

Completeness for FOL

James Margetson, ported by Tom Ridge

April 29, 2009

Contents

1	Permutation Lemmas	3
1.1	perm, count equivalence	3
1.2	Properties closed under Perm and Contr hold for x iff hold for remdups x	4
1.3	List properties closed under Perm, Weak and Contr are mono- tonic in the set of the list	4
1.4	Following used in Soundness	5
2	Base	5
2.1	Integrate with Isabelle libraries?	5
2.2	Summation	5
2.3	Termination Measure	6
2.4	Functions	6
3	Formula	7
3.1	Variables	7
3.2	Predicates	9
3.3	Formulas	9
3.4	formula signs induct, formula signs cases	9
3.5	Frees	11
3.6	Substitutions	12
3.7	Models	12
3.8	model, non empty set and positive atom valuation	12
3.9	Validity	13
4	Sequents	14
4.1	Rules	15
4.2	Deductions	15
4.3	Basic Rule sets	15
4.4	Derived Rules	17
4.5	Standard Rule Sets For Predicate Calculus	18
4.6	Monotonicity for CutFreePC deductions	18

4.7	Tree	19
4.8	Terminal	20
4.9	Inherited	20
4.10	bounded, boundedBy	22
4.11	Inherited Properties- bounded	23
4.12	founded	23
4.13	Inherited Properties- founded	23
4.14	Inherited Properties- finite	24
4.15	path: follows a failing inherited property through tree	24
4.16	Branch	25
4.17	failing branch property: abstracts path defn	26
4.18	Tree induction principles	26
5	Completeness	27
5.1	pseq: type represents a processed sequent	27
5.2	subs: SATAxiom	27
5.3	subs: a CutFreePC justifiable backwards proof step	28
5.4	proofTree(Gamma) says whether tree(Gamma) is a proof	28
5.5	path: considers, contains, costBarrier	28
5.6	path: eventually	29
5.7	path: counter model	29
5.8	subs: finite	29
5.9	inherited: proofTree	30
5.10	pseq: lemma	30
5.11	SATAxiom: proofTree	30
5.12	SATAxioms are deductions: - needed	31
5.13	proofTrees are deductions: subs respects rules - messy start and end	31
5.14	proofTrees are deductions: instance of boundedTreeInduction	31
5.15	contains, considers:	31
5.16	path: nforms = [] implies	32
5.17	path: cases	32
5.18	path: contains not terminal and propagate condition	32
5.19	path: no consider lemmas	32
5.20	path: contains initially	33
5.21	termination: (for EV contains implies EV considers)	33
5.22	costBarrier: lemmas	33
5.23	costBarrier: exp3 lemmas - bit specific...	33
5.24	costBarrier: decreases whilst contains and unconsiders	33
5.25	path: EV contains implies EV considers	34
5.26	EV contains: common lemma	34
5.27	EV contains: FConj,FDisj,FAI	34
5.28	EV contains: lemmas (temporal related)	35
5.29	EV contains: FAToms	35

5.30	EV contains: FEx cases	35
5.31	pseq: creates initial pseq	36
5.32	EV contains: contain any (i,FEx y) means contain all (i,FEx y)	36
5.33	EV contains: contain any (i,FEx y) means contain all (i,FEx y)	36
5.34	EV contains: atoms	37
5.35	counterModel: lemmas	37
5.36	counterModel: all path formula value false - step by step . . .	38
5.37	adequacy	38

6 Soundness 38

1 Permutation Lemmas

```
theory PermutationLemmas
imports Permutation Multiset
begin
```

— following function is very close to that in multisets- now we can make the connection that $x \dot{=} y$ iff the multiset of x is the same as that of y

1.1 perm, count equivalence

```
primrec count :: 'a ⇒ 'a list ⇒ nat
```

```
where
```

```
  count x [] = 0
```

```
| count x (y#ys) = (if x=y then 1 else 0) + count x ys
```

```
lemma perm-count: A <~~> B ⇒ (∀ x. count x A = count x B)
  <proof>
```

```
lemma count-0: (∀ x. count x B = 0) = (B = [])
  <proof>
```

```
lemma count-Suc: count a B = Suc m ⇒ a : set B
  <proof>
```

```
lemma count-append: count a (xs@ys) = count a xs + count a ys
  <proof>
```

```
lemma count-perm: !! B. (∀ x. count x A = count x B) ⇒ A <~~> B
  <proof>
```

```
lemma perm-count-conv: A <~~> B = (∀ x. count x A = count x B)
  <proof>
```

1.2 Properties closed under Perm and Contr hold for x iff hold for remdups x

lemma *remdups-append*: $y : \text{set } ys \dashrightarrow \text{remdups } (ws@y\#ys) = \text{remdups } (ws@ys)$
 ⟨proof⟩

lemma *perm-contr'*: **assumes** *perm*[rule-format]: $! xs \ ys. xs <\sim\sim> ys \dashrightarrow (P xs = P ys)$

and *contr'*[rule-format]: $! x \ xs. P(x\#x\#xs) = P(x\#xs)$
shows $! xs. \text{length } xs = n \dashrightarrow (P xs = P(\text{remdups } xs))$
 ⟨proof⟩

lemma *perm-contr*: **assumes** *perm*: $! xs \ ys. xs <\sim\sim> ys \dashrightarrow (P xs = P ys)$

and *contr'*: $! x \ xs. P(x\#x\#xs) = P(x\#xs)$
shows $(P xs = P(\text{remdups } xs))$
 ⟨proof⟩

1.3 List properties closed under Perm, Weak and Contr are monotonic in the set of the list

definition

rem :: $'a \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list}$ **where**
rem $x \ xs = \text{filter } (\%y. y \sim = x) \ xs$

lemma *rem*: $x \sim : \text{set } (rem \ x \ xs)$
 ⟨proof⟩

lemma *length-rem*: $\text{length } (rem \ x \ xs) \leq \text{length } xs$
 ⟨proof⟩

lemma *rem-notin*: $x \ \sim : \text{set } xs \implies rem \ x \ xs = xs$
 ⟨proof⟩

lemma *perm-weak-filter'*: **assumes** *perm*[rule-format]: $! xs \ ys. xs <\sim\sim> ys \dashrightarrow (P xs = P ys)$

and *weak*[rule-format]: $! x \ xs. P \ xs \dashrightarrow P(x\#xs)$
shows $! ys. P(ys@filter \ Q \ xs) \dashrightarrow P(ys@xs)$
 ⟨proof⟩

lemma *perm-weak-filter*: **assumes** *perm*: $! xs \ ys. xs <\sim\sim> ys \dashrightarrow (P xs = P ys)$

and *weak*: $! x \ xs. P \ xs \dashrightarrow P(x\#xs)$
shows $P(\text{filter } Q \ xs) \implies P \ xs$
 ⟨proof⟩

lemma *perm-weak-contr-mono*:

assumes *perm*: $! xs \ ys. xs <\sim\sim> ys \dashrightarrow (P xs = P ys)$
and *contr*: $! x \ xs. P(x\#x\#xs) \dashrightarrow P(x\#xs)$
and *weak*: $! x \ xs. P \ xs \dashrightarrow P(x\#xs)$

```

and  $xy$ : set  $x \leq$  set  $y$ 
and  $Px$  :  $P x$ 
shows  $P y$ 
<proof>

```

1.4 Following used in Soundness

```

primrec multiset-of-list :: 'a list  $\Rightarrow$  'a multiset
where

```

```

  multiset-of-list [] = {#}
| multiset-of-list (x#xs) = {#x#} + multiset-of-list xs

```

```

lemma count-count[symmetric]: count  $x A =$  Multiset.count (multiset-of-list  $A$ )  $x$ 
<proof>

```

```

lemma perm-multiset:  $A <\sim\sim> B =$  (multiset-of-list  $A =$  multiset-of-list  $B$ )
<proof>

```

```

lemma set-of-multiset-of-list: set-of (multiset-of-list  $A$ ) = set  $A$ 
<proof>

```

```

end

```

2 Base

```

theory Base
imports PermutationLemmas
begin

```

2.1 Integrate with Isabelle libraries?

— Misc

- FIXME added by tjr, forms basis of a lot of proofs of existence of inf sets
- something like this should be in FiniteSet, asserting nats are not finite

```

lemma natset-finite-max: assumes  $a$ : finite  $A$ 

```

```

  shows Suc (Max  $A$ )  $\notin A$ 

```

```

<proof>

```

```

lemma not-finite-univ:  $\sim$  finite (UNIV::nat set)

```

```

<proof>

```

```

lemma LeastI-ex:  $(\exists x. P (x::'a::wellorder)) \implies P (LEAST x. P x)$ 

```

```

<proof>

```

2.2 Summation

```

primrec summation :: (nat  $\Rightarrow$  nat)  $\Rightarrow$  nat  $\Rightarrow$  nat

```

```

where

```

```

  summation  $f$  0 =  $f$  0

```

| $\text{summation } f \text{ (Suc } n) = f \text{ (Suc } n) + \text{summation } f \text{ } n$

2.3 Termination Measure

primrec $\text{exp} :: [\text{nat}, \text{nat}] \Rightarrow \text{nat}$

where

$$\text{exp } x \ 0 = 1$$

| $\text{exp } x \text{ (Suc } m) = x * \text{exp } x \ m$

primrec $\text{sumList} :: \text{nat list} \Rightarrow \text{nat}$

where

$$\text{sumList } [] = 0$$

| $\text{sumList } (x \# xs) = x + \text{sumList } xs$

2.4 Functions

definition

$\text{preImage} :: ('a \Rightarrow 'b) \Rightarrow 'b \text{ set} \Rightarrow 'a \text{ set}$ **where**

$$\text{preImage } f \ A = \{ x . f \ x \in A \}$$

definition

$\text{pre} :: ('a \Rightarrow 'b) \Rightarrow 'b \Rightarrow 'a \text{ set}$ **where**

$$\text{pre } f \ a = \{ x . f \ x = a \}$$

definition

$\text{equalOn} :: ['a \text{ set}, 'a \Rightarrow 'b, 'a \Rightarrow 'b] \Rightarrow \text{bool}$ **where**

$$\text{equalOn } A \ f \ g = (!x:A. f \ x = g \ x)$$

lemma preImage-insert : $\text{preImage } f \ (\text{insert } a \ A) = \text{pre } f \ a \ \text{Un } \text{preImage } f \ A$

$\langle \text{proof} \rangle$

lemma preImageI : $f \ x : A \ ==> x : \text{preImage } f \ A$

$\langle \text{proof} \rangle$

lemma preImageE : $x : \text{preImage } f \ A \ ==> f \ x : A$

$\langle \text{proof} \rangle$

lemma equalOn-Un : $\text{equalOn } (A \cup B) \ f \ g = (\text{equalOn } A \ f \ g \ \wedge \ \text{equalOn } B \ f \ g)$

$\langle \text{proof} \rangle$

lemma equalOnD : $\text{equalOn } A \ f \ g \ ==> (\forall x \in A . f \ x = g \ x)$

$\langle \text{proof} \rangle$

lemma equalOnI : $(\forall x \in A . f \ x = g \ x) \ ==> \text{equalOn } A \ f \ g$

$\langle \text{proof} \rangle$

lemma equalOn-UnD : $\text{equalOn } (A \ \text{Un } B) \ f \ g \ ==> \text{equalOn } A \ f \ g \ \& \ \text{equalOn } B \ f \ g$

$\langle \text{proof} \rangle$

lemma $\text{inj-inv-singleton[simp]}$: $[\text{inj } f; f \ z = y] \ ==> \{x. f \ x = y\} = \{z\}$

<proof>

lemma *finite-pre[simp]*: $\text{inj } f \implies \text{finite } (\text{pre } f \ x)$
<proof>

lemma *finite-preImage[simp]*: $\llbracket \text{finite } A; \text{inj } f \rrbracket \implies \text{finite } (\text{preImage } f \ A)$
<proof>

end

3 Formula

theory *Formula*
imports *Base*
begin

3.1 Variables

datatype *vbl = X nat*

— FIXME there's a lot of stuff about this datatype that is really just a lifting from *nat* (what else could it be). Makes me wonder whether things wouldn't be clearer if we just identified *vbls* with *nats*

primrec *deX* :: *vbl => nat* **where** $\text{deX } (X \ n) = n$

lemma *X-deX[simp]*: $X \ (\text{deX } a) = a$
<proof>

definition *zeroX* = $X \ 0$

consts *nextX* :: *vbl => vbl*

primrec *nextX* $(X \ n) = X \ (\text{Suc } n)$

definition

vblcase :: $['a, \text{vbl} \Rightarrow 'a, \text{vbl}] \Rightarrow 'a$ **where**

$\text{vblcase } a \ f \ n = (@z. (n = \text{zeroX} \longrightarrow z = a) \wedge (!x. n = \text{nextX } x \longrightarrow z = f(x)))$

translations

$\text{case } p \ \text{of } X\text{CONST } \text{zeroX} \Rightarrow a \mid X\text{CONST } \text{nextX } y \Rightarrow b == (\text{CONST } \text{vblcase } a \ (\%y. b) \ p)$

definition

freshVar :: *vbl set => vbl* **where**

$\text{freshVar } vs = X \ (\text{LEAST } n. n \notin \text{deX } 'vs)$

lemma *nextX-nextX[iff]*: $\text{nextX } x = \text{nextX } y = (x = y)$
<proof>

lemma *inj-nextX*: *inj nextX*

<proof>

lemma *ind'*: $P \text{ zeroX} \implies (! v . P v \dashrightarrow P (\text{nextX } v)) \implies P v'$

<proof>

lemma *ind*: $\llbracket P \text{ zeroX}; \bigwedge v . P v \implies P (\text{nextX } v) \rrbracket \implies P v'$

<proof>

lemma *zeroX-nextX*[*iff*]: $\text{zeroX} \sim = \text{nextX } a$ — FIXME iff?

<proof>

lemmas *nextX-zeroX*[*iff*] = *not-sym*[*OF zeroX-nextX*]

lemma *nextX*: $\text{nextX } (X \ n) = X \ (\text{Suc } n)$

<proof>

lemma *vblcase-zeroX*[*simp*]: $\text{vblcase } a \ b \ \text{zeroX} = a$

<proof>

lemma *vblcase-nextX*[*simp*]: $\text{vblcase } a \ b \ (\text{nextX } n) = b \ n$

<proof>

lemma *vbl-cases*: $x = \text{zeroX} \mid (? y . x = \text{nextX } y)$

<proof>

lemma *vbl-casesE*: $\llbracket x = \text{zeroX} \implies P; \bigwedge y . x = \text{nextX } y \implies P \rrbracket \implies P$

<proof>

lemma *comp-vblcase*: $f \circ \text{vblcase } a \ b = \text{vblcase } (f \ a) \ (f \circ b)$

<proof>

lemma *equalOn-vblcaseI'*: $\text{equalOn } (\text{preImage } \text{nextX } A) \ f \ g \implies \text{equalOn } A \ (\text{vblcase } x \ f) \ (\text{vblcase } x \ g)$

<proof>

lemma *equalOn-vblcaseI*: $(\text{zeroX} : A \dashrightarrow x=y) \implies \text{equalOn } (\text{preImage } \text{nextX } A) \ f \ g \implies \text{equalOn } A \ (\text{vblcase } x \ f) \ (\text{vblcase } y \ g)$

<proof>

lemma *X-deX-connection*: $X \ n : A = (n : (\text{deX } ' A))$

<proof>

lemma *finiteFreshVar*: $\text{finite } A \implies \text{freshVar } A \sim : A$

<proof>

lemma *freshVarI*: $\llbracket \text{finite } A; B \leq A \rrbracket \implies \text{freshVar } A \sim : B$

<proof>

lemma *freshVarI2*: $finite\ A \implies !x . x \sim : A \dashrightarrow P\ x \implies P\ (freshVar\ A)$
 ⟨*proof*⟩

lemmas *vblsimps* = *vblcase-zeroX vblcase-nextX zeroX-nextX*
nextX-zeroX nextX-nextX comp-vblcase

3.2 Predicates

datatype *predicate* = *Predicate nat*

datatype *signs* = *Pos* | *Neg*

lemma *signsE*: $\llbracket signs = Neg \implies P; signs = Pos \implies P \rrbracket \implies P$
 ⟨*proof*⟩

lemma *expand-signs-case*: $Q(signs\text{-}case\ vpos\ vneg\ F) = ($
 $(F = Pos \dashrightarrow Q\ (vpos)) \ \&$
 $(F = Neg \dashrightarrow Q\ (vneg))$
 $)$
 ⟨*proof*⟩

primrec *sign* :: *signs* \Rightarrow *bool* \Rightarrow *bool*

where

sign Pos x = *x*
 | *sign Neg x* = $(\neg\ x)$

lemma *sign-arg-cong*: $x = y \implies sign\ z\ x = sign\ z\ y$ ⟨*proof*⟩

primrec *invSign* :: *signs* \Rightarrow *signs*

where

invSign Pos = *Neg*
 | *invSign Neg* = *Pos*

3.3 Formulas

datatype *formula* =
FAtom signs predicate (vbl list)
 | *FConj signs formula formula*
 | *FAll signs formula*

3.4 formula signs induct, formula signs cases

lemma *formula-signs-induct*: \llbracket
 ! *p vs*. $P\ (FAtom\ Pos\ p\ vs)$;
 ! *p vs*. $P\ (FAtom\ Neg\ p\ vs)$;
 !! *A B* . $\llbracket P\ A; P\ B \rrbracket \implies P\ (FConj\ Pos\ A\ B)$;
 !! *A B* . $\llbracket P\ A; P\ B \rrbracket \implies P\ (FConj\ Neg\ A\ B)$;
 !! *A* . $\llbracket P\ A \rrbracket \implies P\ (FAll\ Pos\ A)$;
 !! *A* . $\llbracket P\ A \rrbracket \implies P\ (FAll\ Neg\ A)$
 \rrbracket

\llbracket
 $\implies P A$
 $\langle \text{proof} \rangle$

lemma *formula-signs-cases*: $\llbracket P$.
 $\llbracket \llbracket p \text{ vs} . P (\text{FAtom Pos } p \text{ vs});$
 $\llbracket p \text{ vs} . P (\text{FAtom Neg } p \text{ vs});$
 $\llbracket f1 \text{ f2} . P (\text{FConj Pos } f1 \text{ f2});$
 $\llbracket f1 \text{ f2} . P (\text{FConj Neg } f1 \text{ f2});$
 $\llbracket f1 . P (\text{FAll Pos } f1);$
 $\llbracket f1 . P (\text{FAll Neg } f1) \llbracket$
 $\implies P A$
 $\langle \text{proof} \rangle$

lemma *strong-formula-induct'*: $\llbracket A. (\llbracket B. \text{size } B < \text{size } A \dashrightarrow P B) \dashrightarrow P A$
 $\implies \llbracket A. \text{size } A = n \dashrightarrow P (A::\text{formula})$
 $\langle \text{proof} \rangle$

lemma *strong-formula-induct*: $\llbracket A. (\llbracket B. \text{size } B < \text{size } A \dashrightarrow P B) \dashrightarrow P A$
 $\implies P (A::\text{formula})$
 $\langle \text{proof} \rangle$

lemma *sizelemmas*: $\text{size } A < \text{size } (\text{FConj } z \text{ A } B)$
 $\text{size } B < \text{size } (\text{FConj } z \text{ A } B)$
 $\text{size } A < \text{size } (\text{FAll } z \text{ A})$
 $\langle \text{proof} \rangle$

lemma *expand-formula-case*:
 $Q(\text{formula-case } \text{fatom } \text{fconj } \text{fall } F) = ($\llbracket z \text{ P vs} . F = \text{FAtom } z \text{ P vs} \dashrightarrow Q (\text{fatom } z \text{ P vs}) \&$
 $\llbracket z \text{ A0 A1} . F = \text{FConj } z \text{ A0 A1} \dashrightarrow Q (\text{fconj } z \text{ A0 A1}) \&$
 $\llbracket z \text{ A} . F = \text{FAll } z \text{ A} \dashrightarrow Q (\text{fall } z \text{ A})$
 $)$
 $\langle \text{proof} \rangle$$

primrec *FNot* :: *formula* \implies *formula*

where

$\text{FNot-FAtom}: \text{FNot } (\text{FAtom } z \text{ P vs}) = \text{FAtom } (\text{invSign } z) \text{ P vs}$
 $\text{FNot-FConj}: \text{FNot } (\text{FConj } z \text{ A0 A1}) = \text{FConj } (\text{invSign } z) (\text{FNot } \text{A0}) (\text{FNot } \text{A1})$
 $\text{FNot-FAll}: \text{FNot } (\text{FAll } z \text{ body}) = \text{FAll } (\text{invSign } z) (\text{FNot } \text{body})$

primrec *neg* :: *signs* \implies *signs*

where

$\text{neg Pos} = \text{Neg}$
 $\text{neg Neg} = \text{Pos}$

primrec

$\text{dual} :: [(\text{signs} \implies \text{signs}), (\text{signs} \implies \text{signs}), (\text{signs} \implies \text{signs})] \implies \text{formula} \implies$
formula

where

$dual\text{-}FAtom: dual\ p\ q\ r\ (FAtom\ z\ P\ vs) = FAtom\ (p\ z)\ P\ vs$
| $dual\text{-}FConj: dual\ p\ q\ r\ (FConj\ z\ A0\ A1) = FConj\ (q\ z)\ (dual\ p\ q\ r\ A0)\ (dual\ p\ q\ r\ A1)$
| $dual\text{-}FAll: dual\ p\ q\ r\ (FAll\ z\ body) = FAll\ (r\ z)\ (dual\ p\ q\ r\ body)$

lemma $dualCompose: dual\ p\ q\ r\ o\ dual\ P\ Q\ R = dual\ (p\ o\ P)\ (q\ o\ Q)\ (r\ o\ R)$
 $\langle proof \rangle$

lemma $dualFNot': dual\ invSign\ invSign\ invSign = FNot$
 $\langle proof \rangle$

lemma $dualFNot: dual\ invSign\ id\ id\ (FNot\ A) = FNot\ (dual\ invSign\ id\ id\ A)$
 $\langle proof \rangle$

lemma $dualId: dual\ id\ id\ id\ A = A$
 $\langle proof \rangle$

3.5 Frees

primrec $freeVarsF :: formula \Rightarrow vbl\ set$

where

$freeVarsFAtom: freeVarsF\ (FAtom\ z\ P\ vs) = set\ vs$
| $freeVarsFConj: freeVarsF\ (FConj\ z\ A0\ A1) = (freeVarsF\ A0)\ Un\ (freeVarsF\ A1)$
| $freeVarsFAll: freeVarsF\ (FAll\ z\ body) = preImage\ nextX\ (freeVarsF\ body)$

definition

$freeVarsFL :: formula\ list \Rightarrow vbl\ set$ **where**
 $freeVarsFL\ gamma = Union\ (freeVarsF\ ` (set\ gamma))$

lemma $freeVarsF\text{-}FNot[simp]: freeVarsF\ (FNot\ A) = freeVarsF\ A$
 $\langle proof \rangle$

lemma $finite\text{-}freeVarsF[simp]: finite\ (freeVarsF\ A)$
 $\langle proof \rangle$

lemma $freeVarsFL\text{-}nil[simp]: freeVarsFL\ ([]) = \{\}$
 $\langle proof \rangle$

lemma $freeVarsFL\text{-}cons: freeVarsFL\ (A\#\Gamma) = freeVarsF\ A \cup freeVarsFL\ \Gamma$
 $\langle proof \rangle$

lemma $finite\text{-}freeVarsFL[simp]: finite\ (freeVarsFL\ gamma)$
 $\langle proof \rangle$

lemma $freeVarsDual: freeVarsF\ (dual\ p\ q\ r\ A) = freeVarsF\ A$
 $\langle proof \rangle$

3.6 Substitutions

primrec *subF* :: [*vbl* => *vbl,formula*] => *formula*

where

subFAtom: *subF theta (FAtom z P vs) = FAtom z P (map theta vs)*
| *subFConj*: *subF theta (FConj z A0 A1) = FConj z (subF theta A0) (subF theta A1)*
| *subFAll*: *subF theta (FAll z body) =*
FAll z (subF (% v . (case v of zeroX => zeroX | nextX v => nextX (theta v)))
body)

lemma *size-subF*: *!!theta. size (subF theta A) = size (A::formula)*
<proof>

lemma *subFNot*: *!!theta. subF theta (FNot A) = FNot (subF theta A)*
<proof>

lemma *subFDual*: *!!theta. subF theta (dual p q r A) = dual p q r (subF theta A)*
<proof>

definition

instanceF :: [*vbl,formula*] => *formula* **where**
instanceF w body = subF (%v. case v of zeroX => w | nextX v => v) body

lemma *size-instance*: *!!v. size (instanceF v A) = size (A::formula)*
<proof>

lemma *instanceFDual*: *instanceF u (dual p q r A) = dual p q r (instanceF u A)*
<proof>

3.7 Models

typedecl

object

axiomatization *obj* :: *nat* => *object*

where *inj-obj*: *inj obj*

3.8 model, non empty set and positive atom valuation

typedef *model* = { *z* :: (*object set* * ([*predicate,object list*] => *bool*)) . (*fst z* ~={}) } *<proof>*

definition

objects :: *model* => *object set* **where**
objects M = fst (Rep-model M)

definition

evalP :: *model* => *predicate* => *object list* => *bool* **where**
evalP M = snd (Rep-model M)

lemma *evalP-arg2-cong*: $x = y \implies \text{evalP } M \text{ p } x = \text{evalP } M \text{ p } y$ $\langle \text{proof} \rangle$

lemma *objectsNonEmpty*: $\text{objects } M \neq \{\}$
 $\langle \text{proof} \rangle$

lemma *modelsNonEmptyI*: $\text{fst } (\text{Rep-model } M) \neq \{\}$
 $\langle \text{proof} \rangle$

3.9 Validity

primrec *evalF* :: $[\text{model}, \text{vbl} \implies \text{object}, \text{formula}] \implies \text{bool}$

where

evalFAtom: $\text{evalF } M \text{ phi } (\text{FAtom } z \text{ P } vs) = \text{sign } z (\text{evalP } M \text{ P } (\text{map } \text{phi } vs))$
| *evalFConj*: $\text{evalF } M \text{ phi } (\text{FConj } z \text{ A0 } \text{ A1}) = \text{sign } z (\text{sign } z (\text{evalF } M \text{ phi } \text{ A0}) \ \& \ \text{sign } z (\text{evalF } M \text{ phi } \text{ A1}))$
| *evalFAll*: $\text{evalF } M \text{ phi } (\text{FAll } z \text{ body}) = \text{sign } z (!x: (\text{objects } M)).$
 $\text{sign } z$
 $(\text{evalF } M \ (\%v \ . \ (\text{case } v \ \text{of}$
 $\text{zeroX} \ \implies \ x$
| $\text{nextX } v \ \implies \ \text{phi } v)) \ \text{body}))$

definition

valid :: $\text{formula} \implies \text{bool}$ **where**
 $\text{valid } F \longleftrightarrow (\forall M \ \text{phi}. \ \text{evalF } M \ \text{phi } F = \text{True})$

lemma *evalF-FAll*: $\text{evalF } M \ \text{phi } (\text{FAll } \text{Pos } A) = (!x: (\text{objects } M)). (\text{evalF } M \ (\text{vblcase } x \ (\%v \ . \ \text{phi } v)) \ A))$
 $\langle \text{proof} \rangle$

lemma *evalF-FEx*: $\text{evalF } M \ \text{phi } (\text{FAll } \text{Neg } A) = (\ ? \ x: (\text{objects } M)). (\text{evalF } M \ (\text{vblcase } x \ (\%v \ . \ \text{phi } v)) \ A))$
 $\langle \text{proof} \rangle$

lemma *evalF-arg2-cong*: $x = y \implies \text{evalF } M \ \text{p } x = \text{evalF } M \ \text{p } y$ $\langle \text{proof} \rangle$

lemma *evalF-FNot*: $!\text{phi}. \ \text{evalF } M \ \text{phi } (\text{FNot } A) = (\neg \ \text{evalF } M \ \text{phi } A)$
 $\langle \text{proof} \rangle$

lemma *evalF-equiv[rule-format]*: $! \ f \ g. \ (\text{equalOn } (\text{freeVarsF } A) \ f \ g) \longrightarrow (\text{evalF } M \ f \ A = \text{evalF } M \ g \ A)$
 $\langle \text{proof} \rangle$

lemma *evalF-subF-eq*: $!\text{phi } \text{theta}. \ \text{evalF } M \ \text{phi } (\text{subF } \text{theta } A) = \text{evalF } M \ (\text{phi } \circ \ \text{theta}) \ A$
 $\langle \text{proof} \rangle$

lemma *o-id'[simp]*: $f \circ (\% \ x. \ x) = f$

<proof>

lemma *evalF-instance*: $evalF\ M\ \phi\ (instanceF\ u\ A) = evalF\ M\ (vblcase\ (\phi\ u))\ A$
<proof>

lemma *s[simp]*: $FConj\ signs\ formula1\ formula2 \neq formula1$
<proof>

lemma *s'[simp]*: $FConj\ signs\ formula1\ formula2 \neq formula2$
<proof>

lemma *instanceF-E*: $instanceF\ g\ formula \neq FAll\ signs\ formula$
<proof>

end

4 Sequents

theory *Sequents*
imports *Formula*
begin

types *sequent* = *formula list*

definition

evalS :: [*model*, *vbl* => *object*, *formula list*] => *bool* **where**
 $evalS\ M\ \phi\ fs \longleftrightarrow (\exists f : set\ fs . evalF\ M\ \phi\ f = True)$

lemma *evalS-nil[simp]*: $evalS\ M\ \phi\ [] = False$
<proof>

lemma *evalS-cons[simp]*: $evalS\ M\ \phi\ (A \# \Gamma) = (evalF\ M\ \phi\ A \mid evalS\ M\ \phi\ \Gamma)$
<proof>

lemma *evalS-append*: $evalS\ M\ \phi\ (\Gamma @ \Delta) = (evalS\ M\ \phi\ \Gamma \mid evalS\ M\ \phi\ \Delta)$
<proof>

lemma *evalS-equiv[rule-format]*: $(equalOn\ (freeVarsFL\ \Gamma)\ f\ g) \longrightarrow (evalS\ M\ f\ \Gamma = evalS\ M\ g\ \Gamma)$
<proof>

definition

modelAssigns :: [*model*] => (*vbl* => *object*) *set* **where**
 $modelAssigns\ M = \{ \phi . range\ \phi \leq objects\ M \}$

lemma *modelAssignsI*: $\text{range } f \leq \text{objects } M \implies f : \text{modelAssigns } M$
 ⟨*proof*⟩

lemma *modelAssignsD*: $f : \text{modelAssigns } M \implies \text{range } f \leq \text{objects } M$
 ⟨*proof*⟩

definition

validS :: *formula list* => *bool* **where**
validS *fs* $\longleftrightarrow (! M . ! \text{phi} : \text{modelAssigns } M . \text{evalS } M \text{ phi } \text{fs} = \text{True})$

4.1 Rules

types *rule* = *sequent* * (*sequent set*)

definition

concR :: *rule* => *sequent* **where**
concR = (%(*conc*,*prems*)). *conc*

definition

premsR :: *rule* => *sequent set* **where**
premsR = (%(*conc*,*prems*)). *prems*

definition

mapRule :: (*formula* => *formula*) => *rule* => *rule* **where**
mapRule = (%*f* (*conc*,*prems*) . (*map f conc*,(*map f*) ‘ *prems*))

lemma *mapRuleI*: $[[A = \text{map } f \ a; B = (\text{map } f) \ ' \ b \]] \implies (A,B) = \text{mapRule } f \ (a,b)$
 ⟨*proof*⟩

4.2 Deductions

inductive-set

deductions :: *rule set* => *formula list set*
for *rules* :: *rule set*

where

inferI: $[[(\text{conc},\text{prems}) : \text{rules};$
 prems : *Pow*(*deductions*(*rules*))
]] $\implies \text{conc} : \text{deductions}(\text{rules})$

monos *Pow-mono*

lemma *mono-deductions*: $[[A \leq B \]] \implies \text{deductions}(A) \leq \text{deductions}(B)$
 ⟨*proof*⟩

4.3 Basic Rule sets

definition

$Axioms = \{ z. ? p vs. \quad z = ([FAtom Pos p vs, FAtom Neg p vs], \{\}) \}$
definition
 $Conjs = \{ z. ? A0 A1 Delta Gamma. z = (FConj Pos A0 A1 \# Gamma @ Delta, \{A0 \# Gamma, A1 \# Delta\}) \}$
definition
 $Disjs = \{ z. ? A0 A1 \quad Gamma. z = (FConj Neg A0 A1 \# Gamma, \{A0 \# A1 \# Gamma\}) \}$
definition
 $Alls = \{ z. ? A x \quad Gamma. z = (FAll Pos A \# Gamma, \{instanceF x A \# Gamma\}) \ \& \ x \sim: freeVarsFL (FAll Pos A \# Gamma) \}$
definition
 $Exs = \{ z. ? A x \quad Gamma. z = (FAll Neg A \# Gamma, \{instanceF x A \# Gamma\}) \}$
definition
 $Weaks = \{ z. ? A \quad Gamma. z = (A \# Gamma, \{Gamma\}) \}$
definition
 $Contrs = \{ z. ? A \quad Gamma. z = (A \# Gamma, \{A \# A \# Gamma\}) \}$
definition
 $Cuts = \{ z. ? C Delta \quad Gamma. z = (Gamma @ Delta, \{C \# Gamma, FNot C \# Delta\}) \}$
definition
 $Perms = \{ z. ? Gamma Gamma' \quad . z = (Gamma, \{Gamma'\}) \ \& \ Gamma <\sim\sim> Gamma' \}$
definition
 $DAxioms = \{ z. ? p vs. \quad z = ([FAtom Neg p vs, FAtom Pos p vs], \{\}) \}$

lemma *AxiomI*: $[[Axioms <= A]] ==> [FAtom Pos p vs, FAtom Neg p vs] : deductions(A)$
 $\langle proof \rangle$

lemma *DAxiomsI*: $[[DAxioms <= A]] ==> [FAtom Neg p vs, FAtom Pos p vs] : deductions(A)$
 $\langle proof \rangle$

lemma *DisjI*: $[[A0 \# A1 \# Gamma : deductions(A); Disjs <= A]] ==> (FConj Neg A0 A1 \# Gamma) : deductions(A)$
 $\langle proof \rangle$

lemma *ConjI*: $[[(A0 \# Gamma) : deductions(A); (A1 \# Delta) : deductions(A); Conjs <= A]] ==> FConj Pos A0 A1 \# Gamma @ Delta : deductions(A)$
 $\langle proof \rangle$

lemma *AllI*: $[[instanceF w A \# Gamma : deductions(R); w \sim: freeVarsFL (FAll Pos A \# Gamma); Alls <= R]] ==> (FAll Pos A \# Gamma) : deductions(R)$
 $\langle proof \rangle$

lemma *ExI*: $[[instanceF w A \# Gamma : deductions(R); Exs <= R]] ==> (FAll Neg A \# Gamma) : deductions(R)$

$\langle \text{proof} \rangle$

lemma *WeakI*: $[[\text{Gamma} : \text{deductions } R; \text{Weaks} \leq R]] \implies A \# \text{Gamma} : \text{deductions}(R)$
 $\langle \text{proof} \rangle$

lemma *ContrI*: $[[A \# A \# \text{Gamma} : \text{deductions } R; \text{Contrs} \leq R]] \implies A \# \text{Gamma} : \text{deductions}(R)$
 $\langle \text{proof} \rangle$

lemma *PermI*: $[[\text{Gamma}' : \text{deductions } R; \text{Gamma} \sim \sim \text{Gamma}'; \text{Perms} \leq R]] \implies \text{Gamma} : \text{deductions}(R)$
 $\langle \text{proof} \rangle$

4.4 Derived Rules

lemma *WeakI1*: $[[\text{Gamma} : \text{deductions}(A); \text{Weaks} \leq A]] \implies (\text{Delta} @ \text{Gamma}) : \text{deductions}(A)$
 $\langle \text{proof} \rangle$

lemma *WeakI2*: $[[\text{Gamma} : \text{deductions}(A); \text{Perms} \leq A; \text{Weaks} \leq A]] \implies (\text{Gamma} @ \text{Delta}) : \text{deductions}(A)$
 $\langle \text{proof} \rangle$

lemma *SATAxiomI*: $[[\text{Axioms} \leq A; \text{Weaks} \leq A; \text{Perms} \leq A; \text{forms} = [\text{FAtom Pos } n \text{ vs, FAtom Neg } n \text{ vs}] @ \text{Gamma}]] \implies \text{forms} : \text{deductions}(A)$
 $\langle \text{proof} \rangle$

lemma *DisjI1*: $[[(A1 \# \text{Gamma}) : \text{deductions}(A); \text{Disjs} \leq A; \text{Weaks} \leq A]] \implies \text{FConj Neg } A0 \ A1 \# \text{Gamma} : \text{deductions}(A)$
 $\langle \text{proof} \rangle$

lemma *DisjI2*: $!!A. [[(A0 \# \text{Gamma}) : \text{deductions}(A); \text{Disjs} \leq A; \text{Weaks} \leq A; \text{Perms} \leq A]] \implies \text{FConj Neg } A0 \ A1 \# \text{Gamma} : \text{deductions}(A)$
 $\langle \text{proof} \rangle$

lemma *perm-tmp4*: $\text{Perms} \subseteq R \implies A @ (a \# \text{list}) @ (a \# \text{list}) : \text{deductions } R \implies (a \# a \# A) @ \text{list} @ \text{list} : \text{deductions } R$
 $\langle \text{proof} \rangle$

lemma *weaken-append[rule-format]*: $\text{Contrs} \leq R \implies \text{Perms} \leq R \implies !A. A @ \text{Gamma} @ \text{Gamma} : \text{deductions}(R) \dashrightarrow A @ \text{Gamma} : \text{deductions}(R)$
 $\langle \text{proof} \rangle$

lemma *ListWeakI*: $\text{Perms} \leq R \implies \text{Contrs} \leq R \implies x \# \text{Gamma} @ \text{Gamma} : \text{deductions}(R) \implies x \# \text{Gamma} : \text{deductions}(R)$
 $\langle \text{proof} \rangle$

lemma *ConjI'*: $[[(A0 \# \text{Gamma}) : \text{deductions}(A); (A1 \# \text{Gamma}) : \text{deductions}(A); \text{Contrs} \leq A; \text{Conjs} \leq A; \text{Perms} \leq A]] \implies \text{FConj Pos } A0 \ A1 \# \text{Gamma} :$

deductions(A)
<proof>

4.5 Standard Rule Sets For Predicate Calculus

definition

PC :: rule set **where**
PC = Union {*Perms*,*Axioms*,*Conjs*,*Disjs*,*Alls*,*Exs*,*Weaks*,*Contrs*,*Cuts*}

definition

CutFreePC :: rule set **where**
CutFreePC = Union {*Perms*,*Axioms*,*Conjs*,*Disjs*,*Alls*,*Exs*,*Weaks*,*Contrs*}

lemma *rulesInPCs*: *Axioms* <= *PC Axioms* <= *CutFreePC*

Conjs <= *PC Conjs* <= *CutFreePC*
Disjs <= *PC Disjs* <= *CutFreePC*
Alls <= *PC Alls* <= *CutFreePC*
Exs <= *PC Exs* <= *CutFreePC*
Weaks <= *PC Weaks* <= *CutFreePC*
Contrs <= *PC Contrs* <= *CutFreePC*
Perms <= *PC Perms* <= *CutFreePC*
Cuts <= *PC*
CutFreePC <= *PC*
<proof>

4.6 Monotonicity for CutFreePC deductions

- these lemmas can be used to replace complicated permutation reasoning above
- essentially if *x* is a deduction, and set *x* subset set *y*, then *y* is a deduction

definition

inDed :: formula list => bool **where**
inDed xs <=> *xs* : deductions *CutFreePC*

lemma *perm*: ! *xs ys*. *xs* <~> *ys* --> (*inDed xs* = *inDed ys*)
<proof>

lemma *contr*: ! *x xs*. *inDed (x#x#xs)* --> *inDed (x#xs)*
<proof>

lemma *weak*: ! *x xs*. *inDed xs* --> *inDed (x#xs)*
<proof>

lemma *inDed-mono*'[simplified *inDed-def*]: set *x* <= set *y* ==> *inDed x* ==>
inDed y
<proof>

lemma *inDed-mono*[simplified *inDed-def*]: *inDed x* ==> set *x* <= set *y* ==>
inDed y

<proof>

end

theory *Tree* **imports** *Main* **begin**

4.7 Tree

inductive-set

tree :: [*'a* => *'a set*, *'a*] => (*nat * 'a*) *set*
for *subs* :: *'a* => *'a set* **and** *gamma* :: *'a*

where

tree0: (*0*, *gamma*) : *tree subs gamma*

| *tree1*: [| (*n*, *delta*) : *tree subs gamma*; *sigma* : *subs(delta)* |]
=> (*Suc n*, *sigma*) : *tree subs gamma*

declare *tree.cases* [*elim*]

declare *tree.intros* [*intro*]

lemma *tree0Eq*: (*0*, *y*) : *tree subs gamma* = (*y* = *gamma*)

<proof>

lemma *tree1Eq* [*rule-format*]:

$\forall Y. (Suc\ n, Y) \in tree\ subs\ gamma = (\exists\ sigma \in\ subs\ gamma . (n, Y) \in tree\ subs\ sigma)$

<proof>

definition

incLevel :: *nat * 'a* => *nat * 'a* **where**
incLevel = (% (*n*, *a*). (*Suc n*, *a*))

lemma *injIncLevel*: *inj incLevel*

<proof>

lemma *treeEquation*: *tree subs gamma* = *insert (0, gamma) (UN delta:subs gamma . incLevel ` tree subs delta)*

<proof>

definition

fans :: [*'a* => *'a set*] => *bool* **where**
fans subs \longleftrightarrow (!*x*. *finite (subs x)*)

lemma *fansD*: *fans subs* ==> *finite (subs A)*

<proof>

lemma *fansI*: (!*A*. *finite (subs A)*) ==> *fans subs*

<proof>

lemma *inheritedUnEq*[rule-format, symmetric]: *inherited subs* $P \dashrightarrow (P A \ \& \ P B) = P (A \ \text{Un} \ B)$
<proof>

lemma *inheritedIncLevelEq*[rule-format, symmetric]: *inherited subs* $P \dashrightarrow P A = P (\text{incLevel} \ 'A)$
<proof>

lemma *inheritedInsertEq*[rule-format, symmetric]: *inherited subs* $P \dashrightarrow \sim(\text{terminal subs } \Gamma) \dashrightarrow P A = P (\text{insert } (n, \Gamma) \ A)$
<proof>

lemmas *inheritedUnD* = *iffD1*[OF *inheritedUnEq*]

lemmas *inheritedInsertD* = *inheritedInsertEq*[THEN *iffD1*]

lemmas *inheritedIncLevelD* = *inheritedIncLevelEq*[THEN *iffD1*]

lemma *inheritedUNEQ*[rule-format]:
finite A \dashrightarrow *inherited subs* $P \dashrightarrow (!x:A. P (B \ x)) = P (UN \ a:A. B \ a)$
<proof>

lemmas *inheritedUN* = *inheritedUNEQ*[THEN *iffD1*]

lemmas *inheritedUND*[rule-format] = *inheritedUNEQ*[THEN *iffD2*]

lemma *inheritedPropagateEq*[rule-format]: **assumes** *a*: *inherited subs* P
and *b*: *fans subs*
and *c*: $\sim(\text{terminal subs } \delta)$
shows $P(\text{tree subs } \delta) = (!\sigma:\text{subs } \delta. P(\text{tree subs } \sigma))$
<proof>

lemma *inheritedPropagate*:
[[$\sim P(\text{tree subs } \delta)$; *inherited subs* P ; *fans subs*; $\sim(\text{terminal subs } \delta)$]]
 $\implies \exists \sigma \in \text{subs } \delta . \sim P(\text{tree subs } \sigma)$
<proof>

lemma *inheritedViaSub*: [[*inherited subs* P ; *fans subs*; $P(\text{tree subs } \delta)$; $\sigma \in \text{subs } \delta$]]
 $\implies P(\text{tree subs } \sigma)$
<proof>

lemma *inheritedJoin*[rule-format]:
(inherited subs $P \ \& \ \text{inherited subs } Q) \dashrightarrow$ *inherited subs* $(\%x. P \ x \ \& \ Q \ x)$
<proof>

lemma *inheritedJoinI*[*rule-format*]: $[[\textit{inherited subs } P; \textit{inherited subs } Q; R = (\% x . P x \ \& \ Q x)]] \implies \textit{inherited subs } R$
 ⟨*proof*⟩

4.10 bounded, boundedBy

definition

boundedBy :: $\textit{nat} \implies (\textit{nat} * 'a) \textit{set} \implies \textit{bool}$ **where**
boundedBy $N A \iff (\forall (n, \textit{delta}) \in A. n < N)$

definition

bounded :: $(\textit{nat} * 'a) \textit{set} \implies \textit{bool}$ **where**
bounded $A \iff (\exists N . \textit{boundedBy } N A)$

lemma *boundedByEmpty*[*simp*]: *boundedBy* $N \{\}$
 ⟨*proof*⟩

lemma *boundedByInsert*: *boundedBy* $N (\textit{insert } (n, \textit{delta}) B)$ = $(n < N \ \& \ \textit{boundedBy } N B)$
 ⟨*proof*⟩

lemma *boundedByUn*: *boundedBy* $N (A \ \textit{Un} \ B) = (\textit{boundedBy } N A \ \& \ \textit{boundedBy } N B)$
 ⟨*proof*⟩

lemma *boundedByIncLevel'*: *boundedBy* $(\textit{Suc } N) (\textit{incLevel } ' A) = \textit{boundedBy } N A$
 ⟨*proof*⟩

lemma *boundedByAdd1*: *boundedBy* $N B \implies \textit{boundedBy } (N+M) B$
 ⟨*proof*⟩

lemma *boundedByAdd2*: *boundedBy* $M B \implies \textit{boundedBy } (N+M) B$
 ⟨*proof*⟩

lemma *boundedByMono*: *boundedBy* $m B \implies m < M \implies \textit{boundedBy } M B$
 ⟨*proof*⟩

lemmas *boundedByMonoD* = *boundedByMono*

lemma *boundedBy0*: *boundedBy* $0 A = (A = \{\})$
 ⟨*proof*⟩

lemma *boundedBySuc'*: *boundedBy* $N A \implies \textit{boundedBy } (\textit{Suc } N) A$
 ⟨*proof*⟩

lemma *boundedByIncLevel*: *boundedBy* $n (\textit{incLevel } ' (\textit{tree subs } \textit{gamma})) = (\exists m . n = \textit{Suc } m \ \& \ \textit{boundedBy } m (\textit{tree subs } \textit{gamma}))$
 ⟨*proof*⟩

lemma *boundedByUN*: *boundedBy N (UN x:A. B x) = (!x:A. boundedBy N (B x))*
 ⟨*proof*⟩

lemma *boundedBySuc*[*rule-format*]: *sigma ∈ subs Gamma ⇒ boundedBy (Suc n) (tree subs Gamma) → boundedBy n (tree subs sigma)*
 ⟨*proof*⟩

4.11 Inherited Properties- bounded

lemma *boundedEmpty*: *bounded {}*
 ⟨*proof*⟩

lemma *boundedUn*: *bounded (A Un B) = (bounded A & bounded B)*
 ⟨*proof*⟩

lemma *boundedIncLevel*: *bounded (incLevel' A) = (bounded A)*
 ⟨*proof*⟩

lemma *boundedInsert*: *bounded (insert a B) = (bounded B)*
 ⟨*proof*⟩

lemma *inheritedBounded*: *inherited subs bounded*
 ⟨*proof*⟩

4.12 founded

definition

*founded :: ['a => 'a set, 'a => bool, (nat * 'a) set] => bool* **where**
founded subs Pred = (%A. !(n,delta):A. terminal subs delta --> Pred delta)

lemma *foundedD*: *founded subs P (tree subs delta) ==> terminal subs delta ==> P delta*
 ⟨*proof*⟩

lemma *foundedMono*: *[| founded subs P A; ∀x. P x --> Q x |] ==> founded subs Q A*
 ⟨*proof*⟩

lemma *foundedSubs*: *founded subs P (tree subs Gamma) ⇒ sigma ∈ subs Gamma ⇒ founded subs P (tree subs sigma)*
 ⟨*proof*⟩

4.13 Inherited Properties- founded

lemma *foundedInsert*[*rule-format*]: *~ terminal subs delta --> founded subs P (insert (n,delta) B) = (founded subs P B)*
 ⟨*proof*⟩

lemma *foundedUn*: (*founded subs P (A Un B)*) = (*founded subs P A & founded subs P B*)

⟨*proof*⟩

lemma *foundedIncLevel*: *founded subs P (incLevel ' A)* = (*founded subs P A*)

⟨*proof*⟩

lemma *foundedEmpty*: *founded subs P {}*

⟨*proof*⟩

lemma *inheritedFounded*: *inherited subs (founded subs P)*

⟨*proof*⟩

4.14 Inherited Properties- finite

lemmas *finiteInsert = finite-insert*

lemma *finiteUn*: *finite (A Un B)* = (*finite A & finite B*)

⟨*proof*⟩

lemma *finiteIncLevel*: *finite (incLevel ' A)* = *finite A*

⟨*proof*⟩

lemma *finiteEmpty*: *finite {}* ⟨*proof*⟩

lemma *inheritedFinite*: *inherited subs (%A. finite A)*

⟨*proof*⟩

4.15 path: follows a failing inherited property through tree

definition

failingSub :: [*'a => 'a set, (nat * 'a) set => bool, 'a*] => *'a* **where**

failingSub subs P gamma = (*SOME sigma. (sigma:subs gamma & ~P(tree subs sigma))*)

lemma *failingSubProps*: [[*inherited subs P; ~P (tree subs gamma); ~(terminal subs gamma); fans subs*]]

==> *failingSub subs P gamma* ∈ *subs gamma & ~(P (tree subs (failingSub subs P gamma)))*

⟨*proof*⟩

lemma *failingSubFailsI*: [[*inherited subs P; ~P (tree subs gamma); ~(terminal subs gamma); fans subs*]]

==> *~(P (tree subs (failingSub subs P gamma)))*

⟨*proof*⟩

lemmas *failingSubFailsE = failingSubFailsI[THEN notE]*

lemma *failingSubSubs*: [[*inherited subs P; ~P (tree subs gamma); ~(terminal subs gamma); fans subs*]]

\implies *failingSub subs P gamma* \in *subs gamma*
 \langle *proof* \rangle

primrec *path* :: [*'a* \implies *'a set, 'a, (nat * 'a) set* \implies *bool, nat*] \implies *'a*
where

path0: *path subs gamma P 0* = *gamma*
| *pathSuc*: *path subs gamma P (Suc n)* = (if *terminal subs (path subs gamma P n)*
then *path subs gamma P n*
else *failingSub subs P (path subs gamma P n)*)

lemma *pathFailsP*: [[*inherited subs P; fans subs; ~P(tree subs gamma)*]]
 \implies \sim (*P (tree subs (path subs gamma P n))*)
 \langle *proof* \rangle

lemmas *PpathE* = *pathFailsP[THEN notE]*

lemma *pathTerminal[rule-format]*: [[*inherited subs P; fans subs; terminal subs gamma*]]
 \implies *terminal subs (path subs gamma P n)*
 \langle *proof* \rangle

lemma *pathStarts*: *path subs gamma P 0* = *gamma*
 \langle *proof* \rangle

lemma *pathSubs*: [[*inherited subs P; fans subs; ~P(tree subs gamma); ~ (terminal subs (path subs gamma P n))*]]
 \implies *path subs gamma P (Suc n)* \in *subs (path subs gamma P n)*
 \langle *proof* \rangle

lemma *pathStops*: *terminal subs (path subs gamma P n)* \implies *path subs gamma P (Suc n)* = *path subs gamma P n*
 \langle *proof* \rangle

4.16 Branch

definition

branch :: [*'a* \implies *'a set, 'a, nat* \implies *'a*] \implies *bool* **where**
branch subs Gamma f \longleftrightarrow *f 0 = Gamma*
& (!*n* . *terminal subs (f n)* \longrightarrow *f (Suc n) = f n*)
& (!*n* . \sim *terminal subs (f n)* \longrightarrow *f (Suc n) \in subs (f n)*)

lemma *branch0*: *branch subs Gamma f* \implies *f 0 = Gamma*
 \langle *proof* \rangle

lemma *branchStops*: *branch subs Gamma f* \implies *terminal subs (f n)* \implies *f (Suc n) = f n*
 \langle *proof* \rangle

lemma *branchSubs*: $\text{branch subs Gamma } f \implies \sim \text{terminal subs } (f \ n) \implies f \ (Suc \ n) \in \text{subs } (f \ n)$
 ⟨proof⟩

lemma *branchI*: $\llbracket (f \ 0 = \text{Gamma}); !n . \text{terminal subs } (f \ n) \dashrightarrow f \ (Suc \ n) = f \ n; !n . \sim \text{terminal subs } (f \ n) \dashrightarrow f \ (Suc \ n) \in \text{subs } (f \ n) \rrbracket \implies \text{branch subs Gamma } f$
 ⟨proof⟩

lemma *branchTerminalPropagates*: $\text{branch subs Gamma } f \implies \text{terminal subs } (f \ m) \implies \text{terminal subs } (f \ (m + n))$
 ⟨proof⟩

lemma *branchTerminalMono*: $\text{branch subs Gamma } f \implies m < n \implies \text{terminal subs } (f \ m) \implies \text{terminal subs } (f \ n)$
 ⟨proof⟩

lemma *branchPath*:
 $\llbracket \text{inherited subs } P; \text{fans subs}; \sim P(\text{tree subs gamma}) \rrbracket \implies \text{branch subs gamma } (\text{path subs gamma } P)$
 ⟨proof⟩

4.17 failing branch property: abstracts path defn

lemma *failingBranchExistence*: $!!\text{subs} . \llbracket \text{inherited subs } P; \text{fans subs}; \sim P(\text{tree subs gamma}) \rrbracket \implies \exists f . \text{branch subs gamma } f \ \& \ (\forall n . \sim P(\text{tree subs } (f \ n)))$
 ⟨proof⟩

definition

$\text{infBranch} :: ['a \implies 'a \ \text{set}, 'a, \text{nat} \implies 'a] \implies \text{bool}$ **where**
 $\text{infBranch subs Gamma } f \longleftrightarrow f \ 0 = \text{Gamma} \ \& \ (\forall n . f \ (Suc \ n) \in \text{subs } (f \ n))$

lemma *infBranchI*: $\llbracket (f \ 0 = \text{Gamma}); !n . f \ (Suc \ n) \in \text{subs } (f \ n) \rrbracket \implies \text{infBranch subs Gamma } f$
 ⟨proof⟩

4.18 Tree induction principles

— we work hard to use nothing fancier than induction over naturals

lemma *boundedTreeInduction'*:
 $\llbracket \text{fans subs}; \forall \text{delta} . \sim \text{terminal subs delta} \dashrightarrow (\forall \text{sigma} \in \text{subs delta} . P \ \text{sigma}) \dashrightarrow P \ \text{delta} \rrbracket \implies \forall \text{Gamma} . \text{boundedBy } m \ (\text{tree subs Gamma}) \longrightarrow \text{founded subs } P \ (\text{tree subs Gamma}) \longrightarrow P \ \text{Gamma}$
 ⟨proof⟩

lemma *boundedTreeInduction*:

```
[[ fans subs;
   bounded (tree subs Gamma); founded subs P (tree subs Gamma);
   ∀ delta. ~ terminal subs delta --> (∀ sigma ∈ subs delta. P sigma) --> P delta
]] ==> P Gamma
⟨proof⟩
```

lemma *boundedTreeInduction2'*:

```
[| fans subs;
   ∀ delta. (∀ sigma ∈ subs delta. P sigma) --> P delta |]
==> ∀ Gamma. boundedBy m (tree subs Gamma) → P Gamma
⟨proof⟩
```

lemma *boundedTreeInduction2*:

```
[| fans subs; boundedBy m (tree subs Gamma);
   ∀ delta. (∀ sigma ∈ subs delta. P sigma) --> P delta |]
==> P Gamma
⟨proof⟩
```

end

5 Completeness

```
theory Completeness
imports Tree Sequents
begin
```

5.1 pseq: type represents a processed sequent

```
types atom = (signs * predicate * vbl list)
       nform = (nat * formula)
       pseq = (atom list * nform list)
```

definition

```
sequent :: pseq => formula list where
sequent = (%(atoms,nforms) . map snd nforms @ map (% (z,p,vs) . FAtom z p
vs) atoms)
```

definition

```
pseq :: formula list => pseq where
pseq fs = ([], map (%f.(0,f)) fs)
```

definition *atoms* :: *pseq* => *atom list* **where** *atoms* = *fst*

definition *nforms* :: *pseq* => *nform list* **where** *nforms* = *snd*

5.2 subs: SATAxiom

definition

SATAxiom :: formula list => bool **where**
SATAxiom fs \longleftrightarrow (? n vs . *FAtom* Pos n vs : set fs & *FAtom* Neg n vs : set fs)

5.3 subs: a CutFreePC justifiable backwards proof step

definition

subsFAtom :: [atom list, (nat * formula) list, signs, predicate, vbl list] => pseq set
where
subsFAtom atms nAs z P vs = { ((z,P,vs)#atms,nAs) }

definition

subsFConj :: [atom list, (nat * formula) list, signs, formula, formula] => pseq set
where
subsFConj atms nAs z A0 A1 =
 (case z of
 Pos => { (atms,(0,A0)#nAs),(atms,(0,A1)#nAs) }
 Neg => { (atms,(0,A0)#(0,A1)#nAs) }

definition

subsFAll :: [atom list, (nat * formula) list, nat, signs, formula, vbl set] => pseq set
where
subsFAll atms nAs n z A frees =
 (case z of
 Pos => { let v = freshVar frees in (atms,(0,instanceF v A)#nAs) }
 Neg => { (atms,(0,instanceF (X n) A)#nAs @ [(Suc n,FAll Neg A)]) }

definition

subs :: pseq => pseq set **where**
subs = (% pseq .
 if *SATAxiom* (sequent pseq) then
 {}
 else let (atms,nforms) = pseq
 in case nforms of
 [] => {}
 | nA#nAs => let (n,A) = nA
 in (case A of
 FAtom z P vs => *subsFAtom* atms nAs z P vs
 | *FConj* z A0 A1 => *subsFConj* atms nAs z A0 A1
 | *FAll* z A => *subsFAll* atms nAs n z A (freeVarsFL
 (sequent pseq))))

5.4 proofTree(Gamma) says whether tree(Gamma) is a proof

definition

proofTree :: (nat * pseq) set => bool **where**
proofTree A \longleftrightarrow bounded A & founded subs (*SATAxiom* o sequent) A

5.5 path: considers, contains, costBarrier

definition

considers :: [nat => pseq,nat * formula,nat] => bool **where**
considers f nA n = (case (snd (f n)) of [] => False | x#xs => x=nA)

definition

contains :: [nat => pseq,nat * formula,nat] => bool **where**
contains f nA n \longleftrightarrow nA : set (snd (f n))

definition

costBarrier :: [nat * formula,pseq] => nat **where**

costBarrier nA = (%(atms,nAs).
 let barrier = takeWhile (%x. nA \sim x) nAs
 in let costs = map (exp 3 o size o snd) barrier
 in sumList costs)

5.6 path: eventually

definition

EV :: [nat => bool] => bool **where**
EV f == (? n . f n)

5.7 path: counter model

definition

counterM :: (nat => pseq) => object set **where**
counterM f = range obj

definition

counterEvalP :: (nat => pseq) => predicate => object list => bool **where**
counterEvalP f = (%p args . ! i . \sim (EV (contains f (i,FAtom Pos p (map (X o inv obj) args))))))

definition

counterModel :: (nat => pseq) => model **where**
counterModel f = Abs-model (counterM f, counterEvalP f)

primrec *counterAssign* :: vbl => object
where *counterAssign (X n)* = obj n

5.8 subs: finite

lemma *finite-subs: finite (subs gamma)*
 <proof>

lemma *fansSubs: fans subs*
 <proof>

lemma *subs-def2:*
 !!gamma.

```

~ SATAxiom (sequent gamma) ==>
subs gamma = (case nforms gamma of
  [] => {}
  | nA#nAs => let (n,A) = nA
              in (case A of
                  FAtom z P vs => subsFAtom (atoms gamma)
                  | FConj z A0 A1 => subsFConj (atoms gamma)
                  | FAll z A => subsFAll (atoms gamma) nAs n
                  z A (freeVarsFL (sequent gamma))))
  <proof>

```

5.9 inherited: proofTree

lemma *proofTree-def2*: *proofTree* = (% x . bounded x & founded subs (SATAxiom o sequent) x)
 <proof>

lemma *inheritedProofTree*: *inherited subs proofTree*
 <proof>

lemma *proofTreeI*: [| bounded A; founded subs (SATAxiom o sequent) A |] ==>
proofTree A
 <proof>

lemma *proofTreeBounded*: *proofTree A* ==> *bounded A*
 <proof>

lemma *proofTreeTerminal*: *proofTree A* ==> (n,delta) : A ==> *terminal subs delta* ==> *SATAxiom (sequent delta)*
 <proof>

5.10 pseq: lemma

lemma *snd-o-Pair*: (*snd o (Pair x)*) = (% x. x)
 <proof>

lemma *sequent-pseq*: *sequent (pseq fs)* = *fs*
 <proof>

5.11 SATAxiom: proofTree

lemma *SATAxiomTerminal[rule-format]*: *SATAxiom (sequent gamma)* --> *terminal subs gamma*
 <proof>

lemma *SATAxiomBounded*: *SATAxiom (sequent gamma)* ==> *bounded (tree subs gamma)*
 <proof>

lemma *SATAxiomFounded*: *SATAxiom* (sequent gamma) ==> founded subs (*SATAxiom* o sequent) (tree subs gamma)
 ⟨proof⟩

lemma *SATAxiomProofTree*[rule-format]: *SATAxiom* (sequent gamma) --> proofTree (tree subs gamma)
 ⟨proof⟩

lemma *SATAxiomEq*: (proofTree (tree subs gamma) & terminal subs gamma) = *SATAxiom* (sequent gamma)
 ⟨proof⟩

5.12 SATAxioms are deductions: - needed

lemma *SAT-deduction*: *SATAxiom* x ==> x : deductions CutFreePC
 ⟨proof⟩

5.13 proofTrees are deductions: subs respects rules - messy start and end

lemma *subsJustified'*:

notes ss = subs-def2 nforms-def Let-def atoms-def sequent-def subsFAtom-def subsFConj-def subsFAll-def

shows \neg *SATAxiom* (sequent (ats, (n, f) # list)) --> \neg terminal subs (ats, (n, f) # list)
 --> (\forall sigma \in subs (ats, (n, f) # list). sequent sigma \in deductions CutFreePC)

--> sequent (ats, (n, f) # list) \in deductions CutFreePC
 ⟨proof⟩

lemma *subsJustified*: !! gamma. \sim terminal subs gamma
 ==> ! sigma : subs gamma . sequent sigma : deductions (CutFreePC)
 ==> sequent gamma : deductions (CutFreePC)
 ⟨proof⟩

5.14 proofTrees are deductions: instance of boundedTreeInduction

lemmas *proofTreeD* = proofTree-def [THEN iffD1]

lemma *proofTreeDeductionD*[rule-format]: proofTree(tree subs gamma) ==> sequent gamma : deductions (CutFreePC)
 ⟨proof⟩

5.15 contains, considers:

lemma *contains-def2*: contains f iA n = (iA : set (nforms (f n)))
 ⟨proof⟩

lemma *considers-def2*: *considers f iA n = (? nAs . nforms (f n) = iA#nAs)*
 ⟨*proof*⟩

lemmas *containsI = contains-def2[THEN iffD2]*

5.16 path: nforms = [] implies

lemma *nformsNoContains*: *[] branch subs gamma f; !n . ~proofTree (tree subs (f n)); nforms (f n) = [] ==> ~ contains f iA n*
 ⟨*proof*⟩

lemma *nformsTerminal*: *nforms (f n) = [] ==> terminal subs (f n)*
 ⟨*proof*⟩

lemma *nformsStops*: *!!f.*
[] branch subs gamma f; !n . ~proofTree (tree subs (f n));
nforms (f n) = []
==> nforms (f (Suc n)) = [] & atoms (f (Suc n)) = atoms (f n)
 ⟨*proof*⟩

5.17 path: cases

lemma *terminalNFormCases*: *!!f. terminal subs (f n) | (? i A nAs . nforms (f n) = (i,A)#nAs)*
 ⟨*proof*⟩

lemma *cases[elim-format]*: *terminal subs (f n) | (¬ (terminal subs (f n) ∧ (? i A nAs . nforms (f n) = (i,A)#nAs)))*
 ⟨*proof*⟩

5.18 path: contains not terminal and propagate condition

lemma *containsNotTerminal*: *[] branch subs gamma f; !n . ~proofTree (tree subs (f n)); contains f iA n ==> ~ (terminal subs (f n))*
 ⟨*proof*⟩

lemma *containsPropagates*: *!!f.*
[] branch subs gamma f; !n . ~proofTree (tree subs (f n));
contains f iA n
==> contains f iA (Suc n) | considers f iA n
 ⟨*proof*⟩

5.19 path: no consider lemmas

lemma *noConsidersD*: *!!f. ~ considers f iA n ==> nforms (f n) = x#xs ==> iA ~ = x*
 ⟨*proof*⟩

lemma *considersD*: *!!f. considers f iA n ==> ? xs . nforms (f n) = iA#xs*
 ⟨*proof*⟩

5.20 path: contains initially

lemma *contains-initially*:

branch subs (pseq gamma) f \implies A : set gamma \implies (contains f (0,A) 0)
<proof>

lemma *contains-initialEVs*:

branch subs (pseq gamma) f \implies A : set gamma \implies EV (contains f (0,A))
<proof>

5.21 termination: (for EV contains implies EV considers)

lemmas *r = wf-induct[of measure msrFn, OF wf-measure]*

lemmas *r' = r[simplified measure-def inv-image-def less-than-def less-eq mem-Collect-eq]*

lemma *r''*: $(\forall x. (\forall y. ((msrFn::'a \Rightarrow nat) y) < ((msrFn :: 'a \Rightarrow nat) x)) \longrightarrow P y) \longrightarrow P x) \implies P a$
<proof>

lemma *terminationRule* [rule-format]:

! n. P n \dashrightarrow ($\sim(P (Suc n)) \mid (P (Suc n) \ \& \ msrFn (Suc n) < (msrFn::nat \Rightarrow nat) n)$) \implies P m \dashrightarrow (? n . P n $\ \& \ \sim(P (Suc n))$)
(is - \implies ?P m)
<proof>

5.22 costBarrier: lemmas

5.23 costBarrier: exp3 lemmas - bit specific...

lemma *exp3Min*: *exp 3 a > 0*

<proof>

lemma *exp1*: *exp 3 (A) + exp 3 (B) < 3 * ((exp 3 A) * (exp 3 B))*

<proof>

lemma *exp1'*: *exp 3 (A) < 3 * ((exp 3 A) * (exp 3 B)) + C*

<proof>

lemma *exp2*: *Suc 0 < 3 * exp 3 (B)*

<proof>

lemma *expSum*: *exp x (a+b) = (exp x a) * (exp x b)*

<proof>

5.24 costBarrier: decreases whilst contains and unconsiders

lemma *costBarrierDecreases'*:

notes *ss = subs-def2 nforms-def Let-def subsFAtom-def subsFConj-def subsFAll-def costBarrier-def atoms-def exp3Min expSum*

shows *\sim SATAxiom (sequent*

$(a, (num, fm) \# list) \dashrightarrow iA \sim = (num, fm) \dashrightarrow \neg \text{proofTree } (tree \text{ subs } (a, (num, fm) \# list)) \dashrightarrow fSucn : \text{subs } (a, (num, fm) \# list) \dashrightarrow iA \in \text{set list} \dashrightarrow \text{costBarrier } iA (fSucn) < \text{costBarrier } iA (a, (num, fm) \# list)$
 $\langle \text{proof} \rangle$

lemma *costBarrierDecreases*:

$[[\text{branch subs gamma } f;$
 $!n . \sim \text{proofTree } (tree \text{ subs } (f \ n));$
 $\text{contains } f \ iA \ n;$
 $\sim(\text{considers } f \ iA) \ n$
 $]] \implies \text{costBarrier } iA (f \ (Suc \ n)) < \text{costBarrier } iA (f \ n)$
 $\langle \text{proof} \rangle$

5.25 path: EV contains implies EV considers

lemma *considersContains*: $\text{considers } f \ iA \ n \implies \text{contains } f \ iA \ n$
 $\langle \text{proof} \rangle$

lemma *containsConsiders*: $[[\text{branch subs gamma } f; !n . \sim \text{proofTree } (tree \text{ subs } (f \ n));$
 $EV (\text{contains } f \ iA) \]]$
 $\implies EV (\text{considers } f \ iA)$
 $\langle \text{proof} \rangle$

5.26 EV contains: common lemma

lemma *lemmaA*:

$[[\text{branch subs gamma } f; !n . \sim \text{proofTree } (tree \text{ subs } (f \ n));$
 $EV (\text{contains } f \ (i, A)) \]]$
 $\implies ? \ n \ nAs . \sim \text{SATAxiom } (\text{sequent } (f \ n)) \ \& \ (nforms \ (f \ n) = (i, A) \ \# \ nAs \ \& \ f \ (Suc \ n) : \text{subs } (f \ n))$
 $\langle \text{proof} \rangle$

5.27 EV contains: FConj, FDisj, FAll

lemma *EV-disj*: $(EV \ P \ | \ EV \ Q) = EV \ (\lambda n. \ P \ n \ | \ Q \ n)$
 $\langle \text{proof} \rangle$

lemma *evContainsConj*: $[[EV (\text{contains } f \ (i, FConj \ Pos \ A0 \ A1));$
 $\text{branch subs gamma } f; !n . \sim \text{proofTree } (tree \text{ subs } (f \ n))$
 $]] \implies EV (\text{contains } f \ (0, A0)) \ | \ EV (\text{contains } f \ (0, A1))$
 $\langle \text{proof} \rangle$

lemma *evContainsDisj*: $[[EV (\text{contains } f \ (i, FConj \ Neg \ A0 \ A1));$
 $\text{branch subs gamma } f; !n . \sim \text{proofTree } (tree \text{ subs } (f \ n))$
 $]] \implies EV (\text{contains } f \ (0, A0)) \ \& \ EV (\text{contains } f \ (0, A1))$
 $\langle \text{proof} \rangle$

lemma *evContainsAll*:

$$\begin{aligned} & \llbracket EV (\text{contains } f (i, \text{Fall Pos } \text{body})); \text{branch subs } \gamma f; !n . \sim \text{proofTree } (\text{tree} \\ & \text{subs } (f n)) \\ & \quad \llbracket \\ & \implies ? v . EV (\text{contains } f (0, \text{instanceF } v \text{ body})) \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *evContainsEx-instance:*

$$\begin{aligned} & \llbracket EV (\text{contains } f (i, \text{Fall Neg } \text{body})); \text{branch subs } \gamma f; !n . \sim \text{proofTree } (\text{tree} \\ & \text{subs } (f n)) \\ & \quad \llbracket \\ & \implies EV (\text{contains } f (0, \text{instanceF } (X i) \text{ body})) \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *evContainsEx-repeat:*

$$\begin{aligned} & \llbracket \text{branch subs } \gamma f; !n . \sim \text{proofTree } (\text{tree subs } (f n)); \\ & \quad EV (\text{contains } f (i, \text{Fall Neg } \text{body})) \rrbracket \\ & \implies EV (\text{contains } f (\text{Suc } i, \text{Fall Neg } \text{body})) \\ & \langle \text{proof} \rangle \end{aligned}$$

5.28 EV contains: lemmas (temporal related)

lemma *lemma1:* $\llbracket P A n ; !A n . P A n \dashrightarrow P A (\text{Suc } n) \rrbracket$
 $\implies P A (n + m)$
 $\langle \text{proof} \rangle$

lemma *lemma2:*

$$\begin{aligned} & \llbracket P A n ; P B m ; ! A n . P A n \dashrightarrow P A (\text{Suc } n) \rrbracket \\ & \implies ? n . P A n \& P B n \\ & \langle \text{proof} \rangle \end{aligned}$$

5.29 EV contains: FAtoms

lemma *notTerminalNotSATAxiom:* $\neg \text{terminal subs } \gamma \implies \neg \text{SATAxiom}$
(sequent gamma)
 $\langle \text{proof} \rangle$

lemma *notTerminalNforms:* $\neg \text{terminal subs } (f n) \implies \text{nforms } (f n) \neq \llbracket$
 $\langle \text{proof} \rangle$

lemma *atomsPropagate:* $\llbracket \text{branch subs } \gamma f \rrbracket$
 $\implies x : \text{set } (\text{atoms } (f n)) \dashrightarrow x : \text{set } (\text{atoms } (f (\text{Suc } n)))$
 $\langle \text{proof} \rangle$

5.30 EV contains: FEx cases

lemma *evContainsEx0-allRepeats:*

$$\begin{aligned} & \llbracket \text{branch subs } \gamma f; !n . \sim \text{proofTree } (\text{tree subs } (f n)); \\ & \quad EV (\text{contains } f (0, \text{Fall Neg } A)) \rrbracket \\ & \implies EV (\text{contains } f (i, \text{Fall Neg } A)) \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *evContainsEx0-allInstances*:

[[*branch subs gamma f; !n . ~ proofTree (tree subs (f n));*
EV (contains f (0, Fall Neg A))]]
 \implies *EV (contains f (0, instanceF (X i) A))*
 <proof>

5.31 pseq: creates initial pseq

lemma *containsPSeq0D*: *branch subs (pseq fs) f* \implies *contains f (i, A) 0* \implies *i=0*
 <proof>

5.32 EV contains: contain any (i, FEx y) means contain all (i, FEx y)

lemma *claim*: $(A \mid B \mid C) = (\sim C \dashrightarrow \sim B \dashrightarrow A)$ <proof>

lemma *natPredCases*: $(!n. P n) \mid (\sim P 0) \mid (?n . P n \ \& \ \sim P (Suc n))$
 <proof>

lemma *containsNotTerminal'*:

[[*branch subs gamma f; !n . ~ proofTree (tree subs (f n)); contains f iA n*]] \implies
 \sim (*terminal subs (f n)*)
 <proof>

lemma *notTerminalSucNotTerminal*: [[\neg *terminal subs (f (Suc n)); branch subs gamma f*]] \implies \neg *terminal subs (f n)*
 <proof>

lemma *evContainsExSuc-containsEx*:

[[*branch subs (pseq fs) f; !n . ~ proofTree (tree subs (f n));*
EV (contains f (Suc i, Fall Neg body))]]
 \implies *EV (contains f (i, Fall Neg body))*
 <proof>

5.33 EV contains: contain any (i, FEx y) means contain all (i, FEx y)

lemma *evContainsEx-containsEx0*:

[[*branch subs (pseq fs) f; !n . ~ proofTree (tree subs (f n))*]]
 \implies *EV (contains f (i, Fall Neg A))* \dashrightarrow
EV (contains f (0, Fall Neg A))
 <proof>

lemma *evContainsExval*:

[[*EV (contains f (i, Fall Neg body)); branch subs (pseq fs) f; !n . ~ proofTree*
(tree subs (f n))]]
 \implies $!v .$ *EV (contains f (0, instanceF v body))*
 <proof>

5.34 EV contains: atoms

lemma *atomsInSequentI*[*rule-format*]: $(z, P, vs) : \text{set } (\text{fst } ps) \dashrightarrow$
 $\text{FAtom } z \ P \ vs : \text{set } (\text{sequent } ps)$
 $\langle \text{proof} \rangle$

lemma *evContainsAtom1*:
 $\llbracket \text{branch subs } (pseq \ fs) \ f; !n . \sim \text{proofTree } (\text{tree subs } (f \ n));$
 $\text{EV } (\text{contains } f \ (i, \text{FAtom } z \ P \ vs)) \rrbracket$
 $\implies ?n . (z, P, vs) : \text{set } (\text{fst } (f \ n))$
 $\langle \text{proof} \rangle$

lemmas *atomsPropagate''* = *atomsPropagate*[*rule-format*]
lemmas *atomsPropagate'* = *atomsPropagate''*[*simplified atoms-def, simplified*]

lemma *evContainsAtom*:
 $\llbracket \text{branch subs } (pseq \ fs) \ f; !n . \sim \text{proofTree } (\text{tree subs } (f \ n));$
 $\text{EV } (\text{contains } f \ (i, \text{FAtom } z \ P \ vs)) \rrbracket$
 $\implies ?n . (!m . \text{FAtom } z \ P \ vs : \text{set } (\text{sequent } (f \ (n + m))))$
 $\langle \text{proof} \rangle$

lemma *notEvContainsBothAtoms*:
 $\llbracket \text{branch subs } (pseq \ fs) \ f; !n . \sim \text{proofTree } (\text{tree subs } (f \ n)) \rrbracket$
 $\implies \sim \text{EV } (\text{contains } f \ (i, \text{FAtom } \text{Pos } p \ vs)) \mid$
 $\sim \text{EV } (\text{contains } f \ (j, \text{FAtom } \text{Neg } p \ vs))$
 $\langle \text{proof} \rangle$

5.35 counterModel: lemmas

lemma *counterModelInRepn*: $(\text{counterM } f, \text{counterEvalP } f) : \text{model}$
 $\langle \text{proof} \rangle$

lemmas *Abs-counterModel-inverse* = *counterModelInRepn*[*THEN Abs-model-inverse*]

lemma *inv-obj-obj*: $\text{inv obj } (\text{obj } n) = n$
 $\langle \text{proof} \rangle$

lemma *map-X-map-counterAssign*: $\text{map } X \ (\text{map } (\text{inv obj}) \ (\text{map } \text{counterAssign}$
 $\text{xs})) = \text{xs}$
 $\langle \text{proof} \rangle$

lemma *objectsCounterModel*: $\text{objects } (\text{counterModel } f) = \{ z . ?i . z = \text{obj } i \}$
 $\langle \text{proof} \rangle$

lemma *inCounterM*: $\text{counterAssign } v : \text{objects } (\text{counterModel } f)$
 $\langle \text{proof} \rangle$

lemma *counterAssign-eqI*[*rule-format*]: $x : \text{objects } (\text{counterModel } f) \dashrightarrow z = X$
 $(\text{inv obj } x) \dashrightarrow \text{counterAssign } z = x$
 $\langle \text{proof} \rangle$

lemma *evalPCounterModel*: $M = \text{counterModel } f \implies \text{evalP } M = \text{counterEvalP } f$
 <proof>

5.36 counterModel: all path formula value false - step by step

lemma *path-evalF'*:
notes $ss = \text{evalPCounterModel counterEvalP-def map-X-map-counterAssign map-compose}$
and $ss1 = \text{instanceF-def evalF-subF-eq comp-vblcase id-def[symmetric]}$
shows $[\![\text{branch subs (pseq fs) } f; \!]$
 $!n . \sim \text{proofTree (tree subs (f n))}$
 $]\!] \implies (? i . \text{EV (contains f (i,A))}) \longrightarrow \sim(\text{evalF (counterModel f) counterAssign } A)$
 <proof>

lemmas *path-evalF''* = $\text{mp}[\text{OF path-evalF}']$

5.37 adequacy

lemma *counterAssignModelAssign*: $\text{counterAssign} : \text{modelAssigns (counterModel gamma)}$
 <proof>

lemma *branch-contains-initially*: $\text{branch subs (pseq fs) } f \implies x : \text{set fs} \implies \text{contains } f (0,x) 0$
 <proof>

lemma *path-evalF*:
 $[\![\text{branch subs (pseq fs) } f; \!]$
 $\forall n. \neg \text{proofTree (tree subs (f n))};$
 $x \in \text{set fs}$
 $]\!] \implies \neg \text{evalF (counterModel f) counterAssign } x$
 <proof>

lemma *validProofTree*: $\sim \text{proofTree (tree subs (pseq fs))} \implies \sim(\text{validS fs})$
 <proof>

lemma *adequacy[simplified sequent-pseq]*: $\text{validS fs} \implies (\text{sequent (pseq fs)}) : \text{deductions CutFreePC}$
 <proof>

end

6 Soundness

theory *Soundness* **imports** *Completeness Multiset* **begin**

lemma *permutation-validS*: $fs \langle \sim \sim \rangle gs \dashv\vdash (validS\ fs = validS\ gs)$
<proof>

lemma *modelAssigns-vblcase*: $\phi \in modelAssigns\ M \implies x \in objects\ M \implies$
 $vblcase\ x\ \phi \in modelAssigns\ M$
<proof>

lemma *tmp*: $(!x : A. P\ x \mid Q) \implies (!x : A. P\ x) \mid Q$ *<proof>*

lemma *soundnessFAll*: $!!Gamma.$
 $[\![\ u \sim : freeVarsFL\ (FAll\ Pos\ A\ \# \ Gamma) ;$
 $validS\ (instanceF\ u\ A\ \# \ Gamma) \]\]$
 $\implies validS\ (FAll\ Pos\ A\ \# \ Gamma)$
<proof>

lemma *soundnessFEx*: $validS\ (instanceF\ x\ A\ \# \ Gamma) \implies validS\ (FAll\ Neg$
 $A\ \# \ Gamma)$
<proof>

lemma *soundnessFCut*: $[\![\ validS\ (C\ \# \ Gamma) ; validS\ (FNot\ C\ \# \ Delta) \]\]$
 $\implies validS\ (Gamma\ @\ Delta)$

<proof>

lemma *soundness*: $fs : deductions(PC) \implies (validS\ fs)$
<proof>

lemma *completeness*: $fs : deductions\ (PC) = validS\ fs$
<proof>

end