply(simp)
  apply(rule)
  apply(erule impE) apply(assumption)
  apply(subgoal-tac ~is-axiom (s-of-ns (snd (f (n+na)))))) defer
  apply(subgoal-tac infinite (deriv (snd (f (n+na)))))) defer

```

```

  apply(force dest: is-path-f)
  defer
  apply(rule) apply(simp add: deriv-is-axiom)
  apply(simp add: contains-def)
  apply(drule-tac p=snd (f (n + na)) in list-decomp)
  apply(elim exE conjE)
  apply(case-tac xs)
  apply(simp)
  apply(subgoal-tac (snd (f (Suc (n + na)))) : set (subs (snd (f (n + na))))))
  apply(simp add: Let-def split: if-splits)
  apply(frule-tac is-path-f) apply(drule-tac x=n+na in spec) apply(force)
  apply(drule-tac progress)
  apply(erule impE) apply(force)
  apply(force)
  done

```

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)

```

  apply(induct-tac q) apply(simp)
  apply(rule)
  apply(erule impE) apply(assumption)
  apply(subgoal-tac ~is-axiom (s-of-ns (snd (f (n+na)))))) defer
  apply(subgoal-tac infinite (deriv (snd (f (n+na)))))) defer
  apply(force dest: is-path-f)
  defer
  apply(rule) apply(simp add: deriv-is-axiom)
  apply(simp add: contains-def)
  apply(drule-tac p = snd (f (n + na)) in list-decomp)
  apply(elim exE conjE)
  apply(case-tac xs)
  apply(simp)
  apply(subgoal-tac (snd (f (Suc (n + na)))) : set (subs (snd (f (n + na))))))
  apply(simp add: Let-def split: if-splits)
  apply(frule-tac is-path-f) apply(drule-tac x=n+na in spec) apply(force)
  apply(drule-tac progress)
  apply(erule impE) apply(force)
  apply(force)
  done

```

lemma (in loc1) contains-propagates-fconj: infinite (deriv s) ==> contains f n (0, FConj g h) ==> ($\exists y$. contains f (n+y) (0,g) | contains f (n+y) (0,h))

```

  apply(subgoal-tac ( $\exists l$ . considers f (n+l) (0,FConj g h))) defer apply(rule
contains-considers) apply(assumption) apply(assumption)
  apply(erule exE)
  apply(rule-tac x=Suc l in exI)
  apply(simp add: considers-def) apply(case-tac snd (f (n + l)), simp)
  apply(simp)
  apply(subgoal-tac (snd (f (Suc (n + l)))) : set (subs (snd (f (n + l))))))
  apply(simp add: contains-def Let-def) apply(force)

```

apply(*frule-tac is-path-f*) **apply**(*drule-tac x=n+l in spec*) **apply**(*force*)
done

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))*
apply(*subgoal-tac (∃ l. considers f (n+l) (0,FDisj g h))*) **defer** **apply**(*rule contains-considers*) **apply**(*assumption*) **apply**(*assumption*)
apply(*erule exE*)
apply(*rule-tac x=Suc l in exI*)
apply(*simp add: considers-def*) **apply**(*case-tac snd (f (n + l)), simp*)
apply(*simp*)
apply(*subgoal-tac (snd (f (Suc (n + l)))) : set (subs (snd (f (n + l))))*)
apply(*simp add: contains-def Let-def*)
apply(*frule-tac is-path-f*) **apply**(*drule-tac x=n+l in spec*) **apply**(*force*)
done

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
apply(*subgoal-tac (∃ l. considers f (n+l) (0,FALL g))*) **defer** **apply**(*rule contains-considers*)
apply(*assumption*) **apply**(*assumption*)
apply(*erule exE*)
apply(*rule-tac x=l in exI*)
apply(*simp add: considers-def*) **apply**(*case-tac snd (f (n + l)), simp*)
apply(*simp*)
apply(*subgoal-tac (snd (f (Suc (n + l)))) : set (subs (snd (f (n + l))))*)
apply(*simp add: contains-def Let-def*)
apply(*frule-tac is-path-f*) **apply**(*drule-tac x=n+l in spec*) **apply**(*force*)
done

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)))
apply(*subgoal-tac (∃ l. considers f (n+l) (m,FEx g))*) **defer** **apply**(*rule contains-considers*)
apply(*assumption*) **apply**(*assumption*)
apply(*erule exE*)
apply(*rule-tac x=Suc l in exI*)
apply(*simp add: considers-def*) **apply**(*case-tac snd (f (n + l)), simp*)
apply(*simp*)
apply(*subgoal-tac (snd (f (Suc (n + l)))) : set (subs (snd (f (n + l))))*)
apply(*simp add: contains-def Let-def*)
apply(*frule-tac is-path-f*) **apply**(*drule-tac x=n+l in spec*) **apply**(*force*)
done

— 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 $(\text{Suc } m, x)$, by induction on n

lemma (in *loc1*) *FEx-downward: infinite (deriv s) ==> init s ==> $\forall m. (\text{Suc } m, \text{FEx } g) \in \text{set } (\text{snd } (f \ n)) \dashrightarrow (\exists n'. (m, \text{FEx } g) \in \text{set } (\text{snd } (f \ n')))$*

```

apply(frule-tac is-path-f)
apply(induct-tac n)
apply(drule-tac x=0 in spec) apply(case-tac f 0) apply(force simp: init-def)
apply(intro allI impI)
apply(frule-tac x=Suc n in spec, elim conjE) apply(drule-tac x=n in spec, elim conjE)
apply(thin-tac (snd (f (Suc (Suc n)))) : set (subs (snd (f (Suc n))))))
apply(case-tac f n) apply(simp)
apply(case-tac b) apply(simp)
apply(case-tac aa) apply(case-tac ba)
apply(simp add: Let-def split: if-splits)
apply(simp add: Let-def split: if-splits)
apply(force simp add: Let-def)
apply(force simp add: Let-def)
apply(force simp add: Let-def)
apply(rename-tac form)
apply(case-tac (ab, FEx form) = (m, FEx g))
apply(rule-tac x=n in exI) apply(force)
apply(force simp add: Let-def)
done

```

lemma (in *loc1*) *FEx0: infinite (deriv s) ==> init s ==> $\forall n. \text{contains } f \ n \ (m, \text{FEx } g) \dashrightarrow (\exists n'. \text{contains } f \ n' \ (0, \text{FEx } g))$*

```

apply(simp add: contains-def)
apply(induct-tac m)
apply(force)
apply(intro allI impI) apply(erule exE)
apply(drule-tac FEx-downward[rule-format]) apply(assumption) apply(force)
apply(elim exE conjE)
apply(force)
done

```

lemma (in *loc1*) *FEx-upward': infinite (deriv s) ==> init s ==> $\forall n. \text{contains } f \ n \ (0, \text{FEx } g) \dashrightarrow (\exists n'. \text{contains } f \ n' \ (m, \text{FEx } g))$*

```

apply(intro allI impI)
apply(induct-tac m) apply(force)
apply(erule exE)
apply(frule-tac n=n' in contains-considers) apply(assumption)
apply(erule exE)
apply(rule-tac x=Suc(n'+m) in exI)
apply(frule-tac is-path-f)
apply(simp add: considers-def) apply(case-tac snd (f (n'+m))) apply(force)
apply(simp)

```

apply(*drule-tac* $x=n'+m$ **in** *spec*)
apply(*force simp add: contains-def considers-def Let-def*)
done
— FIXME contains and considers aren't buying us much

lemma (**in** *loc1*) *FEx-upward: infinite (deriv s) ==> init s ==> contains f n*
($m, FEx\ g$) ==> ($\forall m'. \exists n'. \text{contains } f\ n'\ (0, \text{first } g\ m^\wedge)$)
apply(*intro allI impI*)
apply(*subgoal-tac* $\exists n'. \text{contains } f\ n'\ (m', FEx\ g)$) **defer**
apply(*frule-tac* $m = m$ **and** $g = g$ **in** *FEx0*) **apply**(*assumption*)
apply(*drule-tac* $x=n$ **in** *spec*)
apply(*simp*)
apply(*elim exE*)
apply(*frule-tac* $g=g$ **and** $m=m'$ **in** *FEx-upward'*) **apply**(*assumption*)
apply (*blast dest: contains-propagates-fex intro: elim:*)
done

2.7 Models 2

axiomatization *ntou* :: $\text{nat} \Rightarrow U$
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*
apply(*simp add: uton-def*) **apply**(*simp add: ntou inv-f-f*) **done**

lemma *map-uton-ntou*[*simp*]: *map uton (map ntou xs) = xs*
apply(*induct xs, auto simp: uton-ntou*) **done**

lemma *ntou-uton*: $x \in \text{range } \text{ntou} \Rightarrow \text{ntou } (\text{uton } x) = x$
apply(*simp add: uton-def*)
apply(*simp add: f-inv-into-f*) **done**

2.8 Falsifying Model From Failing Path

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

locale *loc2* = *loc1* +
fixes *mo*
assumes *mo*: $mo = \text{model } s$

lemma *is-env-model-ntou*: *is-env (model s) ntou*
apply(*simp add: is-env-def*) **apply**(*simp add: model-def*) **done**

lemma (**in** *loc1*) [*simp*]: *infinite (deriv s) ==> init s ==> (contains f n (m,A))*
==> $\sim \text{is-FEx } A \Rightarrow m = 0$
apply(*frule-tac* $n=n$ **in** *index0*)

```

apply(frule-tac is-path-f) apply(drule-tac x=n in spec) apply(case-tac f n)
apply(simp)
apply(simp add: contains-def)
apply(force)
done

```

```

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

```

```

apply(rule-tac nat-less-induct) apply(rule, rule) apply(case-tac A)
apply(rule,rule,rule) apply(simp add: mo Let-def) apply(simp add: model-def
Let-def del: map-map) apply(simp only: f[symmetric]) apply(force)

```

```

apply(rule,rule,rule) apply(simp add: mo Let-def) apply(simp add: model-def
Let-def del: map-map) apply(simp only: f[symmetric]) apply(rule ccontr) ap-
ply(simp del: map-map) apply(elim exE)
apply(subgoal-tac m = 0 & ma = 0) prefer 2 apply(simp del: map-map)
apply(simp del: map-map)
apply(rename-tac nat list m na nb ma)
apply(subgoal-tac ? y. considers f (nb+na+y) (0, PAtom nat list)) prefer 2
apply(rule contains-considers) apply(assumption)
apply(rule contains-propagates-patoms) apply(assumption) apply(assumption)
apply(erule exE)
apply(subgoal-tac contains f (na+nb+y) (0, NAtom nat list))
apply(subgoal-tac nb+na=na+nb)
apply(simp) apply(subgoal-tac is-axiom (s-of-ns (snd (f (na+nb+y))))))
apply(drule-tac is-axiom-finite-deriv) apply(force dest: is-path-f)
apply(simp add: contains-def considers-def) apply(case-tac snd (f (na +
nb + y))) apply(simp) apply(simp add: s-of-ns-def is-axiom.simps) apply(force)
apply(force)
apply(force intro: contains-propagates-natoms contains-propagates-patoms)

```

```

apply(intro impI allI)
apply(subgoal-tac m=0) prefer 2 apply(simp) apply(simp del: FEval.simps)
apply(frule-tac contains-propagates-fconj) apply(assumption)
apply(rename-tac form1 form2 m na)
apply(frule-tac x=size form1 in spec) apply(erule impE) apply(force)
apply(drule-tac x=form1 in spec) apply(drule-tac x=size form2 in spec) ap-
ply(erule impE) apply(force) apply(simp)
apply(force)

```

```

apply(intro impI allI)
apply(subgoal-tac m=0) prefer 2 apply(simp) apply(simp del: FEval.simps)
apply(frule-tac contains-propagates-fdisj) apply(assumption)
apply(rename-tac form1 form2 m na)
apply(frule-tac x=size form1 in spec) apply(erule impE) apply(force) ap-

```



```

ply(drule-tac x=form1 in spec) apply(drule-tac x=size form2 in spec) apply(erule
impE) apply(force) apply(simp)
  apply(force)

```

```

apply(intro impI allI)
apply(subgoal-tac m=0) prefer 2 apply(simp) apply(simp del: FEval.simps)
apply(frule-tac contains-propagates-fall) apply(assumption)
apply(erule exE) — all case
apply(rename-tac form m na y)
apply(drule-tac x=size form in spec) apply(erule impE, force) apply(drule-tac
x=first form (newvar (sfv (s-of-ns (snd (f (na + y)))))) in spec) apply(erule
impE, force)
  apply(erule impE, force) apply(simp add: FEval-first) apply(rule-tac x=ntou
(newvar (sfv (s-of-ns (snd (f (na + y)))))) in bexI) apply(assumption)
  using is-env-model-ntou[of s] apply(simp add: is-env-def mo)

```

```

apply(intro impI allI) apply(simp del: FEval.simps)
apply(frule-tac FEx-upward) apply(assumption) apply(assumption)
apply(simp)
apply(rule)
apply(rename-tac form m na ma)
apply(subgoal-tac  $\forall m'. \sim FEval\ mo\ ntou\ (first\ form\ m')$ )
  prefer 2 apply(rule)
  apply(drule-tac x=size form in spec) apply(erule impE, force)
  apply(drule-tac x=first form m' in spec) apply(erule impE, force) apply(erule
impE) apply(force) apply(simp add: id-def)
  apply(simp add: FEval-first id-def)
  apply(drule-tac x=uton ma and  $P=\%m'. \sim (P\ m')$  for P in spec)
  apply(subgoal-tac ma  $\in\ range\ ntou$ ) apply(frule-tac ntou-uton) apply(simp)
  apply(simp add: mo model-def)
done

```

```

lemma (in loc2) model: infinite (deriv s) ==> init s ==> ( $\forall A\ m\ n.$  contains f
n (m,A) -->  $\sim (FEval\ mo\ ntou\ A)$ )
  apply(rule)
  apply(frule-tac model') apply(auto) done

```

2.9 Completeness

```

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

```

```

apply(frule-tac model) apply(assumption)
apply(rule)
apply(case-tac mA)
  apply(drule-tac x=b in spec) apply(drule-tac x=0 in spec) apply(drule-tac
x=0 in spec) apply(erule impE)
  apply(simp add: contains-def) apply(frule-tac is-path-f-0) apply(simp)
  apply(subgoal-tac a=0)

```

```

prefer 2 apply(simp only: init-def, force)
apply auto
done — FIXME very ugly

```

```

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

```

```

lemma completeness': infinite (deriv s) ==> init s ==>  $\forall mA \in set s. \sim FEval$ 
(model s) ntou (snd mA)
apply(rule loc2.completeness'[simplified loc2-def loc2-axioms-def loc1-def])
apply(auto) done

```

```

lemma completeness'': infinite (deriv (ns-of-s s)) ==> init (ns-of-s s) ==>  $\forall A.$ 
A  $\in set s \dashrightarrow \sim FEval (model (ns-of-s s)) ntou A$ 
apply(frule-tac completeness') apply(assumption)
apply(simp add: ns-of-s-def)
done

```

```

lemma completeness: infinite (deriv (ns-of-s s)) ==>  $\sim Svalid s$ 
apply(subgoal-tac init (ns-of-s s))
prefer 2 apply(simp add: init-def ns-of-s-def)
apply(frule-tac completeness'') apply(assumption)
apply(simp add: Svalid-def)
apply(simp add: SEval-def2)
apply(rule-tac x=fst (model (ns-of-s s)) in exI)
apply(rule-tac x=snd (model (ns-of-s s)) in exI)
apply(rule-tac x=ntou in exI)
apply(simp add: model-def)
apply(simp add: is-env-def)
done

```

— FIXME silly splitting of quantified pairs

2.10 Sound and Complete

```

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

```

2.11 Algorithm

```

primrec iter :: ('a => 'a) => 'a => nat => 'a — fold for nats
where

```

```

  iter g a 0 = a
| iter g a (Suc n) = g (iter g a n)

```

```

lemma iter:  $\forall a. (iter\ g\ (g\ a)\ n) = (g\ (iter\ g\ a\ n))$ 
apply(induct n)
apply(simp)
apply(simp)
done

```

```

lemma ex-iter':  $(\exists n. R\ (iter\ g\ a\ n)) = (R\ a\ | (\exists n. R\ (iter\ g\ (g\ a)\ n)))$ 

```

```

apply(rule)
apply(erule exE)
apply(case-tac n)
  apply(simp)
apply(rule disjI2)
apply(rule-tac x=nat in exI)
apply(simp add: iter)
apply(erule disjE)
apply(rule-tac x=0 in exI, simp)
apply(erule exE) apply(rule-tac x=Suc n in exI)
apply(simp add: iter)
done

```

— version suitable for computation

```

lemma ex-iter: ( $\exists n. R (iter\ g\ a\ n) = (if\ R\ a\ then\ True\ else\ (\exists n. R (iter\ g\ (g\ a)\ n))$ )
apply (rule trans)
apply (rule ex-iter^)
apply (force )
done

```

definition

```

f :: nseq list => nat => nseq list where
f s n = iter (% x. flatten (map subs x)) s n

```

```

lemma f-upwards: f s n = [] ==> f s (n+m) = []
apply(induct-tac m) apply(simp)
apply(simp add: f-def)
done

```

```

lemma flatten-append: flatten (xs@ys) = ((flatten xs) @ (flatten ys))
apply(induct xs) by auto

```

```

lemma set-flatten: set (flatten xs) = Union (set ` set xs)
apply(induct xs) apply(simp)
apply(simp)
done

```

```

lemma f:  $\forall x. ((n,x) \in deriv\ s) = (x \in set\ (f\ [s]\ n))$ 
apply(induct n)
apply(simp) apply(simp add: f-def)
apply(rule)
apply(rule)
apply(drule-tac deriv-downwards)
apply(elim exE conjE)
apply(drule-tac x=y in spec)
apply(simp)
apply(drule-tac list-decomp) apply(elim exE conjE)
apply(simp add: flatten-append f-def Let-def)

```

```

apply(simp add: f-def)
apply(simp add: set-flatten)
apply(erule bexE)
apply(drule-tac x=xa in spec)
apply(rule step) apply(auto)
done

```

```

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

```

```

lemma finite-f: finite (set (f s x))
by (fact finite-set)

```

```

lemma finite-deriv: finite (deriv s) = ( $\exists m. f [s] m = []$ )
apply(rule)
apply(subgoal-tac finite (fst ' (deriv s))) prefer 2 apply(simp)
apply(frule-tac max-exists) apply(erule impE) apply(simp) apply(subgoal-tac
( $0, s \in \text{deriv } s$ ) apply(force) apply(simp)
apply(elim exE conjE)
apply(rule-tac x=Suc x in exI)
apply(simp)
apply(rule ccontr) apply(case-tac f [s] (Suc x)) apply(simp)
apply(subgoal-tac (Suc x, a)  $\in \text{deriv } s$ ) apply(force)
apply(simp add: f)
apply(erule exE)
apply(subgoal-tac  $\forall y. f [s] (m+y) = []$ )
prefer 2 apply(rule) apply(rule f-upwards) apply(assumption)
apply(simp add: deriv-f)
apply(subgoal-tac (UNIV::nat set) = {y. y < m} Un {y. m  $\leq$  y})
prefer 2 apply force
apply(erule-tac t=UNIV::nat set in ssubst)
apply(simp)
apply(subgoal-tac (UN x:Collect (( $\leq$ ) m). Pair x ' set (f [s] x)) = (UN x:Collect
(( $\leq$ ) m). {})) apply(simp only:) apply(force)
apply(rule SUP-cong) apply(force) apply(drule-tac x=x-m in spec) apply(force)
done

```

```

lemma ex-iter-fSucn: ( $\exists m. \text{iter } (\% x. \text{flat } (\text{map } \text{subs } x)) l m = []$ ) = (if l = []
then True else ( $\exists n. (\text{iter } (\% x. \text{flat } (\text{map } \text{subs } x)) ((\% x. \text{flat } (\text{map } \text{subs } x)) l) n) = []$ ))
using ex-iter[of  $\% x. x = []$ , of  $(\% x. \text{flat } (\text{map } \text{subs } x)) l$ ] apply(force) done

```

```

definition prove' :: nseq list => bool where
  prove' s = ( $\exists m. \text{iter } (\% x. \text{flatten } (\text{map } \text{subs } x)) s m = []$ )

```

```

lemma prove': prove' l = (if l = [] then True else prove' (( $\% x. \text{flatten } (\text{map } \text{subs } x)$ ) l))
apply(simp only: prove'-def)
apply(rule ex-iter-fSucn)

```

done

— this is the main claim for efficiency- we have a tail recursive implementation via this lemma

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

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

2.12 Computation

— a sample formula to prove

lemma $(\exists x. A\ x \mid B\ x) \dashrightarrow ((\exists x. B\ x) \mid (\exists x. A\ x))$ **by** *blast*

— convert to our form

lemma $((\exists x. A\ x \mid B\ x) \dashrightarrow ((\exists x. B\ x) \mid (\exists x. A\ x)))$
 $= ((\forall x. \sim A\ x \ \& \ \sim B\ x) \mid ((\exists x. B\ x) \mid (\exists x. A\ x)))$ **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* *flatten.simps* *list.map*
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*
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*)])

apply(*simp* *add*: *my-f-def* *finite-deriv-prove* *prove-def*)

apply(*simp* *only*: *prove'-Nil* *prove'-Cons* *ss*)

done

abbreviation *Sprove* :: *form* *list* => *bool* **where** *Sprove* \equiv *prove* *o* *ns-of-s*

abbreviation *check* :: *form* => *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 *flatten* [] = []
| *flatten* (*a::list*) = *a* @ (*flatten 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 *first* *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* = *flatten* (*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 = flatten (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.