

Completeness for FOL

James Margetson, ported by Tom Ridge

March 17, 2025

Contents

1	Permutation Lemmas	1
1.1	perm, count equivalence	1
1.2	Properties closed under Perm and Contr hold for x iff hold for remdups x	2
1.3	List properties closed under Perm, Weak and Contr are mono- tonic in the set of the list	2
2	Base	4
2.1	Integrate with Isabelle libraries?	4
2.2	Summation	4
2.3	Termination Measure	4
2.4	Functions	4
3	Formula	5
3.1	Variables	5
3.2	Predicates	7
3.3	Formulas	7
3.4	formula signs induct, formula signs cases	8
3.5	Frees	9
3.6	Substitutions	9
3.7	Models	10
3.8	model, non empty set and positive atom valuation	10
3.9	Validity	11
4	Sequents	12
4.1	Rules	13
4.2	Deductions	13
4.3	Basic Rule sets	14
4.4	Derived Rules	15
4.5	Standard Rule Sets For Predicate Calculus	16
4.6	Monotonicity for CutFreePC deductions	17
4.7	Tree	17

4.8	Terminal	18
4.9	Inherited	18
4.10	bounded, boundedBy	20
4.11	Inherited Properties- bounded	21
4.12	founded	22
4.13	Inherited Properties- founded	22
4.14	Inherited Properties- finite	23
4.15	path: follows a failing inherited property through tree	23
4.16	Branch	24
4.17	failing branch property: abstracts path defn	25
4.18	Tree induction principles	25
5	Completeness	26
5.1	pseq: type represents a processed sequent	26
5.2	subs: SATAxiom	26
5.3	subs: a CutFreePC justifiable backwards proof step	26
5.4	proofTree(Gamma) says whether tree(Gamma) is a proof	27
5.5	path: considers, contains, costBarrier	27
5.6	path: eventually	28
5.7	path: counter model	28
5.8	subs: finite	28
5.9	inherited: proofTree	29
5.10	pseq: lemma	29
5.11	SATAxiom: proofTree	29
5.12	SATAxioms are deductions: - needed	30
5.13	proofTrees are deductions: subs respects rules - messy start and end	30
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	termination: (for EV contains implies EV considers)	33
5.20	costBarrier: lemmas	33
5.21	costBarrier: exp3 lemmas - bit specific...	33
5.22	costBarrier: decreases whilst contains and unconsiders	34
5.23	path: EV contains implies EV considers	34
5.24	EV contains: common lemma	35
5.25	EV contains: FConj,FDisj,FAll	35
5.26	EV contains: lemmas (temporal related)	37
5.27	EV contains: FAtoms	37
5.28	EEx contains: FEx cases	38
5.29	pseq: creates initial pseq	38

5.30	EV contains: contain any (i,FEx y) means contain all (i,FEx y)	38
5.31	EV contains: contain any (i,FEx y) means contain all (i,FEx y)	39
5.32	EV contains: atoms	40
5.33	counterModel: lemmas	40
5.34	counterModel: all path formula value false - step by step . . .	41
5.35	adequacy	42
6	Soundness	43

1 Permutation Lemmas

```
theory PermutationLemmas
imports HOL-Library.Multiset
begin
```

— following function is very close to that in multisets- now we can make the connection that $x < > y$ iff the multiset of x is the same as that of y

1.1 perm, count equivalence

```
lemma count-eq:
  ‹count-list xs x = Multiset.count (mset xs) x›
  by (induction xs) simp-all

lemma perm-count: mset A = mset B ⟹ (∀ x. count-list A x = count-list B x)
  by (simp add: count-eq)

lemma count-0: (∀ x. count-list B x = 0) = (B = [])
  by (simp add: count-list-0-iff)

lemma count-Suc: count-list B a = Suc m ⟹ a ∈ set B
  by (metis Zero-not-Suc count-notin)

lemma count-perm: !! B. (∀ x. count-list A x = count-list B x) ⟹ mset A =
  mset B
  by (simp add: count-eq multiset-eq-iff)

lemma perm-count-conv: mset A = mset B ⟷ (∀ x. count-list A x = count-list
  B x)
  by (simp add: count-eq multiset-eq-iff)
```

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

```
lemma remdups-append: y ∈ set ys ⟹ remdups (ws@y#ys) = remdups (ws@ys)
  by (induct ws; force)
```

```

lemma perm-contr':
  assumes perm:  $\bigwedge xs\ ys.\ mset\ xs = mset\ ys \implies (P\ xs = P\ ys)$ 
  and contr':  $\bigwedge x\ xs.\ P(x \# x \# xs) = P(x \# xs)$ 
  shows length\ xs = n  $\implies (P\ xs = P(\text{remdups}\ xs))$ 
proof(induction n arbitrary: xs rule: less-induct)
  case (less x)
  show ?case
  proof (cases distinct xs)
    case True
    then show ?thesis
    by (simp add: distinct-remdups-id)
  next
    case False
    then obtain ws\ ys\ zs\ y where xs: xs = ws@[y]@ys@[y]@zs
      using not-distinct-decomp by blast
    have P\ xs = P\ (ws@[y]@ys@[y]@zs) by (simp add: xs)
    also have ... = P\ ([y,y]@ws@ys@zs)
      by (intro perm) auto
    also have ... = P\ ([y]@ws@ys@zs)
      by (simp add: contr')
    also have ... = P\ (ws@ys@[y]@zs)
      by (intro perm) auto
    also have ... = P\ (remdups (ws@ys@[y]@zs))
      using less\ xs by force
    also have (remdups (ws@ys@[y]@zs)) = (remdups\ xs)
      by (simp add: xs remdups-append)
    finally show ?thesis .
  qed
qed

```

```

lemma perm-contr:
  assumes perm:  $\bigwedge xs\ ys.\ mset\ xs = mset\ ys \implies (P\ xs = P\ ys)$ 
  and contr':  $\bigwedge x\ xs.\ P(x \# x \# xs) = P(x \# xs)$ 
  shows (P\ xs = P(\text{remdups}\ xs))
  using perm-contr'[OF perm contr']
  by presburger

```

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

definition

```

rem :: 'a => 'a list => 'a list where
rem x\ xs = filter (%y. y ~ = x)\ xs

```

```

lemma rem: x \notin set (rem x\ xs)
  by(simp add: rem-def)

```

```

lemma length-rem: length (rem x\ xs) <= length\ xs

```

```

by(simp add: rem-def)

lemma rem-notin:  $x \notin \text{set } xs \implies \text{rem } x \text{ } xs = xs$ 
  by (metis (mono-tags, lifting) filter-True rem-def)

lemma perm-weak-filter':
  assumes perm:  $\bigwedge xs \text{ } ys. \text{mset } xs = \text{mset } ys \implies (P \text{ } xs = P \text{ } ys)$ 
  and weak:  $\bigwedge x \text{ } xs. P \text{ } xs \implies P \text{ } (x \# xs)$ 
  and P:  $P \text{ } (\text{ys}@\text{filter } Q \text{ } xs)$ 
  shows  $P \text{ } (\text{ys}@xs)$ 
proof -
  define zs where  $\langle zs = \text{filter } (\text{Not} \circ Q) \text{ } xs \rangle$ 
  have  $\langle P \text{ } (\text{filter } Q \text{ } xs @ ys) \rangle$ 
    by (metis P perm union-code union-commute)
  then have  $\langle P \text{ } (zs @ \text{filter } Q \text{ } xs @ ys) \rangle$ 
    by (induction zs) (simp-all add: weak)
  moreover have  $\text{mset } (zs @ \text{filter } Q \text{ } xs @ ys) = \text{mset } (ys @ xs)$ 
    by (simp add: zs-def)
  ultimately show  $\langle P \text{ } (ys @ xs) \rangle$ 
    using perm by blast
qed

lemma perm-weak-filter:
  assumes perm:  $\bigwedge xs \text{ } ys. \text{mset } xs = \text{mset } ys \implies (P \text{ } xs = P \text{ } ys)$ 
  and weak:  $\bigwedge x \text{ } xs. P \text{ } xs \implies P \text{ } (x \# xs)$ 
  shows  $P \text{ } (\text{filter } Q \text{ } xs) \implies P \text{ } xs$ 
  by (metis append-Nil perm perm-weak-filter' weak)

— Now can prove that in presence of perm, contr and weak, if  $\text{set } x \subseteq \text{set } y$  and  $P \text{ } x$  then  $P \text{ } y$ 

lemma perm-weak-contr-mono:
  assumes perm:  $\bigwedge xs \text{ } ys. \text{mset } xs = \text{mset } ys \implies (P \text{ } xs = P \text{ } ys)$ 
  and contr:  $\bigwedge x \text{ } xs. P(x \# x \# xs) \implies P \text{ } (x \# xs)$ 
  and weak:  $\bigwedge x \text{ } xs. P \text{ } xs \implies P \text{ } (x \# xs)$ 
  and xy:  $\text{set } x \subseteq \text{set } y$ 
  and Px:  $P \text{ } x$ 
  shows  $P \text{ } y$ 
proof -
  from contr weak have contr':  $\bigwedge x \text{ } xs. P(x \# x \# xs) = P \text{ } (x \# xs)$  by blast
  define y' where  $y' \equiv \text{filter } (\lambda z. z \in \text{set } x) \text{ } y$ 
  from xy have set x = set y'
    by (force simp: y'-def)
  hence rxry':  $\text{mset } (\text{remdups } x) = \text{mset } (\text{remdups } y')$ 
    using set-eq-iff-mset-remdups-eq by auto
  from Px perm-contr[OF perm contr] have Prx:  $P \text{ } (\text{remdups } x)$ 
    by blast
  with rxry' have P (remdups y')
    using perm by blast

```

```

with perm-contr[OF perm contr'] have  $P y'$  by simp
thus  $P y$ 
  using perm perm-weak-filter weak  $y'$ -def by blast
qed

end

```

2 Base

```

theory Base
imports PermutationLemmas
begin

```

2.1 Integrate with Isabelle libraries?

```

lemma natset-finite-max:
  assumes  $a: \text{finite } A$ 
  shows  $\text{Suc}(\text{Max } A) \notin A$ 
  using Max-ge Suc-n-not-le-n assms by blast

```

2.2 Summation

```

primrec summation ::  $(\text{nat} \Rightarrow \text{nat}) \Rightarrow \text{nat} \Rightarrow \text{nat}$ 
  where
     $\text{summation } f 0 = f 0$ 
  |  $\text{summation } f (\text{Suc } n) = f (\text{Suc } n) + \text{summation } f n$ 

```

2.3 Termination Measure

```

primrec sumList ::  $\text{nat list} \Rightarrow \text{nat}$ 
  where
     $\text{sumList } [] = 0$ 
  |  $\text{sumList } (x#xs) = x + \text{sumList } xs$ 

```

2.4 Functions

```
abbreviation (input) preImage  $\equiv \text{vimage}$ 
```

```
abbreviation (input) pre  $f a \equiv f^{-1}\{a\}$ 
```

definition

```

equalOn ::  $['a \text{ set}, 'a \Rightarrow 'b, 'a \Rightarrow 'b] \Rightarrow \text{bool}$  where
equalOn  $A f g = (\forall x \in A. f x = g x)$ 

```

```

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

```

```

lemma equalOn-Un:  $\text{equalOn } (A \cup B) f g = (\text{equalOn } A f g \wedge \text{equalOn } B f g)$ 
  by(auto simp add: equalOn-def)

```

```

lemma equalOnD: equalOn A f g  $\implies (\forall x \in A . f x = g x)$ 
by(simp add: equalOn-def)

lemma equalOnI:( $\forall x \in A . f x = g x$ )  $\implies$  equalOn A f g
by(simp add: equalOn-def)

lemma equalOn-UnD: equalOn (A Un B) f g ==> equalOn A f g & equalOn B f g
by(auto simp: equalOn-def)

lemma inj-inv-singleton[simp]:  $\llbracket \text{inj } f; f z = y \rrbracket \implies \{x. f x = y\} = \{z\}$ 
using inj-eq by fastforce

lemma finite-pre[simp]: inj f  $\implies$  finite (pre f x)
by (simp add: finite-vimageI)

declare finite-vimageI [simp]

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 \Rightarrow nat **where** deX (X n) = n

lemma X-deX[simp]: X (deX a) = a
by(cases a) simp

definition zeroX = X 0

primrec
 nextX :: vbl \Rightarrow vbl **where**
 nextX (X n) = X (Suc n)

definition
 vblcase :: [$'a, vbl \Rightarrow 'a, vbl$] $\Rightarrow 'a$ **where**
 vblcase a f n \equiv (@z. (n=zeroX \longrightarrow z=a) \wedge (!x. n=nextX x \longrightarrow z=f(x)))

```

declare [[case-translation vblcase zeroX nextX]]

definition
  freshVar :: vbl set  $\Rightarrow$  vbl where
    freshVar vs = X (LEAST n. n  $\notin$  deX ` vs)

lemma nextX-nextX[iff]: nextX x = nextX y = (x = y)
  by (metis Suc-inject nextX.simps vbl.exhaust vbl.inject)

lemma inj-nextX: inj nextX
  by(auto simp add: inj-on-def)

lemma ind:
  assumes P zeroX  $\wedge$  v. P v  $\Longrightarrow$  P (nextX v)
  shows P v'
proof –
  have P (X n) for n
    by (induction n) (use assms zeroX-def nextX-def in force)+  

    then show ?thesis
      by (metis X-deX)
qed

lemma zeroX-nextX[iff]: zeroX  $\neq$  nextX a
  by (metis nat.discI nextX.simps vbl.exhaust vbl.inject zeroX-def)

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

lemma nextX: nextX (X n) = X (Suc n)
  by simp

lemma vblcase-zeroX[simp]: vblcase a b zeroX = a
  by(simp add: vblcase-def)

lemma vblcase-nextX[simp]: vblcase a b (nextX n) = b n
  by(simp add: vblcase-def)

lemma vbl-cases: x = zeroX  $\vee$  ( $\exists$  y. x = nextX y)
  by (metis X-deX nextX old.nat.exhaust zeroX-def)

lemma vbl-casesE: [ x = zeroX  $\Longrightarrow$  P;  $\wedge$  y. x = nextX y  $\Longrightarrow$  P ]  $\Longrightarrow$  P
  using vbl-cases by blast

lemma comp-vblcase: f  $\circ$  vblcase a b = vblcase (f a) (f  $\circ$  b)
proof
  fix x
  show (f  $\circ$  vblcase a b) x = (case x of zeroX  $\Rightarrow$  f a | nextX x  $\Rightarrow$  (f  $\circ$  b) x)
    using vbl-cases [of x] by auto
qed

```

```

lemma equalOn-vblcaseI': equalOn (preImage nextX A) f g ==> equalOn A (vblcase x f) (vblcase x g)
  unfolding equalOn-def
  by (metis vbl-casesE vblcase-nextX vblcase-zeroX vimageI2)

lemma equalOn-vblcaseI: (zeroX ∈ A ==> x=y) ==> equalOn (preImage nextX A) f g ==> equalOn A (vblcase x f) (vblcase y g)
  by (smt (verit) equalOnD equalOnI equalOn-vblcaseI' vbl-casesE vblcase-nextX)

lemma X-deX-connection: X n ∈ A = (n ∈ (deX ` A))
  by force

lemma finiteFreshVar: finite A ==> freshVar A ∉ A
  unfolding freshVar-def
  by (metis (no-types, lifting) LeastI-ex X-deX finite-imageI
    finite-nat-set-iff-bounded image-eqI order-less-imp-triv
    vbl.inject)

lemma freshVarI: [|finite A; B ⊆ A|] ==> freshVar A ∉ B
  using finiteFreshVar in-mono by blast

lemma freshVarI2: [|finite A; ∀x . x ∉ A ==> P x|] ==> P (freshVar A)
  using finiteFreshVar by presburger

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
```

```
primrec sign :: signs ⇒ bool ⇒ bool
  where
    sign Pos x = x
  | sign Neg x = (¬ x)
```

```
primrec invSign :: signs ⇒ 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 [case-names FAtomP FAtomN FConjP FConjN FAllP
FAllN, cases type: formula]:
   $\llbracket \bigwedge p \text{ vs. } P (FAtom Pos p \text{ vs}); \bigwedge p \text{ vs. } P (FAtom Neg p \text{ vs}); \bigwedge A B . \llbracket P A; P B \rrbracket \implies P (FConj Pos A B); \bigwedge A B . \llbracket P A; P B \rrbracket \implies P (FConj Neg A B); \bigwedge A . \llbracket P A \rrbracket \implies P (FAll Pos A); \bigwedge A . \llbracket P A \rrbracket \implies P (FAll Neg A) \rrbracket \implies P A$ 
  by (induct A; rule signs.induct; force)

  — induction using nat induction, not wellfounded induction

lemma sizelemmas: size A < size (FConj z A B)
size B < size (FConj z A B)
size A < size (FAll z A)
by auto

primrec FNot :: formula  $\Rightarrow$  formula
where
  FNot-FAtom: FNot (FAtom z P vs) = FAtom (invSign z) P vs
  | FNot-FConj: FNot (FConj z A0 A1) = FConj (invSign z) (FNot A0) (FNot A1)
  | FNot-FAll: FNot (FAll z body) = FAll (invSign z) (FNot body)

primrec neg :: signs  $\Rightarrow$  signs
where
  neg Pos = Neg
  | neg Neg = Pos

primrec
  dual :: [(signs  $\Rightarrow$  signs), (signs  $\Rightarrow$  signs), (signs  $\Rightarrow$  signs)]  $\Rightarrow$  formula  $\Rightarrow$  formula
where
  dual-FAtom: dual p q r (FAtom z P vs) = FAtom (p z) P vs
  | dual-FConj: dual p q r (FConj z A0 A1) = FConj (q z) (dual p q r A0) (dual p q r A1)
  | dual-FAll: dual p q r (FAll z body) = FAll (r z) (dual p q r body)

lemma dualCompose: dual p q r  $\circ$  dual P Q R = dual (p  $\circ$  P) (q  $\circ$  Q) (r  $\circ$  R)
proof
  fix x
  show (dual p q r  $\circ$  dual P Q R) x = dual (p  $\circ$  P) (q  $\circ$  Q) (r  $\circ$  R) x
  by (induct x) auto
qed

lemma dualFNot': dual invSign invSign invSign = FNot
proof
  fix x

```

```

show dual invSign invSign invSign x = FNot x
  by (induct x) auto
qed

lemma dualFNot: dual invSign id id (FNot A) = FNot (dual invSign id id A)
  by(induct A) (auto simp: id-def)

lemma dualId: dual id id id A = A
  by(induct A) (auto simp: id-def)

```

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)  $\cup$  (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-FNot[simp]: freeVarsF (FNot A) = freeVarsF A
  by(induct A) auto

lemma finite-freeVarsF[simp]: finite (freeVarsF A)
  by(induct A) (auto simp add: inj-nextX)

lemma freeVarsFL-nil[simp]: freeVarsFL [] = {}
  by(simp add: freeVarsFL-def)

lemma freeVarsFL-cons: freeVarsFL (A# $\Gamma$ ) = freeVarsF A  $\cup$  freeVarsFL  $\Gamma$ 
  by(simp add: freeVarsFL-def)

lemma finite-freeVarsFL[simp]: finite (freeVarsFL  $\Gamma$ )
  by(induct  $\Gamma$ ) (auto simp: freeVarsFL-cons)

lemma freeVarsDual: freeVarsF (dual p q r A) = freeVarsF A
  by(induct A) auto

```

3.6 Substitutions

```

primrec subF :: [vbl  $\Rightarrow$  vbl,formula]  $\Rightarrow$  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 (λv . (case v of zeroX ⇒ zeroX | nextX v ⇒ nextX (theta v)))
body)

lemma size-subF: size (subF theta A) = size (A::formula)
  by(induct A arbitrary: theta) auto

lemma subFNot: subF theta (FNot A) = FNot (subF theta A)
  by(induct A arbitrary: theta) auto

lemma subFDual: subF theta (dual p q r A) = dual p q r (subF theta A)
  by(induct A arbitrary: theta) auto

definition
  instanceF :: [vbl,formula] ⇒ formula where
    instanceF w body = subF (λv. case v of zeroX ⇒ w | nextX v ⇒ v) body

lemma size-instance: size (instanceF v A) = size A
  by (simp add: instanceF-def size-subF)

lemma instanceFDual: instanceF u (dual p q r A) = dual p q r (instanceF u A)
  by(induct A) (simp-all add: instanceF-def subFDual)

```

3.7 Models

```

typedef
  object

```

```

axiomatization obj :: nat ⇒ object
  where inj-obj: inj obj

```

3.8 model, non empty set and positive atom valuation

```

definition model = {z :: (object set * ([predicate,object list] ⇒ bool)). (fst z ≠ {})}

```

```

typedef model = model
  unfolding model-def by auto

```

```

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 objectsNonEmpty: objects M ≠ {}
  using Rep-model[of M]
  by(simp add: objects-def model-def)

```

```

lemma modelsNonEmptyI: fst (Rep-model M) ≠ {}
  using Rep-model[of M] by(simp add: objects-def model-def)

```

3.9 Validity

```

primrec evalF :: [model,vbl ⇒ object,formula] ⇒ bool
  where
    evalFAtom: evalF M φ (FAtom z P vs) = sign z (evalP M P (map φ vs))
    | evalFConj: evalF M φ (FConj z A0 A1) = sign z (sign z (evalF M φ A0) ∧
      sign z (evalF M φ A1))
    | evalFAll: evalF M φ (FAll z body) =
      sign z (∀x∈objects M. sign z (evalF M (λv . (case v of zeroX ⇒ x | nextX
      v ⇒ φ v)) body))

```

definition

```

  valid :: formula ⇒ bool where
  valid F ≡ (∀M φ. evalF M φ F = True)

```

```

lemma evalF-FAll: evalF M φ (FAll Pos A) = (∀x∈objects M. (evalF M (vblcase
x φ) A))
  by simp

```

```

lemma evalF-FEx: evalF M φ (FAll Neg A) = (∃x∈objects M. (evalF M (vblcase
x φ) A))
  by simp

```

```

lemma evalF-arg2-cong: x = y ⇒ evalF M p x = evalF M p y
  by simp

```

```

lemma evalF-FNot: !!φ. evalF M φ (FNot A) = (¬ evalF M φ A)
  by(induct A rule: formula-signs-induct) simp-all

```

```

lemma evalF-equiv: equalOn (freeVarsF A) f g ⇒ evalF M f A = evalF M g A
  proof(induction A arbitrary: f g)

```

```

    case (FAtom x1 x2 x3)

```

```

    then show ?case

```

```

      unfolding equalOn-def

```

```

      by (metis (mono-tags, lifting) evalFAtom freeVarsFAtom map-eq-conv)

```

```

    next

```

```

    case (FConj x1 A1 A2)

```

```

    then show ?case

```

```

      by (smt (verit, ccfv-SIG) FConj equalOn-UnD evalFConj freeVarsFConj)

```

```

    next

```

```

    case (FAll x1 A)

```

```

    then show ?case

```

```

      by (metis (full-types) equalOn-vblcaseI' evalF-FAll evalF-FEx freeVarsFAll
          signs.exhaust)

```

```

qed

```

— composition of substitutions

```

lemma evalF-subF-eq: evalF M φ (subF theta A) = evalF M (φ o theta) A
proof (induction A arbitrary: φ theta)
  case (FAtom x1 x2 x3)
  then show ?case
    by (metis evalFAtom list.map-comp subFAtom)
next
  case (FConj x1 A1 A2)
  then show ?case
    by auto
next
  case (FAll x1 A)
  then have (vblcase x φ o vblcase zeroX (λv. nextX (theta v))) = (vblcase x (φ o theta))
    if x ∈ objects M for x
    using that
    apply (clar simp simp add: o-def fun-eq-iff split: vbl.splits)
    by (metis vbl-cases vblcase-nextX vblcase-zeroX)
  with FAll show ?case
    by (simp add: comp-def)
qed

lemma evalF-instance: evalF M φ (instanceF u A) = evalF M (vblcase (φ u) φ)
A
by (metis comp-id comp-vblcase evalF-subF-eq fun.map-ident instanceF-def)

lemma instanceF-E: instanceF g formula ≠ FAll signs formula
by (metis less-irrefl-nat size-instance sizemmas(3))

end

```

4 Sequents

```

theory Sequents
imports Formula
begin

type-synonym sequent = formula list

definition
evalS :: [model,vbl ⇒ object,formula list] ⇒ bool where
evalS M φ fs ≡ (∃f ∈ set fs . evalF M φ f = True)

```

```

lemma evalS-nil[simp]: evalS M φ [] = False
by(simp add: evalS-def)

```

```

lemma evalS-cons[simp]: evalS M φ (A # Γ) = (evalF M φ A ∨ evalS M φ Γ)
by(simp add: evalS-def)

```

```

lemma evalS-append: evalS M φ (Γ @ Δ) = (evalS M φ Γ ∨ evalS M φ Δ)
by(force simp add: evalS-def)

lemma evalS-equiv: (equalOn (freeVarsFL Γ) f g)  $\implies$  (evalS M f Γ = evalS M g
Γ)
by (induction Γ) (auto simp: equalOn-Un evalF-equiv freeVarsFL-cons)

```

definition

```

modelAssigns :: [model]  $\Rightarrow$  (vbl  $\Rightarrow$  object) set where
modelAssigns M = { φ . range φ  $\subseteq$  objects M }

```

```

lemma modelAssigns-iff [simp]: f ∈ modelAssigns M  $\longleftrightarrow$  range f  $\subseteq$  objects M
by(simp add: modelAssigns-def)

```

definition

```

validS :: formula list  $\Rightarrow$  bool where
validS fs  $\equiv$  ( $\forall$  M.  $\forall$  φ ∈ modelAssigns M . evalS M φ fs = True)

```

4.1 Rules

type-synonym rule = sequent * (sequent set)

definition

```

concR :: rule  $\Rightarrow$  sequent where
concR = ( $\lambda$ (conc,prems). conc)

```

definition

```

premsR :: rule  $\Rightarrow$  sequent set where
premsR = ( $\lambda$ (conc,prems). prems)

```

definition

```

mapRule :: (formula  $\Rightarrow$  formula)  $\Rightarrow$  rule  $\Rightarrow$  rule where
mapRule = ( $\lambda$ f (conc,prems) . (map f conc, (map f ` prems)))

```

```

lemma mapRuleI: [|A = map f a; B = map f ` b|]  $\implies$  (A, B) = mapRule f (a, b)
by(simp add: mapRule-def)
— FIXME tjr would like symmetric

```

4.2 Deductions

lemmas Powp-mono [mono] = Pow-mono [to-pred pred-subset-eq]

inductive-set

```

deductions :: rule set  $\Rightarrow$  formula list set
for rules :: rule set

```

where

```

inferI: [(conc,prems) ∈ rules; prems ∈ Pow(deductions(rules))]

```

$\] \implies conc \in deductions(rules)$

lemma *mono-deductions*: $\llbracket A \subseteq B \rrbracket \implies deductions(A) \subseteq deductions(B)$
apply(*best intro*: *deductions.inferI* *elim*: *deductions.induct*) **done**

4.3 Basic Rule sets

definition

$$Axioms = \{z. \exists p \text{ vs. } z = ([FAtom\ Pos\ p\ vs, FAtom\ Neg\ p\ vs], \{\})\}$$

definition

$$Conjs = \{z. \exists A0\ A1\ \Delta\ \Gamma. z = (FConj\ Pos\ A0\ A1\#\Gamma @ \Delta, \{A0\#\Gamma, A1\#\Delta\})\}$$

definition

$$Disjs = \{z. \exists A0\ A1 \quad \Gamma. z = (FConj\ Neg\ A0\ A1\#\Gamma, \{A0\#A1\#\Gamma\})\}$$

definition

$$Ails = \{z. \exists A\ x \quad \Gamma. z = (FAll\ Pos\ A\#\Gamma, \{instanceF\ x\ A\#\Gamma\}) \wedge x \notin freeVarsFL(FAll\ Pos\ A\#\Gamma)\}$$

definition

$$Exs = \{z. \exists A\ x \quad \Gamma. z = (FAll\ Neg\ A\#\Gamma, \{instanceF\ x\ A\#\Gamma\})\}$$

definition

$$Weaks = \{z. \exists A \quad \Gamma. z = (A\#\Gamma, \{\Gamma\})\}$$

definition

$$Contrs = \{z. \exists A \quad \Gamma. z = (A\#\Gamma, \{A\#A\#\Gamma\})\}$$

definition

$$Cuts = \{z. \exists C\ \Delta \quad \Gamma. z = (\Gamma @ \Delta, \{C\#\Gamma, FNot\ C\#\Delta\})\}$$

definition

$$Perms = \{z. \exists \Gamma\ \Delta \quad .\ z = (\Gamma, \{\Delta\}) \wedge mset\ \Gamma = mset\ \Delta\}$$

definition

$$DAxioms = \{z. \exists p \text{ vs. } z = ([FAtom\ Neg\ p\ vs, FAtom\ Pos\ p\ vs], \{\})\}$$

lemma *AxiomI*: $\llbracket Axioms \subseteq A \rrbracket \implies [FAtom\ Pos\ p\ vs, FAtom\ Neg\ p\ vs] \in deductions(A)$

by (*auto simp*: *Axioms-def deductions.simps*)

lemma *DAxiomsI*: $\llbracket DAxioms \subseteq A \rrbracket \implies [FAtom\ Neg\ p\ vs, FAtom\ Pos\ p\ vs] \in deductions(A)$

by (*auto simp*: *DAxioms-def deductions.simps*)

lemma *DisjI*: $\llbracket A0\#A1\#\Gamma \in deductions(A); Disjs \subseteq A \rrbracket \implies (FConj\ Neg\ A0\ A1\#\Gamma) \in deductions(A)$

by (force simp: Disjs-def deductions.simps)

lemma ConjI: $\llbracket (A0 \# \Gamma) \in \text{deductions}(A); (A1 \# \Delta) \in \text{deductions}(A); \text{Conjs} \subseteq A \rrbracket \implies F\text{Conj } \text{Pos } A0 \text{ } A1 \# \Gamma @ \Delta \in \text{deductions}(A)$
by (force simp: Conjs-def deductions.simps)

lemma AllI: $\llbracket \text{instanceF } w \text{ } A \# \Gamma \in \text{deductions}(R); w \notin \text{freeVarsFL } (\text{FAll } \text{Pos } A \# \Gamma); \text{Alls} \subseteq R \rrbracket \implies (\text{FAll } \text{Pos } A \# \Gamma) \in \text{deductions}(R)$
by (force simp: Alls-def deductions.simps)

lemma ExI: $\llbracket \text{instanceF } w \text{ } A \# \Gamma \in \text{deductions}(R); \text{Exs} \subseteq R \rrbracket \implies (\text{FAll } \text{Neg } A \# \Gamma) \in \text{deductions}(R)$
by (force simp: Exs-def deductions.simps)

lemma WeakI: $\llbracket \Gamma \in \text{deductions } R; \text{Weak}s \subseteq R \rrbracket \implies A \# \Gamma \in \text{deductions}(R)$
by (force simp: Weaks-def deductions.simps)

lemma ContrI: $\llbracket A \# A \# \Gamma \in \text{deductions } R; \text{Contrs} \subseteq R \rrbracket \implies A \# \Gamma \in \text{deductions}(R)$
by (force simp: Contrs-def deductions.simps)

lemma PermI: $\llbracket \text{Gamma}' \in \text{deductions } R; \text{mset } \Gamma = \text{mset } \text{Gamma}'; \text{Perms} \subseteq R \rrbracket \implies \Gamma \in \text{deductions}(R)$
using deductions.inferI [where prems={Gamma'}] Perms-def **by** blast

4.4 Derived Rules

lemma WeakI1: $\llbracket \Gamma \in \text{deductions}(A); \text{Weak}s \subseteq A \rrbracket \implies (\Delta @ \Gamma) \in \text{deductions}(A)$
by (induct Δ) (use WeakI in force)+

lemma WeakI2: $\llbracket \Gamma \in \text{deductions}(A); \text{Perms} \subseteq A; \text{Weak}s \subseteq A \rrbracket \implies (\Gamma @ \Delta) \in \text{deductions}(A)$
by (metis PermI WeakI1 mset-append union-commute)

lemma SATAxiomI: $\llbracket \text{Axioms} \subseteq A; \text{Weak}s \subseteq A; \text{Perms} \subseteq A; \text{forms} = [\text{FAtom } \text{Pos } n \text{ vs}, \text{FAtom } \text{Neg } n \text{ vs}] @ \Gamma \rrbracket \implies \text{forms} \in \text{deductions}(A)$
using AxiomI WeakI2 **by** presburger

lemma DisjI1: $\llbracket (A1 \# \Gamma) \in \text{deductions}(A); \text{Disjs} \subseteq A; \text{Weak}s \subseteq A \rrbracket \implies F\text{Conj } \text{Neg } A0 \text{ } A1 \# \Gamma \in \text{deductions}(A)$
using DisjI WeakI **by** presburger

lemma DisjI2: $\llbracket (A0 \# \Gamma) \in \text{deductions}(A); \text{Disjs} \subseteq A; \text{Weak}s \subseteq A; \text{Perms} \subseteq A \rrbracket \implies F\text{Conj } \text{Neg } A0 \text{ } A1 \# \Gamma \in \text{deductions}(A)$
using PermI [of $\langle A1 \# A0 \# \Gamma \rangle$]
by (metis DisjI WeakI append-Cons mset-append mset-rev rev.simps(2))

— FIXME the following 4 lemmas could all be proved for the standard rule sets using monotonicity as below

— we keep proofs as in original, but they are slightly ugly, and do not state

what is intuitively happening

```

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

 $\implies (a \# a \# A) @ \text{list} @ \text{list} \in \text{deductions } R$   

by (rule PermI, auto)

lemma weaken-append:  

 $\text{Contrs} \subseteq R \implies \text{Perms} \subseteq R \implies A @ \Gamma @ \Gamma \in \text{deductions}(R) \implies A @ \Gamma \in \text{deductions}(R)$   

proof (induction  $\Gamma$  arbitrary:  $A$ )  

  case Nil  

  then show ?case  

  by auto  

  next  

  case (Cons  $a \Gamma$ )  

  then have  $(a \# a \# A) @ \Gamma \in \text{deductions } R$   

  using perm-tmp4 by blast  

  then have  $a \# A @ \Gamma \in \text{deductions } R$   

  by (metis Cons.prems(1) ContrI append-Cons)  

  then show ?case  

  using Cons.prems(2) PermI by force  

qed

lemma ListWeakI:  $\text{Perms} \subseteq R \implies \text{Contrs} \subseteq R \implies x \# \Gamma @ \Gamma \in \text{deductions}(R)$   

 $\implies x \# \Gamma \in \text{deductions}(R)$   

by (metis append.left-neutral append-Cons weaken-append)

lemma ConjI':  $\llbracket (A0 \# \Gamma) \in \text{deductions}(A); (A1 \# \Gamma) \in \text{deductions}(A); \text{Contrs} \subseteq A; \text{Conjs} \subseteq A; \text{Perms} \subseteq A \rrbracket \implies F\text{Conj Pos } A0 A1 \# \Gamma \in \text{deductions}(A)$   

by (metis ConjI ListWeakI)

```

4.5 Standard Rule Sets For Predicate Calculus

definition

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

definition

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

```

lemma rulesInPCs:  $\text{Axioms} \subseteq PC$   $\text{Axioms} \subseteq CutFreePC$   

 $\text{Conjs} \subseteq PC$   $\text{Conjs} \subseteq CutFreePC$   

 $\text{Disjs} \subseteq PC$   $\text{Disjs} \subseteq CutFreePC$   

 $\text{Alls} \subseteq PC$   $\text{Alls} \subseteq CutFreePC$   

 $\text{Exs} \subseteq PC$   $\text{Exs} \subseteq CutFreePC$   

 $\text{Weaks} \subseteq PC$   $\text{Weaks} \subseteq CutFreePC$   

 $\text{Contrs} \subseteq PC$   $\text{Contrs} \subseteq CutFreePC$   

 $\text{Perms} \subseteq PC$   $\text{Perms} \subseteq CutFreePC$   

 $\text{Cuts} \subseteq PC$ 

```

```

 $CutFreePC \subseteq PC$ 
by(auto simp: PC-def CutFreePC-def)

```

4.6 Monotonicity for CutFreePC deductions

definition

```

inDed :: formula list  $\Rightarrow$  bool where
  inDed xs  $\equiv$  xs  $\in$  deductions CutFreePC

```

```

lemma perm: mset xs = mset ys  $\Rightarrow$  (inDed xs = inDed ys)
  by (metis PermI inDed-def rulesInPCs(16))

```

```

lemma contr: inDed (x#x#xs)  $\Rightarrow$  inDed (x#xs)
  using ContrI inDed-def rulesInPCs(14) by blast

```

```

lemma weak: inDed xs  $\Rightarrow$  inDed (x#xs)
  by (simp add: WeakI inDed-def rulesInPCs(12))

```

```

lemma inDed-mono'[simplified inDed-def]: set x  $\subseteq$  set y  $\Rightarrow$  inDed x  $\Rightarrow$  inDed y
  using contr perm perm-weak-contr-mono weak by blast

```

```

lemma inDed-mono[simplified inDed-def]: inDed x  $\Rightarrow$  set x  $\subseteq$  set y  $\Rightarrow$  inDed y
  using inDed-def inDed-mono' by auto

```

end

```

theory Tree
  imports Main

```

begin

4.7 Tree

inductive-set

```

tree :: ['a  $\Rightarrow$  'a set,'a]  $\Rightarrow$  (nat * 'a) set
for subs :: 'a  $\Rightarrow$  'a set and  $\gamma$  :: 'a

```

— This set represents the nodes in a tree which may represent a proof of γ .
Only storing the annotation and its level.

where

```

tree0: (0, $\gamma$ )  $\in$  tree subs  $\gamma$ 

```

```

| tree1: [(n,delta)  $\in$  tree subs  $\gamma$ ; sigma  $\in$  subs delta]
   $\Rightarrow$  (Suc n,sigma)  $\in$  tree subs  $\gamma$ 

```

```

declare tree.cases [elim]
declare tree.intros [intro]

```

```

lemma tree0Eq: (0,y)  $\in$  tree subs  $\gamma$  = (y =  $\gamma$ )
  by auto

```

```

lemma tree1Eq:

```

$(Suc\ n, Y) \in tree\ subs\ \gamma \longleftrightarrow (\exists\ sigma \in subs\ \gamma .\ (n, Y) \in tree\ subs\ sigma)$
by (induct n arbitrary: Y) force+
 — moving down a tree

definition

$incLevel :: nat * 'a \Rightarrow nat * 'a$ **where**
 $incLevel = (\% (n, a). (Suc\ n, a))$

lemma $injIncLevel: inj\ incLevel$
by (simp add: incLevel-def inj-on-def)

lemma $treeEquation: tree\ subs\ \gamma = insert\ (0, \gamma) (\bigcup_{\delta \in subs\ \gamma} incLevel\ ' tree\ subs\ \delta)$

proof —

have $a = 0 \wedge b = \gamma$
if $(a, b) \in tree\ subs\ \gamma$
and $\forall x \in subs\ \gamma. \forall (n, x) \in tree\ subs\ x. (a, b) \neq (Suc\ n, x)$
for a b
using that Zero-not-Suc
by (smt (verit) case-prod-conv tree.cases tree1Eq)
then show ?thesis
by (auto simp: incLevel-def image-iff tree1Eq case-prod-unfold)
qed

definition

$fans :: ['a \Rightarrow 'a set] \Rightarrow bool$ **where**
 $fans\ subs \equiv (\forall x. finite\ (subs\ x))$

4.8 Terminal

definition

$terminal :: ['a \Rightarrow 'a set, 'a] \Rightarrow bool$ **where**
 $terminal\ subs\ delta \equiv subs(\delta) = \{\}$

lemma $terminalD: terminal\ subs\ \Gamma \implies x \sim: subs\ \Gamma$
by (simp add: terminal-def)
 — not a good dest rule

lemma $terminalI: x \in subs\ \Gamma \implies \neg terminal\ subs\ \Gamma$
by (auto simp add: terminal-def)
 — not a good intro rule

4.9 Inherited

definition

$inherited :: ['a \Rightarrow 'a set, (nat * 'a) set \Rightarrow bool] \Rightarrow bool$ **where**
 $inherited\ subs\ P \equiv (\forall A\ B. (P\ A \wedge P\ B) = P\ (A\ Un\ B))$
 $\wedge (\forall A. P\ A = P\ (incLevel\ ' A))$
 $\wedge (\forall n\ \Gamma\ A. \neg(terminal\ subs\ \Gamma) \longrightarrow P\ A = P\ (insert\ (n, \Gamma)\ A))$
 $\wedge (P\ \{\})$

— FIXME tjr why does it have to be invariant under inserting nonterminal nodes?

```
lemma inheritedUn[rule-format]: inherited subs P → P A → P B → P (A Un B)
  by (auto simp add: inherited-def)

lemma inheritedIncLevel[rule-format]: inherited subs P → P A → P (incLevel ` A)
  by (auto simp add: inherited-def)

lemma inheritedEmpty[rule-format]: inherited subs P → P {}
  by (auto simp add: inherited-def)

lemma inheritedInsert[rule-format]:
  inherited subs P → ∼(terminal subs Γ) → P A → P (insert (n,Γ) A)
  by (auto simp add: inherited-def)

lemma inheritedI[rule-format]: ∀ A B . (P A ∧ P B) = P (A Un B);
  ∀ A . P A = P (incLevel ` A);
  ∀ n Γ A . ∼(terminal subs Γ) → P A = P (insert (n,Γ) A);
  P {}] ⇒ inherited subs P
  by (simp add: inherited-def)

lemma inheritedUnEq[rule-format, symmetric]: inherited subs P → (P A ∧ P B)
= P (A Un B)
  by (auto simp add: inherited-def)

lemma inheritedIncLevelEq[rule-format, symmetric]: inherited subs P → P A =
P (incLevel ` A)
  by (auto simp add: inherited-def)

lemma inheritedInsertEq[rule-format, symmetric]: inherited subs P → ∼(terminal subs Γ) → P A = P (insert (n,Γ) A)
  by (auto simp add: inherited-def)

lemmas inheritedUnD = iffD1[OF inheritedUnEq]

lemmas inheritedInsertD = inheritedInsertEq[THEN iffD1]

lemmas inheritedIncLevelD = inheritedIncLevelEq[THEN iffD1]

lemma inheritedUNEq:
  finite A ⇒ inherited subs P ⇒ (∀ x∈A. P (B x)) = P (∪ a∈A. B a)
  by (induction rule: finite-induct) (simp-all add: inherited-def)
```

```

lemmas inheritedUN = inheritedUNEq[THEN iffD1]

lemmas inheritedUND[rule-format] = inheritedUNEq[THEN iffD2]

lemma inheritedPropagateEq:
  assumes inherited subs P
  and fans subs
  and  $\neg$  (terminal subs delta)
  shows  $P(\text{tree subs } \delta) = (\forall \sigma \in \text{subs } \delta. P(\text{tree subs } \sigma))$ 
  using assms unfolding fans-def
  by (metis (mono-tags, lifting) inheritedIncLevelEq inheritedInsertEq inheritanceUNEq treeEquation)

lemma inheritedPropagate:
   $\llbracket \neg P(\text{tree subs } \delta); \text{inherited subs } P; \text{fans subs}; \neg \text{terminal subs } \delta \rrbracket$ 
   $\implies \exists \sigma \in \text{subs } \delta. \neg P(\text{tree subs } \sigma)$ 
  by (simp add: inheritedPropagateEq)

lemma inheritedViaSub:
   $\llbracket \text{inherited subs } P; \text{fans subs}; P(\text{tree subs } \delta); \sigma \in \text{subs } \delta \rrbracket \implies P(\text{tree subs } \sigma)$ 
  by (simp add: inheritedPropagateEq terminalI)

lemma inheritedJoin:
   $\llbracket \text{inherited subs } P; \text{inherited subs } Q \rrbracket \implies \text{inherited subs } (\lambda x. P x \wedge Q x)$ 
  by (smt (verit, best) inherited-def)

lemma inheritedJoinI:
   $\llbracket \text{inherited subs } P; \text{inherited subs } Q; R = (\lambda x. P x \wedge Q x) \rrbracket$ 
   $\implies \text{inherited subs } R$ 
  by (simp add: inheritedJoin)

```

4.10 bounded, boundedBy

definition
 $\text{boundedBy} :: \text{nat} \Rightarrow (\text{nat} * \text{'a}) \text{ set} \Rightarrow \text{bool}$ **where**
 $\text{boundedBy } N A \equiv (\forall (n, \delta) \in A. n < N)$

definition
 $\text{bounded} :: (\text{nat} * \text{'a}) \text{ set} \Rightarrow \text{bool}$ **where**
 $\text{bounded } A \equiv (\exists N. \text{boundedBy } N A)$

lemma boundedByEmpty[simp]: $\text{boundedBy } N \{\} \equiv \text{False}$
by(simp add: boundedBy-def)

lemma boundedByInsert: $\text{boundedBy } N (\text{insert } (n, \delta) B) = (n < N \wedge \text{boundedBy } N B)$
by(simp add: boundedBy-def)

```

lemma boundedByUn: boundedBy N (A Un B) = (boundedBy N A ∧ boundedBy N B)
  by(auto simp add: boundedBy-def)

lemma boundedByIncLevel': boundedBy (Suc N) (incLevel ` A) = boundedBy N A
  by(auto simp add: incLevel-def boundedBy-def)

lemma boundedByAdd1: boundedBy N B ==> boundedBy (N+M) B
  by(auto simp add: boundedBy-def)

lemma boundedByAdd2: boundedBy M B ==> boundedBy (N+M) B
  by(auto simp add: boundedBy-def)

lemma boundedByMono: boundedBy m B ==> m < M ==> boundedBy M B
  by(auto simp: boundedBy-def)

lemmas boundedByMonoD = boundedByMono

lemma boundedBy0: boundedBy 0 A = (A = {})
  by (auto simp add: boundedBy-def)

lemma boundedBySuc': boundedBy N A ==> boundedBy (Suc N) A
  by (auto simp add: boundedBy-def)

lemma boundedByIncLevel: boundedBy n (incLevel ` (tree subs γ)) ↔ (exists m . n
= Suc m ∧ boundedBy m (tree subs γ))
  by (metis boundedBy0 boundedByIncLevel' boundedBySuc' emptyE old.nat.exhaust
tree.tree0)

lemma boundedByUN: boundedBy N (UN x:A. B x) = (!x:A. boundedBy N (B x))
  by(simp add: boundedBy-def)

lemma boundedBySuc[rule-format]: sigma ∈ subs Γ ==> boundedBy (Suc n) (tree
subs Γ) ==> boundedBy n (tree subs sigma)
  by (metis boundedByIncLevel' boundedByInsert boundedByUN treeEquation)

```

4.11 Inherited Properties- bounded

```

lemma boundedEmpty: bounded {}
  by(simp add: bounded-def)

lemma boundedUn: bounded (A Un B) ↔ (bounded A ∧ bounded B)
  by (metis boundedByAdd1 boundedByAdd2 boundedByUn bounded-def)

lemma boundedIncLevel: bounded (incLevel` A) ↔ (bounded A)
  by (meson boundedByIncLevel' boundedBySuc' bounded-def)

lemma boundedInsert: bounded (insert a B) ↔ (bounded B)
  proof (cases a)

```

```

case (Pair a b)
then show ?thesis
by (metis Un-insert-left Un-insert-right boundedByEmpty boundedByInsert boundedUn
      bounded-def lessI)
qed

lemma inheritedBounded: inherited subs bounded
by(blast intro!: inheritedI boundedUn[symmetric] boundedIncLevel[symmetric]
      boundedInsert[symmetric] boundedEmpty)

```

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
by(simp add: treeEquation [of - delta] founded-def)

```

```

lemma foundedMono: [|founded subs P A; ∀x. P x → Q x|] ⇒ founded subs Q
A
by (auto simp: founded-def)

```

```

lemma foundedSubs: founded subs P (tree subs Γ) ⇒ sigma ∈ subs Γ ⇒ founded
subs P (tree subs sigma)
using tree1Eq by (fastforce simp: founded-def)

```

4.13 Inherited Properties- founded

```

lemma foundedInsert: ¬ terminal subs delta ⇒ founded subs P (insert (n,delta)
B) = (founded subs P B)
by (simp add: terminal-def founded-def)

```

```

lemma foundedUn: (founded subs P (A Un B)) = (founded subs P A ∧ founded
subs P B)
by(force simp add: founded-def)

```

```

lemma foundedIncLevel: founded subs P (incLevel ` A) = (founded subs P A)
by (simp add: case-prod-unfold founded-def incLevel-def)

```

```

lemma foundedEmpty: founded subs P {}
by(auto simp add: founded-def)

```

```

lemma inheritedFounded: inherited subs (founded subs P)
by (simp add: foundedEmpty foundedIncLevel foundedInsert foundedUn inherited-def)

```

4.14 Inherited Properties- finite

```

lemma finiteUn: finite (A Un B) = (finite A ∧ finite B)
  by simp

lemma finiteIncLevel: finite (incLevel ` A) = finite A
  by (meson finite-imageD finite-imageI injIncLevel inj-on-subset subset-UNIV)

lemma inheritedFinite: inherited subs finite
  by (simp add: finiteIncLevel inherited-def)

```

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 γ ≡ (SOME sigma. (sigma:subs γ ∧ ∼P(tree subs sigma)))

```

```

lemma failingSubProps:
  [|inherited subs P; ∼P (tree subs γ); ∼terminal subs γ; fans subs|]
    ⇒ failingSub subs P γ ∈ subs γ ∧ ∼P (tree subs (failingSub subs P γ))
  unfolding failingSub-def
  by (metis (mono-tags, lifting) inheritedPropagateEq some-eq-ex)

```

```

lemma failingSubFailsI:
  [|inherited subs P; ∼P (tree subs γ); ∼terminal subs γ; fans subs|]
    ⇒ ∼P (tree subs (failingSub subs P γ))
  by (simp add: failingSubProps)

```

```
lemmas failingSubFailsE = failingSubFailsI[THEN note]
```

```

lemma failingSubSubs:
  [|inherited subs P; ∼P (tree subs γ); ∼terminal subs γ; fans subs|]
    ⇒ failingSub subs P γ ∈ subs γ
  by (simp add: failingSubProps)

```

```

primrec path :: ['a ⇒ 'a set, 'a, (nat * 'a) set ⇒ bool, nat] ⇒ 'a
where
  path0: path subs γ P 0      = γ
  | pathSuc: path subs γ P (Suc n) = (if terminal subs (path subs γ P n)
    then path subs γ P n
    else failingSub subs P (path subs γ P n))

```

```

lemma pathFailsP:
  [|inherited subs P; fans subs; ∼P (tree subs γ)|]
    ⇒ ∼P (tree subs (path subs γ P n))
  by (induction n) (simp-all add: failingSubProps)

```

```
lemmas PpathE = pathFailsP[THEN note]
```

```

lemma pathTerminal:
   $\llbracket \text{inherited subs } P; \text{fans subs}; \text{terminal subs } \gamma \rrbracket$ 
   $\implies \text{terminal subs}(\text{path subs } \gamma P n)$ 
  by (induct n, simp-all)

lemma pathStarts: path subs  $\gamma$  P 0 =  $\gamma$ 
  by simp

lemma pathSubs:
   $\llbracket \text{inherited subs } P; \text{fans subs}; \neg P(\text{tree subs } \gamma); \neg \text{terminal subs}(\text{path subs } \gamma P n) \rrbracket$ 
   $\implies \text{path subs } \gamma P (\text{Suc } n) \in \text{subs}(\text{path subs } \gamma P n)$ 
  by (metis PpathE failingSubSubs pathSuc)

lemma pathStops: terminal subs (path subs  $\gamma$  P n)  $\implies$  path subs  $\gamma$  P (Suc n) =
  path subs  $\gamma$  P n
  by simp

```

4.16 Branch

definition

```

branch ::  $'a \Rightarrow 'a \text{ set}, 'a, \text{nat} \Rightarrow 'a \Rightarrow \text{bool}$  where
branch subs  $\Gamma f \longleftrightarrow f 0 = \Gamma$ 
 $\wedge (\forall n . \text{terminal subs}(f n) \longrightarrow f(\text{Suc } n) = f n)$ 
 $\wedge (\forall n . \neg \text{terminal subs}(f n) \longrightarrow f(\text{Suc } n) \in \text{subs}(f n))$ 

```

```

lemma branch0: branch subs  $\Gamma f \implies f 0 = \Gamma$ 
  by (simp add: branch-def)

```

```

lemma branchStops: branch subs  $\Gamma f \implies \text{terminal subs}(f n) \implies f(\text{Suc } n) = f n$ 
  by (simp add: branch-def)

```

```

lemma branchSubs: branch subs  $\Gamma f \implies \neg \text{terminal subs}(f n) \implies f(\text{Suc } n) \in$ 
   $\text{subs}(f n)$ 
  by (simp add: branch-def)

```

```

lemma branchI:  $\llbracket f 0 = \Gamma;$ 
 $\forall n . \text{terminal subs}(f n) \longrightarrow f(\text{Suc } n) = f n;$ 
 $\forall n . \neg \text{terminal subs}(f n) \longrightarrow f(\text{Suc } n) \in \text{subs}(f n) \rrbracket \implies \text{branch subs } \Gamma f$ 
  by (simp add: branch-def)

```

```

lemma branchTerminalPropagates: branch subs  $\Gamma f \implies \text{terminal subs}(f m) \implies$ 
   $\text{terminal subs}(f(m + n))$ 
  by (induct n, simp-all add: branchStops)

```

```

lemma branchTerminalMono:
  branch subs  $\Gamma f \implies m < n \implies \text{terminal subs}(f m) \implies \text{terminal subs}(f n)$ 
  by (metis branchTerminalPropagates
    canonically-ordered-monoid-add-class.lessE)

```

```

lemma branchPath:
   $\llbracket \text{inherited subs } P; \text{fans subs}; \neg P(\text{tree subs } \gamma) \rrbracket$ 
   $\implies \text{branch subs } \gamma (\text{path subs } \gamma P)$ 
  by (auto intro!: branchI pathStarts pathSubs pathStops)

```

4.17 failing branch property: abstracts path defn

```

lemma failingBranchExistence:
   $\llbracket \text{inherited subs } P; \text{fans subs}; \sim P(\text{tree subs } \gamma) \rrbracket$ 
   $\implies \exists f . \text{branch subs } \gamma f \wedge (\forall n . \neg P(\text{tree subs } (f n)))$ 
  by (metis PpathE branchPath)

```

definition

```

infBranch ::  $'a \Rightarrow 'a \text{ set}, 'a, \text{nat} \Rightarrow 'a \Rightarrow \text{bool}$  where
  infBranch subs  $\Gamma f \longleftrightarrow f 0 = \Gamma \wedge (\forall n. f(Suc n) \in \text{subs } (f n))$ 

```

```

lemma infBranchI:  $\llbracket f 0 = \Gamma; \forall n . f(Suc n) \in \text{subs } (f n) \rrbracket \implies \text{infBranch subs } \Gamma f$ 
  by (simp add: infBranch-def)

```

4.18 Tree induction principles

```

lemma boundedTreeInduction':
   $\llbracket \text{fans subs};$ 
   $\forall \text{delta}. \neg \text{terminal subs } \text{delta} \longrightarrow (\forall \text{sigma} \in \text{subs } \text{delta}. P \text{ sigma}) \longrightarrow P \text{ delta}$ 
   $\rrbracket \implies \forall \Gamma. \text{boundedBy } m (\text{tree subs } \Gamma) \longrightarrow \text{founded subs } P (\text{tree subs } \Gamma) \longrightarrow P \Gamma$ 
  proof (induction m)
    case 0
    then show ?case by (metis boundedBy0 empty_iff tree.tree0)
  next
    case (Suc m)
    then show ?case by (metis boundedBySuc foundedD foundedSubs)
  qed

```

— tjr tidied and introduced new lemmas

```

lemma boundedTreeInduction:
   $\llbracket \text{fans subs};$ 
   $\text{bounded } (\text{tree subs } \Gamma); \text{founded subs } P (\text{tree subs } \Gamma);$ 
   $\bigwedge \text{delta}. \llbracket \neg \text{terminal subs } \text{delta}; (\forall \text{sigma} \in \text{subs } \text{delta}. P \text{ sigma}) \rrbracket \implies P \text{ delta}$ 
   $\rrbracket \implies P \Gamma$ 
  by (metis boundedTreeInduction' bounded-def)

```

```

lemma boundedTreeInduction2:
   $\llbracket \text{fans subs};$ 
   $\forall \text{delta}. (\forall \text{sigma} \in \text{subs } \text{delta}. P \text{ sigma}) \longrightarrow P \text{ delta}$ 
   $\implies \text{boundedBy } m (\text{tree subs } \Gamma) \longrightarrow P \Gamma$ 
  proof (induction m arbitrary:  $\Gamma$ )

```

```

case 0
then show ?case by (metis boundedBy0 empty-iff tree.tree0)
next
  case (Suc m)
  then show ?case by (metis boundedBySuc)
qed

end

```

5 Completeness

```

theory Completeness
imports Tree Sequents
begin

```

5.1 pseq: type represents a processed sequent

```

type-synonym atom = (signs * predicate * vbl list)
type-synonym nform = (nat * formula)
type-synonym pseq = (atom list * nform list)

```

definition

```

sequent :: pseq  $\Rightarrow$  formula list where
  sequent = ( $\lambda(\text{atoms}, \text{nforms}) . \text{map} \text{ snd} \text{ nforms} @ \text{map} (\lambda(z, p, vs) . FAtom z p vs)$ )  

  atoms)

```

definition

```

pseq :: formula list  $\Rightarrow$  pseq where
  pseq fs = ([] , map ( $\lambda f . (0, f)$ ) fs)

```

```
definition atoms :: pseq  $\Rightarrow$  atom list where atoms = fst
```

```
definition nforms :: pseq  $\Rightarrow$  nform list where nforms = snd
```

5.2 subs: SATAxiom

definition

```

SATAxiom :: formula list  $\Rightarrow$  bool where
  SATAxiom fs  $\equiv$  ( $\exists n vs . FAtom Pos n vs \in set fs \wedge FAtom Neg n vs \in set fs$ )

```

5.3 subs: a CutFreePC justifiable backwards proof step

definition

```

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

```

definition

```

subsFConj :: [atom list, (nat * formula) list, signs, formula, formula]  $\Rightarrow$  pseq set
where

```

```

definition
subsFConj atms nAs z A0 A1 =
  (case z of
    Pos  $\Rightarrow$  { (atms,(0,A0)##nAs),(atms,(0,A1)##nAs) }
  | Neg  $\Rightarrow$  { (atms,(0,A0)##(0,A1)##nAs) })

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

definition
subs :: pseq  $\Rightarrow$  pseq set where
subs = ( $\lambda$ pseq .
  if SATAxiom (sequent pseq) then
    {}
  else let (atms,nforms) = pseq
    in case nforms of
      []  $\Rightarrow$  {}
    | nA##nAs  $\Rightarrow$  let (n,A) = nA
      in (case A of
        FAtom z P vs  $\Rightarrow$  subsFAtom atms nAs z P vs
        | FConj z A0 A1  $\Rightarrow$  subsFConj atms nAs z A0 A1
        | FAll z A  $\Rightarrow$  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  $\Rightarrow$  bool where
proofTree A  $\longleftrightarrow$  bounded A  $\wedge$  founded subs (SATAxiom  $\circ$  sequent) A

```

5.5 path: considers, contains, costBarrier

definition

```

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

```

definition

```

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

```

abbreviation (input) power3 \equiv power (3::nat)

definition

```

costBarrier :: [nat * formula,pseq]  $\Rightarrow$  nat where

```

```

costBarrier nA = ( $\lambda$ (atms,nAs).

```

```

let barrier = takeWhile ( $\lambda x. nA \neq x$ ) nAs
in let costs = map (power3  $\circ$  size  $\circ$  snd) barrier
in sumList costs)

```

5.6 path: eventually

definition

```

EV :: [nat  $\Rightarrow$  bool]  $\Rightarrow$  bool where
EV f  $\equiv$  ( $\exists n . f n$ )

```

5.7 path: counter model

definition

```

counterM :: (nat  $\Rightarrow$  pseq)  $\Rightarrow$  object set where
counterM f  $\equiv$  range obj

```

definition

```

counterEvalP :: (nat  $\Rightarrow$  pseq)  $\Rightarrow$  predicate  $\Rightarrow$  object list  $\Rightarrow$  bool where
counterEvalP f = ( $\lambda p \text{ args} . ! i . \neg (EV (\text{contains } f (i, FAtom Pos p (map (X} \circ
inv obj) args))))$ )

```

definition

```

counterModel :: (nat  $\Rightarrow$  pseq)  $\Rightarrow$  model where
counterModel f = Abs-model (counterM f, counterEvalP f)

```

```

primrec counterAssign :: vbl  $\Rightarrow$  object
where counterAssign (X n) = obj n

```

5.8 subs: finite

```

lemma finite-subs: finite (subs  $\gamma$ )
by (simp add: subs-def subsFAtom-def subsFConj-def subsFAll-def split-beta split:
list.split formula.split signs.split)

```

```

lemma fansSubs: fans subs
by (simp add: fans-def finite-subs)

```

lemma subs-def2:

```

 $\neg SATAxiom (\text{sequent } \gamma) \implies$ 
subs  $\gamma$  =
(case nforms  $\gamma$  of
[]  $\Rightarrow$  {}
| nA#nAs  $\Rightarrow$  let (n,A) = nA
in (case A of
FAtom z P vs  $\Rightarrow$  subsFAtom (atoms  $\gamma$ ) nAs z P vs
| FConj z A0 A1  $\Rightarrow$  subsFConj (atoms  $\gamma$ ) nAs z A0 A1
| FAll z A  $\Rightarrow$  subsFAll (atoms  $\gamma$ ) nAs n z A (freeVarsFL
(sequent  $\gamma$ )))
by (simp add: subs-def nforms-def atoms-def split-beta split: list.split)

```

5.9 inherited: proofTree

```

lemma proofTree-def2: proofTree = ( $\lambda x. \text{bounded } x \wedge \text{founded subs } (\text{SATAxiom} \circ \text{sequent}) x$ )
  by (force simp add: proofTree-def)

lemma inheritedProofTree: inherited subs proofTree
  using inheritedBounded inheritedFounded inheritedJoinI proofTree-def by blast

lemma proofTreeI:  $\llbracket \text{bounded } A; \text{founded subs } (\text{SATAxiom} \circ \text{sequent}) A \rrbracket \implies \text{proofTree } A$ 
  by (simp add: proofTree-def)

lemma proofTreeBounded: proofTree A  $\implies$  bounded A
  using proofTree-def by blast

lemma proofTreeTerminal:
   $\llbracket \text{proofTree } A; (n, \text{delta}) \in A; \text{terminal subs delta} \rrbracket \implies \text{SATAxiom } (\text{sequent delta})$ 
  by(force simp add: proofTree-def founded-def)

```

5.10 pseq: lemma

```

lemma snd-o-Pair: (snd o (Pair x)) = ( $\lambda x. x$ )
  by auto

lemma sequent-pseq: sequent (pseq fs) = fs
  by (simp add: pseq-def sequent-def snd-o-Pair)

```

5.11 SATAxiom: proofTree

```

lemma SATAxiomTerminal[rule-format]: SATAxiom (sequent  $\gamma$ )  $\implies$  terminal subs  $\gamma$ 
  by (simp add: subs-def proofTree-def terminal-def founded-def bounded-def)

lemma SATAxiomBounded:SATAxiom (sequent  $\gamma$ )  $\implies$  bounded (tree subs  $\gamma$ )
  by (metis (lifting) boundedInsert equals0D finite.simps inheritedBounded
    inheritedUNEq subs-def treeEquation)

lemma SATAxiomFounded: SATAxiom (sequent  $\gamma$ )  $\implies$  founded subs (SATAxiom
   $\circ$  sequent) (tree subs  $\gamma$ )
  apply (clarify simp add: founded-def split: prod.splits)
  by (metis SATAxiomTerminal terminalI tree.cases tree1Eq)

lemma SATAxiomProofTree: SATAxiom (sequent  $\gamma$ )  $\implies$  proofTree (tree subs  $\gamma$ )
  by (blast intro: proofTreeI SATAxiomBounded SATAxiomFounded)

lemma SATAxiomEq: (proofTree (tree subs  $\gamma$ )  $\wedge$  terminal subs  $\gamma$ ) = SATAxiom
  (sequent  $\gamma$ )
  using SATAxiomProofTree SATAxiomTerminal proofTreeTerminal tree.tree0 by
  blast

```

5.12 SATAxioms are deductions: - needed

```

lemma SAT-deduction:
  assumes SATAxiom x
  shows x ∈ deductions CutFreePC
  proof -
    obtain n vs where FAtom Pos n vs ∈ set x and FAtom Neg n vs ∈ set x
      using SATAxiom-def assms by blast
    with inDed-mono [where x=[FAtom Pos n vs,FAtom Neg n vs]]
    show ?thesis
      by (simp add: AxiomI rulesInPCs(2))
  qed

```

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 [¬ SATAxiom (sequent (ats, (n, f) # list));
         ¬ terminal subs (ats, (n, f) # list);
         ∀ sigma ∈ subs (ats, (n, f) # list). sequent sigma ∈ deductions CutFreePC]
        ⇒ sequent (ats, (n, f) # list) ∈ deductions CutFreePC
  proof (induction f rule: formula-signs-induct)
    case (FAllP A)
    then show ?case
      by (simp add: ss PermI rulesInPCs freshVarI AllI)
  next
    case (FAllN A)
    have *: Exs ⊆ CutFreePC Contrs ⊆ CutFreePC
      using rulesInPCs by blast+
    moreover
      have instanceF (X n) A # map snd list @ FAll Neg A # map (λ(z, x, y). FAtom z x y) ats
        ∈ deductions CutFreePC
      using FAllN by (simp add: ss)
    then have instanceF (X n) A # FAll Neg A # map snd list @ map (λ(z, x, y). FAtom z x y) ats
        ∈ deductions CutFreePC
      by (simp add: PermI rulesInPCs(16))
    ultimately show ?case
      by (simp add: Contri ExI sequent-def)
  qed (simp-all add: ss PermI rulesInPCs ConjI' DisjI)

lemma subsJustified:
  assumes ¬ terminal subs γ
  and ∀ sigma ∈ subs γ. sequent sigma ∈ deductions (CutFreePC)
  shows sequent γ ∈ deductions (CutFreePC)
  proof(cases SATAxiom (sequent γ))
    case True

```

```

then show ?thesis using SAT-deduction by blast
next
  case False
  show ?thesis
  proof (cases γ)
    case (Pair ats nfs)
    show ?thesis
    proof (cases nfs)
      case Nil
      then show ?thesis
        using False Pair assms nforms-def subs-def2 terminal-def by force
    next
      case (Cons a list)
      then show ?thesis
        by (metis False Pair assms subsJustified' surj-pair)
    qed
  qed
qed

```

5.14 proofTrees are deductions: instance of boundedTreeInduction

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

```

lemma proofTreeDeductionD:
  assumes proofTree(tree subs γ)
  shows sequent γ ∈ deductions (CutFreePC)
  proof (rule boundedTreeInduction [OF fansSubs])
    show bounded (tree subs γ)
      by (simp add: assms proofTreeBounded)
  next
    show founded subs (λa. sequent a ∈ deductions CutFreePC) (tree subs γ)
      by (metis (no-types, lifting) SAT-deduction assms comp-def foundedMono
          proofTreeD)
  qed (use subsJustified in auto)

```

5.15 contains, considers:

```
lemma contains-def2: contains f iA n = (iA ∈ set (nforms (f n)))
  using Completeness.contains-def nforms-def by presburger
```

```
lemma considers-def2: considers f iA n = ( ∃ nAs . nforms (f n) = iA#nAs)
  by (simp add: considers-def nforms-def split: list.split)
```

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

5.16 path: nforms = [] implies

```
lemma nformsNoContains:
  nforms (f n) = [] ==> ¬ contains f iA n
```

```

by (simp add: contains-def2)

lemma nformsTerminal: nforms (f n) = [] ==> terminal subs (f n)
  by (simp add: subs-def Let-def terminal-def nforms-def split-beta)

lemma nformsStops:
  [branch subs γ f; ∀n. ¬ proofTree (tree subs (f n)); nforms (f n) = []]
  ==> nforms (f (Suc n)) = [] ∧ atoms (f (Suc n)) = atoms (f n)
  by (metis (lifting) SATAxiomProofTree branch-def empty-iff list.case(1)
       subs-def2)

```

5.17 path: cases

```

lemma terminalNFormCases: terminal subs (f n) ∨ (∃ i A nAs . nforms (f n) =
(i,A)#nAs)
  by (metis (lifting) list.case(1) neq-Nil-conv subs-def subs-def2 surj-pair
      terminal-def)

lemma cases[elim-format]: terminal subs (f n) ∨ (¬ (terminal subs (f n) ∧ (∃ i A
nAs . nforms (f n) = (i,A)#nAs)))
  by simp

```

5.18 path: contains not terminal and propagate condition

```

lemma containsNotTerminal:
  assumes branch subs γ f ¬ proofTree (tree subs (f n)) contains f iA n
  shows ¬ terminal subs (f n)
proof (cases SATAxiom (sequent (f n)))
  case True
  then show ?thesis
    using SATAxiomEq assms by blast
next
  case False
  then show ?thesis
    using assms
    by (auto simp: subs-def subsFAtom-def subsFConj-def subsFAll-def contains-def
        terminal-def split-beta
        split: list.split signs.split formula.splits)
qed

lemma containsPropagates:
  assumes branch subs γ f
  and ∀n. ¬ proofTree (tree subs (f n))
  and contains f iA n
  shows contains f iA (Suc n) ∨ considers f iA n
proof -
  have §: f (Suc n) ∈ subs (f n)
    by (meson assms branchSubs containsNotTerminal)
  then show ?thesis
proof (cases considers f iA n)

```

```

case True
then show ?thesis
by blast
next
case ncons: False
show ?thesis
proof (cases SATAxiom (sequent (f n)))
case True
then show ?thesis
using SATAxiomEq assms by blast
next
case False
with assms § show ?thesis
by (auto simp: subs-def2 contains-def considers-def2 nforms-def
      subsFAtom-def subsFConj-def subsFAll-def
      split: list.splits formula.splits signs.splits)
qed
qed
qed

```

5.19 termination: (for EV contains implies EV considers)

```

lemma terminationRule [rule-format]:
! n. P n —> (¬(P (Suc n)) ∣ (P (Suc n) ∧ msrFn (Suc n) < (msrFn::nat ⇒
nat) n)) —> P m —> (∃ n . P n ∧ ¬(P (Suc n)))
(is - —> ?P m)
using measure-induct-rule[of msrFn ?P m]
by blast

```

5.20 costBarrier: lemmas

5.21 costBarrier: exp3 lemmas - bit specific...

```

lemma exp1: power3 A + power3 B < 3 * (power3 A * power3 B)
by (induction A) (use less-2-cases not-less-eq in fastforce) +

```

```

lemma exp1': power3 A < 3 * ((power3 A) * (power3 B)) + C
by (meson add-lessD1 exp1 trans-less-add1)

```

```

lemma exp2: Suc 0 < 3 * power3 (B)
by (simp add: Suc-lessI)

```

5.22 costBarrier: decreases whilst contains and unconsiders

```

lemma costBarrierDecreases':
notes ss = subs-def2 nforms-def subsFAtom-def subsFConj-def subsFAll-def cost-
Barrier-def
shows ⟦¬ SATAxiom (sequent (a, (num, fm) # list)); iA ≠ (num, fm);
¬ proofTree (tree subs (a, (num, fm) # list)); ⟦

```

```

 $fSucn \in subs(a, (num, fm) \# list); iA \in set list]$ 
 $\implies costBarrier iA fSucn < costBarrier iA (a, (num, fm) \# list)$ 
proof (induction fm rule: formula-signs-induct)
  case (FConjP A B)
  then show ?case
    by (auto simp: ss exp2 power-add)
next
  case (FConjN A B)
  with exp1' show ?case
    by (simp add: ss exp1 power-add)
qed (auto simp: ss size-instance)

```

lemma *costBarrierDecreases*:

assumes *branch subs* γf

and $\bigwedge n. \neg proofTree(tree subs(f n))$

and *contains f iA n*

and $\neg considers f iA n$

shows *costBarrier iA (f (Suc n)) < costBarrier iA (f n)*

proof –

have 1: $\neg terminal subs(f n)$

using *assms containsNotTerminal* **by** *blast*

have $\neg SATAxiom(sequent(f n))$

using *SATAxiomTerminal 1* **by** *blast*

moreover have *f (Suc n) ∈ subs (f n)*

by (*meson assms(1) branchSubs 1*)

ultimately show ?thesis

using *assms*

unfolding *contains-def considers-def2 nforms-def*

by (*metis costBarrierDecreases' eq-snd-iff list.set-cases*)

qed

5.23 path: EV contains implies EV considers

```

lemma considersContains: considers f iA n  $\implies$  contains f iA n
  using considers-def2 contains-def2 by auto

lemma containsConsiders:


assumes branch subs  $\gamma f$



and  $\bigwedge n. \neg proofTree(tree subs(f n))$



and EV (contains f iA)



shows EV (considers f iA)



proof (rule ccontr)



assume non:  $\neg EV(considers f iA)$



then obtain n where contains f iA n  $\wedge$   $\neg considers f iA n$



using EV-def assms by fastforce



then have  $\exists n'. (contains f iA n' \wedge \neg considers f iA n') \wedge$



$\neg (contains f iA (Suc n') \wedge \neg considers f iA (Suc n'))$



using assms costBarrierDecreases



by (intro terminationRule [where msrFn= λn. costBarrier iA (f n)]; blast)


```

```

with assms show False
  by (metis EV-def non containsPropagates)
qed

```

5.24 EV contains: common lemma

```

lemma lemmA:
  assumes branch subs  $\gamma f$ 
  and  $\bigwedge n. \neg proofTree (tree subs (f n))$ 
  and  $EV (contains f (i, A))$ 
  obtains  $n\ nAs$  where  $\neg SATAxiom (sequent (f n))$ 
    nforms  $(f n) = (i, A) \# nAs f (Suc n) \in subs (f n)$ 
proof –
  obtain  $n$  where  $n: considers f (i, A) n contains f (i, A) n$ 
    by (metis EV-def assms considersContains containsConsiders)
  with assms that show ?thesis
    by (metis SATAxiomTerminal considers-def2 branchSubs containsNotTerminal)
qed

```

5.25 EV contains: FConj,FDisj,FAll

```

lemma evContainsConj:
  assumes  $EV (contains f (i, FConj Pos A0 A1))$ 
  and branch subs  $\gamma f$ 
  and  $\bigwedge n. \neg proofTree (tree subs (f n))$ 
  shows  $EV (contains f (0, A0)) \vee EV (contains f (0, A1))$ 
proof –
  obtain  $n\ nAs$  where  $\neg SATAxiom (sequent (f n))$ 
    nforms  $(f n) = (i, FConj Pos A0 A1) \# nAs f (Suc n) \in subs (f n)$ 
    by (metis assms lemmA)
  with assms have  $EV (\lambda n. contains f (0, A0) n \vee contains f (0, A1) n)$ 
    apply (simp add: EV-def contains-def subsFConj-def subs-def2)
    by (metis list.set-intros(1) snd-conv)
  then show ?thesis
    using EV-def by fastforce
qed

```

```

lemma evContainsDisj:
  assumes  $EV (contains f (i, FConj Neg A0 A1))$ 
  and branch subs  $\gamma f$ 
  and  $\bigwedge n. \neg proofTree (tree subs (f n))$ 
  shows  $EV (contains f (0, A0)) \wedge EV (contains f (0, A1))$ 
proof –
  obtain  $n\ nAs$  where  $\neg SATAxiom (sequent (f n))$ 
    nforms  $(f n) = (i, FConj Neg A0 A1) \# nAs f (Suc n) \in subs (f n)$ 
    by (metis assms lemmA)
  with assms have  $EV (\lambda n. contains f (0, A0) n \wedge contains f (0, A1) n)$ 

```

```

apply (simp add: EV-def contains-def subsFConj-def subs-def2)
by (metis list.set-intros snd-conv)
then show ?thesis
using EV-def by fastforce
qed

lemma evContainsAll:
assumes EV (contains f (i,FAll Pos body))
and branch subs γ f
and ⋀n. ¬ proofTree (tree subs (f n))
shows ∃v . EV (contains f (0,instanceF v body))
proof -
obtain n nAs where ¬ SATAxiom (sequent (f n))
nforms (f n) = (i, FAll Pos body) # nAs f (Suc n) ∈ subs (f n)
by (metis assms lemmaA)
then have (0, instanceF (freshVar (freeVarsFL (sequent (f n)))) body)
∈ set (snd (f (Suc n)))
by (simp add: contains-def subsFAll-def subs-def2)
then show ?thesis
unfolding EV-def
by (metis containsI nforms-def)
qed

lemma evContainsEx-instance:
assumes EV (contains f (i,FAll Neg body))
and branch subs γ f
and ⋀n. ¬ proofTree (tree subs (f n))
shows EV (contains f (0,instanceF (X i) body))
proof -
obtain n nAs where ¬ SATAxiom (sequent (f n))
nforms (f n) = (i, FAll Neg body) # nAs f (Suc n) ∈ subs (f n)
by (metis assms lemmaA)
then have contains f (0, instanceF (X i) body) (Suc n)
by (simp add: containsI nforms-def subsFAll-def subs-def2)
then show ?thesis
unfolding EV-def
by metis
qed

lemma evContainsEx-repeat:
assumes EV (contains f (i,FAll Neg body))
and branch subs γ f
and ⋀n. ¬ proofTree (tree subs (f n))
shows EV (contains f (Suc i,FAll Neg body))
proof -
obtain n nAs where n: ¬ SATAxiom (sequent (f n))
nforms (f n) = (i, FAll Neg body) # nAs f (Suc n) ∈ subs (f n)
by (metis assms lemmaA)
then have ⟦f (Suc n) = (atoms (f n), (0, instanceF (X i) body) # nAs @ [(Suc

```

```

i, FAll Neg body)])]】
 $\implies \exists n. (Suc i, FAll Neg body) \in set (snd (f n))$ 
by (metis in-set-conv-decomp list.set-intros(2) snd-conv)
with assms n show ?thesis
by (simp add: EV-def contains-def2 nforms-def subsFAll-def subs-def2)
qed

```

5.26 EV contains: lemmas (temporal related)

5.27 EV contains: FAtoms

```

lemma notTerminalNotSATAxiom:  $\neg \text{terminal} \text{ subs } \gamma \implies \neg \text{SATAxiom} \text{ (sequent } \gamma)$ 
using SATAxiomTerminal by blast

```

```

lemma notTerminalNforms:  $\neg \text{terminal} \text{ subs } (f n) \implies \text{nforms } (f n) \neq []$ 
by (metis (lifting) list.simps(4) subs-def subs-def2 terminal-def)

```

```

lemma atomsPropagate:
assumes f: branch subs  $\gamma$  f and x:  $x \in set (\text{atoms } (f n))$ 
shows x  $\in set (\text{atoms } (f (\text{Suc } n)))$ 
proof (cases terminal subs  $(f n)$ )
case True
then show ?thesis
by (metis assms branchStops)
next
case nonterm: False
then have  $\S: f (\text{Suc } n) \in \text{subs } (f n)$ 
by (meson branchSubs f)
show ?thesis
proof (cases nforms  $(f n)$ )
case Nil
then show ?thesis
by (metis (lifting) § empty-iff list.case(1) subs-def subs-def2)
next
case (Cons nfm list)
with x nonterm  $\S$  show ?thesis
by (force simp: subs-def2 atoms-def notTerminalNotSATAxiom
      subsFAtom-def subsFConj-def subsFAll-def
      split: signs.splits formula.splits)
qed
qed

```

5.28 EV contains: FEx cases

```

lemma evContainsEx0-allRepeats:
 $[\![\text{branch} \text{ subs } \gamma \text{ f}; \bigwedge n . \neg \text{proofTree } (\text{tree} \text{ subs } (f n));$ 
 $EV (\text{contains } f (0, FAll Neg A))]\!]$ 
 $\implies EV (\text{contains } f (i, FAll Neg A))$ 
by (induction i) (simp-all add: evContainsEx-repeat)

```

```

lemma evContainsEx0-allInstances:
   $\llbracket \text{branch subs } \gamma f; \bigwedge n. \neg \text{proofTree}(\text{tree subs}(f n));$ 
     $\text{EV}(\text{contains } f(0, \text{FAll Neg } A)) \rrbracket$ 
   $\implies \text{EV}(\text{contains } f(0, \text{instanceF}(X i) A))$ 
  using evContainsEx0-allRepeats evContainsEx-instance by blast

```

5.29 pseq: creates initial pseq

```

lemma containsPSeq0D: branch subs (pseq fs) f  $\implies$  contains f (i, A) 0  $\implies$  i=0
  by (simp add: branch-def contains-def2 image-iff nforms-def pseq-def)

```

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

```

lemma natPredCases:
  obtains  $\forall n. P n \mid \neg P 0 \mid n$  where  $P n \neg P (\text{Suc } n)$ 
  using nat-induct by auto

lemma containsNotTerminal':
   $\llbracket \text{branch subs } \gamma f; \bigwedge n. \neg \text{proofTree}(\text{tree subs}(f n)); \text{contains } f(iA n) \rrbracket \implies \neg$ 
  ( $\text{terminal subs}(f n)$ )
  by (simp add: containsNotTerminal)

lemma evContainsExSuc-containsEx:
  assumes branch subs (pseq fs) f
  and  $\bigwedge n. \neg \text{proofTree}(\text{tree subs}(f n))$ 
  and  $\text{EV}(\text{contains } f(\text{Suc } i, \text{FAll Neg body}))$ 
  shows  $\text{EV}(\text{contains } f(i, \text{FAll Neg body}))$ 
  using natPredCases [of  $\lambda n. \neg \text{contains } f(\text{Suc } i, \text{FAll Neg body}) n$ ]
proof cases
  case 1
  with assms show ?thesis
  by (force simp: EV-def)
next
  case 2
  then show ?thesis
  using assms(1) containsPSeq0D by blast
next
  case (? n)
  with assms have nonterm:  $\neg \text{terminal subs}(f n)$ 
  by (metis branchStops containsNotTerminal')
  then have f:  $f(\text{Suc } n) \in \text{subs}(f n)$ 
  by (meson assms(1) branchSubs)
  have considers f (i, FAll Neg body) n
  proof (cases SATAxiom (sequent (f n)))
  case True
  then show ?thesis
  using SATAxiomTerminal nonterm by blast

```

```

next
  case False
    then obtain j A nAs where i: snd (f n) = (j, A) # nAs
      by (metis (lifting) nforms-def empty-iff f list.exhaust list.simps(4) subs-def2
           surj-pair)
    then have nf: nforms (f n) ≠ []
      by (simp add: nforms-def)
    have snd (f n) = (k, A) # nAs → k = i ∧ A = FAll Neg body for k A
      apply(induction A rule: formula-signs-induct)
      using i 3 f False
      by(auto simp: subs-def2 nforms-def subsFAtom-def subsFConj-def subsFAll-def
           contains-def2)
    with nf show ?thesis
      by (auto simp add: considers-def i split: list.splits formula.splits signs.splits)
  qed
  then show ?thesis
    by (meson EV-def considersContains)
qed

```

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

```

lemma evContainsEx-containsEx0:
   $\llbracket \text{branch } \text{subs } (\text{pseq } fs) f; \bigwedge n. \neg \text{proofTree } (\text{tree } \text{subs } (f n));$ 
   $\quad \text{EV } (\text{contains } f (i, \text{FAll Neg } A)) \rrbracket$ 
   $\implies \text{EV } (\text{contains } f (0, \text{FAll Neg } A))$ 
proof (induction i)
  case 0
    then show ?case by simp
  next
    case (Suc i)
    then show ?case
      using evContainsExSuc-containsEx by blast
qed

lemma evContainsExval:
   $\llbracket \text{EV } (\text{contains } f (i, \text{FAll Neg body})); \text{branch } \text{subs } (\text{pseq } fs) f;$ 
   $\quad \bigwedge n. \neg \text{proofTree } (\text{tree } \text{subs } (f n)) \rrbracket$ 
   $\implies \text{EV } (\text{contains } f (0, \text{instanceF } v \text{ body}))$ 
  by (induction v) (simp add: evContainsEx0-allInstances evContainsEx-containsEx0)

```

5.32 EV contains: atoms

```

lemma atomsInSequentI:
   $(z, P, vs) \in \text{set } (\text{fst } ps) \implies \text{FAtom } z P vs \in \text{set } (\text{sequent } ps)$ 
  by (fastforce simp add: sequent-def fst-def split: prod.split)

lemma evContainsAtom1:
  assumes branch subs (pseq fs) f

```

```

and  $\bigwedge n. \neg proofTree (tree subs (f n))$ 
and  $EV (contains f (i, FAtom z P vs))$ 
shows  $\exists n. (z, P, vs) \in set (fst (f n))$ 
proof –
  obtain  $n nAs$  where  $\neg SATAxiom (sequent (f n))$ 
     $nforms (f n) = (i, FAtom z P vs) \# nAs$ 
     $f (Suc n) \in subs (f n)$ 
  by (metis assms lemmaA)
  then have  $(z, P, vs) \in set (fst (f (Suc n)))$ 
  by (simp add: subsFAtom-def subs-def2)
  then show ?thesis ..
qed

```

lemmas atomsPropagate' = atomsPropagate[simplified atoms-def, simplified]

```

lemma evContainsAtom:
  assumes branch subs (pseq fs) f
  and  $\bigwedge n. \neg proofTree (tree subs (f n))$ 
  and  $EV (contains f (i, FAtom z P vs))$ 
  shows  $\exists n. \forall m. FAtom z P vs \in set (sequent (f (n + m)))$ 
proof –
  obtain  $n$  where  $n: (z, P, vs) \in set (fst (f n))$ 
  using assms evContainsAtom1 by blast
  then have  $(z, P, vs) \in set (fst (f (n + m)))$  for  $m$ 
  by (induction m) (use assms atomsPropagate' in auto)
  then show ?thesis
  using atomsInSequentI by blast
qed

```

```

lemma notEvContainsBothAtoms:
   $\llbracket \text{branch subs (pseq fs) } f; \bigwedge n. \neg proofTree (tree subs (f n)) \rrbracket$ 
   $\implies \neg EV (contains f (i, FAtom Pos p vs)) \vee$ 
   $\quad \neg EV (contains f (j, FAtom Neg p vs))$ 
  by (metis SATAxiomProofTree SATAxiom-def add.commute evContainsAtom)

```

5.33 counterModel: lemmas

```

lemma counterModelInRepn: (counterM f, counterEvalP f)  $\in$  model
  by (simp add: model-def counterM-def)

```

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

```

lemma inv-obj-obj: inv obj (obj n) = n
  using inj-obj by simp

```

```

lemma map-X-map-counterAssign [simp]: map X (map (inv obj) (map counterAssign xs)) = xs
proof –
  have  $(X \circ (inv obj \circ counterAssign)) = (\lambda x . x)$ 

```

```

    by (metis X-deX comp-apply counterAssign.simps inv-obj-obj)
then show ?thesis
  by simp
qed

lemma objectsCounterModel: objects (counterModel f) = { z . ∃ i . z = obj i }
  unfolding objects-def counterModel-def Abs-counterModel-inverse
  by(force simp add: counterM-def)

lemma inCounterM: counterAssign v ∈ objects (counterModel f)
  by (induction v) (force simp add: objectsCounterModel)

lemma evalPCounterModel: M = counterModel f ==> evalP M = counterEvalP f
  by (simp add: evalP-def counterModel-def Abs-counterModel-inverse)

```

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

```

lemma path-evalF:
assumes branch subs (pseq fs) f ∧ n. ¬ proofTree (tree subs (f n))
  shows (∃ i . EV (contains f (i,A))) ==> ¬ evalF (counterModel f) counterAssign A
proof (induction A rule: measure-induct-rule [where f = size])
  case (less A)
  show ?case
    using less
  proof (induction A rule: formula-signs-induct)
    case (FAtomP p vs)
    with assms show ?case
      apply (simp add: evalPCounterModel counterEvalP-def list.map-comp)
      by (metis list.map-comp map-X-map-counterAssign)
  next
    case (FAtomN p vs)
    with assms show ?case
      apply (simp add: evalPCounterModel counterEvalP-def list.map-comp)
      by (metis list.map-comp map-X-map-counterAssign notEvContainsBothAtoms)
  next
    case (FConjP A B)
    with assms show ?case
      apply (simp add: evalPCounterModel counterEvalP-def list.map-comp)
      by (meson evContainsConj less-add-Suc1 less-add-Suc2)
  next
    case (FConjN A B)
    with assms show ?case
      apply (clarify simp add: evalPCounterModel counterEvalP-def list.map-comp)
      using evContainsDisj less-add-Suc1 less-add-Suc2 by blast
  next
    case (FAllP A)
    with assms obtain v where EV (contains f (0, instanceF v A))

```

```

using evContainsAll by blast
with FAllP.prems show ?case
  by (metis evalF-FAll evalF-instance inCounterM size-instance
       sizelemmas(3))
next
  case (FAllN A)
  then obtain i where ¬ evalF (counterModel f) counterAssign
    (instanceF (X i) A)
  by (metis assms evContainsExval size-instance sizelemmas(3))
  with assms show ?case
    by (smt (verit, best) FAllN.prems counterAssign.simps evContainsExval
        evalF-FEx
        evalF-instance mem-Collect-eq objectsCounterModel size-instance
        sizelemmas(3))
qed
qed

```

5.35 adequacy

```

lemma counterAssignModelAssign: counterAssign ∈ modelAssigns (counterModel
γ)
  by (simp add: inCounterM subset-eq)

lemma branch-contains-initially: branch subs (pseq fs) f ==> x ∈ set fs ==> con-
tains f (0,x) 0
  by (simp add: contains-def branch0 pseq-def)

lemma validProofTree:
  assumes ¬ proofTree (tree subs (pseq fs))
  shows ¬ validS fs
proof -
  obtain f where f: branch subs (pseq fs) f ∧ n. ¬ proofTree (tree subs (f n))
  by (metis assms failingBranchExistence fansSubs inheritedProofTree)
  then show ?thesis
  by (metis EV-def branch-contains-initially counterAssignModelAssign evalS-def
       path-evalF validS-def)
qed

```

```

theorem adequacy[simplified sequent-pseq]: validS fs ==> (sequent (pseq fs)) ∈ de-
ductions CutFreePC
  using proofTreeDeductionD validProofTree by blast
end

```

6 Soundness

```

theory Soundness imports Completeness begin

```

```

lemma permutation-validS: mset fs = mset gs ==> (validS fs = validS gs)

```

```

unfolding validS-def evalS-def
using mset-eq-setD by blast

lemma modelAssigns-vblcase:  $\varphi \in \text{modelAssigns } M \implies x \in \text{objects } M \implies \text{vblcase}$ 
 $x \varphi \in \text{modelAssigns } M$ 
unfolding modelAssigns-def mem-Collect-eq
by (smt (verit) image-subset-iff mem-Collect-eq range-subsetD vbl-cases
vblcase-nextX vblcase-zeroX)

lemma soundnessFAll:
assumes  $u \notin \text{freeVarsFL} (\text{FAll } \text{Pos } A \# \Gamma)$ 
and validS (instanceF  $u A \# \Gamma$ )
shows validS ( $\text{FAll } \text{Pos } A \# \Gamma$ )
unfolding validS-def
proof (intro strip)
fix  $M \varphi$ 
assume  $\varphi: \varphi \in \text{modelAssigns } M$ 
have evalF  $M (\text{vblcase } x \varphi) A$ 
if  $x: x \in \text{objects } M$  and  $\neg \text{evalS } M \varphi \Gamma$ 
for  $x$ 
proof –
have evalF  $M (\text{vblcase } x (\lambda y. \text{if } y = u \text{ then } x \text{ else } \varphi y)) A$ 
proof –
have evalF  $M (\text{vblcase } x (\lambda y. \text{if } y = u \text{ then } x \text{ else } \varphi y)) A$ 
 $\vee \text{evalS } M (\lambda y. \text{if } y = u \text{ then } x \text{ else } \varphi y) \Gamma$ 
using  $\varphi$  assms(2) evalF-instance image-subset-iff validS-def  $x$  by fastforce
then show ?thesis
using assms(1) equalOn-def evalS-equiv freeVarsFL-cons that(2) by fastforce
qed
moreover
have equalOn (freeVarsF  $A$ ) ( $\text{vblcase } x (\lambda y. \text{if } y = u \text{ then } x \text{ else } \varphi y)$ )
( $\text{vblcase } x \varphi$ )
by (smt (verit, best) Un-iff assms(1) equalOn-def equalOn-vblcaseI' freeVars-
FAll
freeVarsFL-cons)
ultimately show ?thesis
using evalF-equiv by force
qed
then show evalS  $M \varphi (\text{FAll } \text{Pos } A \# \Gamma) = \text{True}$ 
by auto
qed

lemma soundnessFEx: validS (instanceF  $x A \# \Gamma$ )  $\implies$  validS ( $\text{FAll } \text{Neg } A \# \Gamma$ )
unfolding validS-def
by (metis evalF-FEx evalF-instance evalS-cons modelAssigns-iff range-subsetD)

lemma soundnessFCut:  $\llbracket \text{validS } (C \# \Gamma); \text{validS } (\text{FNot } C \# \Delta) \rrbracket \implies$ 
validS ( $\Gamma @ \Delta$ )

```

```

using evalF-FNot evalS-append validS-def by auto

lemma soundness:  $fs : deductions(PC) \implies validS fs$ 
proof (induction  $fs$  rule: deductions.induct)
  case (inferI conc prems)
    then have  $vS: \forall x \in prems. validS x$ 
      by auto
    have  $validS conc$ 
      if  $\S: (conc, prems) \in Perms$ 
      proof –
        obtain  $\Delta$  where  $\Delta: prems = \{\Delta\}$ 
          using  $\S vS$  by (auto simp: Perms-def)
        then have  $mset conc = mset \Delta$ 
          using Perms-def that by fastforce
        with  $\Delta$  permutation-validS vS show ?thesis
          by blast
      qed
    moreover have  $validS conc$ 
      if  $(conc, prems) \in Axioms$ 
        using that by (auto simp add: Axioms-def validS-def)
    moreover have  $validS conc$ 
      if  $(conc, prems) \in Conjs$ 
        using that vS inferI apply (simp add: validS-def Conjs-def)
        by (metis evalFCConj evalS-append evalS-cons insertCI sign.simps(1))
    moreover have  $validS conc$ 
      if  $(conc, prems) \in Disjs$ 
        using that vS inferI by (fastforce simp add: validS-def Disjs-def)
    moreover have  $validS conc$ 
      if  $(conc, prems) \in Alls$ 
        using that vS inferI soundnessFAll
        by (force simp add: validS-def Alls-def subset-iff)
    moreover have  $validS conc$ 
      if  $(conc, prems) \in Exs$ 
        using that vS inferI soundnessFEx
        by (force simp add: validS-def Exs-def subset-iff)
    moreover have  $validS conc$ 
      if  $(conc, prems) \in Weaks$ 
        using that vS by (force simp: Weaks-def validS-def evalS-def)
    moreover have  $validS conc$ 
      if  $(conc, prems) \in Contrs$ 
        using that vS by (fastforce simp add: Contrs-def validS-def evalS-def)
    moreover have  $validS conc$ 
      if  $(conc, prems) \in Cuts$ 
        using that vS by (force simp add: Cuts-def soundnessFCut)
    ultimately show ?case
      using inferI.hyps
      by (auto simp: PC-def subset-iff)
  qed

```

theorem completeness: $fs \in deductions(PC) \longleftrightarrow validS fs$
using adequacy mono-deductions rulesInPCs(18) soundness **by** blast

end