

Completeness for FOL

James Margetson, ported by Tom Ridge

December 12, 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	5
1.4	Following used in Soundness	6
2	Base	7
2.1	Integrate with Isabelle libraries?	7
2.2	Summation	7
2.3	Termination Measure	8
2.4	Functions	8
3	Formula	9
3.1	Variables	9
3.2	Predicates	11
3.3	Formulas	12
3.4	formula signs induct, formula signs cases	12
3.5	Frees	14
3.6	Substitutions	15
3.7	Models	15
3.8	model, non empty set and positive atom valuation	15
3.9	Validity	16
4	Sequents	18
4.1	Rules	18
4.2	Deductions	19
4.3	Basic Rule sets	19
4.4	Derived Rules	21
4.5	Standard Rule Sets For Predicate Calculus	22
4.6	Monotonicity for CutFreePC deductions	22

4.7	Tree	23
4.8	Terminal	24
4.9	Inherited	25
4.10	bounded, boundedBy	27
4.11	Inherited Properties- bounded	28
4.12	founded	29
4.13	Inherited Properties- founded	29
4.14	Inherited Properties- finite	30
4.15	path: follows a failing inherited property through tree	30
4.16	Branch	31
4.17	failing branch property: abstracts path defn	32
4.18	Tree induction principles	33
5	Completeness	34
5.1	pseq: type represents a processed sequent	34
5.2	subs: SATAxiom	35
5.3	subs: a CutFreePC justifiable backwards proof step	35
5.4	proofTree(Gamma) says whether tree(Gamma) is a proof	35
5.5	path: considers, contains, costBarrier	36
5.6	path: eventually	36
5.7	path: counter model	36
5.8	subs: finite	36
5.9	inherited: proofTree	37
5.10	pseq: lemma	37
5.11	SATAxiom: proofTree	38
5.12	SATAxioms are deductions: - needed	38
5.13	proofTrees are deductions: subs respects rules - messy start and end	39
5.14	proofTrees are deductions: instance of boundedTreeInduction	39
5.15	contains, considers:	40
5.16	path: nforms = [] implies	40
5.17	path: cases	40
5.18	path: contains not terminal and propagate condition	41
5.19	path: no consider lemmas	41
5.20	path: contains initially	41
5.21	termination: (for EV contains implies EV considers)	42
5.22	costBarrier: lemmas	42
5.23	costBarrier: exp3 lemmas - bit specific...	42
5.24	costBarrier: decreases whilst contains and unconsiders	43
5.25	path: EV contains implies EV considers	44
5.26	EV contains: common lemma	44
5.27	EV contains: FConj,FDisj,FAI	45
5.28	EV contains: lemmas (temporal related)	46
5.29	EV contains: FAToms	47

5.30	EV contains: FEx cases	47
5.31	pseq: creates initial pseq	47
5.32	EV contains: contain any (i,FEx y) means contain all (i,FEx y)	48
5.33	EV contains: contain any (i,FEx y) means contain all (i,FEx y)	49
5.34	EV contains: atoms	49
5.35	counterModel: lemmas	50
5.36	counterModel: all path formula value false - step by step . . .	51
5.37	adequacy	52

6 Soundness 53

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 $x \dot{\sim} y$ 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)
```

```
  by(induct set: perm) auto
```

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

```
  by(induct B) auto
```

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

```
  apply(induct B)
```

```
  apply auto
```

```
  apply(case-tac a = aa)
```

```
  apply auto
```

```
done
```

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

```
  by(induct xs) auto
```

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

```
  apply(induct A)
```

```
  apply(simp add: count-0)
```

proof –
fix a list B
assume $a: \bigwedge B. \forall x. \text{count } x \text{ list} = \text{count } x B \implies \text{list } \langle \sim \sim \rangle B$
and $b: \forall x. \text{count } x (a \# \text{list}) = \text{count } x B$
from b **have** $a : \text{set } B$
apply *auto*
apply (*drule-tac* $x=a$ **in** *spec, simp*) **apply** (*metis count-Suc*) **done**
from *split-list[OF this]* **obtain** $xs\ ys$ **where** $B: B = xs@a\#ys$ **by** *blast*
let $?B' = xs@ys$
from b **have** $\forall x. \text{count } x \text{ list} = \text{count } x ?B'$ **by** (*simp add: count-append B*)
from a [*OF this*] **have** $c: \text{list } \langle \sim \sim \rangle xs@ys$.
hence $a\#\text{list } \langle \sim \sim \rangle a\#(xs@ys)$ **by** *rule*
also **have** $a\#(xs@ys) \langle \sim \sim \rangle xs@a\#ys$ **by** (*rule perm-append-Cons*)
also (*perm.trans*) **note** B [*symmetric*]
finally **show** $a \# \text{list } \langle \sim \sim \rangle B$.
qed

lemma *perm-count-conv*: $A \langle \sim \sim \rangle B = (\forall x. \text{count } x A = \text{count } x B)$
apply (*blast intro!: perm-count count-perm*) **done**

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

lemma *remdups-append*: $y : \text{set } ys \dashrightarrow \text{remdups } (ws@y\#ys) = \text{remdups } (ws@ys)$
apply (*induct ws, simp*)
apply (*case-tac y = a, simp, simp*)
done

lemma *perm-contr'*: **assumes** *perm[rule-format]*: $! xs\ ys. xs \langle \sim \sim \rangle ys \dashrightarrow (P\ xs = P\ ys)$
and *contr'[rule-format]*: $! x\ xs. P(x\#x\#xs) = P(x\#xs)$
shows $! xs. \text{length } xs = n \dashrightarrow (P\ xs = P(\text{remdups } xs))$
apply (*induct n rule: nat-less-induct*)

proof (*safe*)
fix $xs :: 'a\ \text{list}$
assume a [*rule-format*]: $\forall m < \text{length } xs. \forall ys. \text{length } ys = m \longrightarrow P\ ys = P(\text{remdups } ys)$
show $P\ xs = P(\text{remdups } xs)$
proof (*cases distinct xs*)
case *True*
thus $?thesis$ **by** (*simp add: distinct-remdups-id*)
next
case *False*
from *not-distinct-decomp[OF this]* **obtain** $ws\ ys\ zs\ y$ **where** $xs: xs = ws@[y]@ys@[y]@zs$
by *force*
have $P\ xs = P(ws@[y]@ys@[y]@zs)$ **by** (*simp add: xs*)
also **have** $\dots = P([y,y]@ws@ys@zs)$
apply (*rule perm*) **apply** (*rule iffD2[OF perm-count-conv]*) **apply** *rule* **apply** (*simp add: count-append*) **done**

```

    also have ... = P ([y]@ws@ys@zs) apply simp apply(rule contr') done
    also have ... = P (ws@ys@[y]@zs)
      apply(rule perm) apply(rule iffD2[OF perm-count-conv]) apply rule ap-
ply(simp add: count-append) done
    also have ... = P (remdups (ws@ys@[y]@zs))
      apply(rule a) by(auto simp: xs)
    also have (remdups (ws@ys@[y]@zs)) = (remdups xs)
      apply(simp add: xs remdups-append) done
    finally show P xs = P (remdups xs) .
  qed
qed

```

```

lemma perm-contr: assumes perm: ! xs ys. xs <~~> ys --> (P xs = P ys)
and contr': ! x xs. P(x#x#xs) = P (x#xs)
shows (P xs = P (remdups xs))
apply(rule perm-contr'[OF perm contr', rule-format]) by force

```

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 ~: 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 ~: set xs ==> rem x xs = xs
apply(simp add: rem-def)
apply(rule filter-True)
apply force
done

```

```

lemma perm-weak-filter': assumes perm[rule-format]: ! xs ys. xs <~~> ys -->
(P xs = P ys)
and weak[rule-format]: ! x xs. P xs --> P (x#xs)
shows ! ys. P (ys@filter Q xs) --> P (ys@xs)
apply (induct xs, simp, rule)
apply rule
apply simp
apply (case-tac Q a, simp)
apply (drule-tac x=ys@[a] in spec) apply simp
apply simp
apply (drule-tac x=ys@[a] in spec) apply simp
apply (erule impE)

```

```

apply(subgoal-tac (ys @ a # filter Q xs) <~~> a#ys@filter Q xs)
apply(simp add: perm)
apply(rule weak) apply simp
apply(rule perm-sym) apply(rule perm-append-Cons)

```

```

lemma perm-weak-filter: assumes perm: ! xs ys. xs <~~> ys --> (P xs = P
ys)
and weak: ! x xs. P xs --> P (x#xs)
shows P (filter Q xs) ==> P xs
using perm-weak-filter'[OF perm weak, rule-format, of [], simplified]
by blast

```

— right, now in a position to prove that in presence of perm, contr and weak, set x leq set y and x : ded implies y : ded

```

lemma perm-weak-contr-mono:
assumes perm: ! xs ys. xs <~~> ys --> (P xs = P ys)
and contr: ! x xs. P (x#x#xs) --> P (x#xs)
and weak: ! x xs. P xs --> P (x#xs)
and xy: set x <= set y
and Px : P x
shows P y
proof –
from contr weak have contr': ! x xs. P(x#x#xs) = P (x#xs) by blast

def y' == filter (% z. z : set x) y
from xy have set x = set y' apply(simp add: y'-def) apply blast done
hence rxy': remdups x <~~> remdups y' by(simp add: perm-remdups-iff-eq-set)

from Px perm-contr[OF perm contr'] have Prx: P (remdups x) by simp
with rxy' have P (remdups y') by(simp add: perm)

with perm-contr[OF perm contr'] have P y' by simp
thus P y
apply(simp add: y'-def)
apply(rule perm-weak-filter[OF perm weak]) .
qed

```

1.4 Following used in Soundness

```

primrec multiset-of-list :: 'a list => '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
by (induct A) simp-all

```

```

lemma perm-multiset:  $A <\sim\sim> B = (\text{multiset-of-list } A = \text{multiset-of-list } B)$ 
  apply(simp add: perm-count-conv)
  apply(simp add: multiset-eq-conv-count-eq)
  apply(simp add: count-count)
  done

```

```

lemma set-of-multiset-of-list:  $\text{set-of } (\text{multiset-of-list } A) = \text{set } A$ 
  by (induct A) auto

```

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  $a: \text{finite } A$ 
  shows  $\text{Suc } (\text{Max } A) \notin A$ 
proof (cases  $A = \{\}$ )
  case True
  thus ?thesis by auto
next
  case False
  with  $a$  have  $\text{Max } A \in A \wedge (\forall s \in A. s \leq \text{Max } A)$  by simp
  thus ?thesis by auto
qed

```

— not used

```

lemma not-finite-univ:  $\sim \text{finite } (\text{UNIV}::\text{nat set})$ 
  apply rule
  apply(drule-tac natset-finite-max)
  by force

```

— FIXME should be in main lib

```

lemma LeastI-ex:  $(\exists x. P (x::'a::\text{wellorder})) \implies P (\text{LEAST } x. P x)$ 
  by(blast intro: LeastI)

```

2.2 Summation

```

primrec summation ::  $(\text{nat} \Rightarrow \text{nat}) \Rightarrow \text{nat} \Rightarrow \text{nat}$ 
where

```

$summation\ f\ 0 = f\ 0$
 $|\ summation\ f\ (Suc\ n) = f\ (Suc\ n) + summation\ f\ n$

2.3 Termination Measure

primrec $exp :: [nat, nat] \Rightarrow nat$
where
 $exp\ x\ 0 = 1$
 $| exp\ x\ (Suc\ m) = x * exp\ x\ m$

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

2.4 Functions

definition
 $preImage :: ('a \Rightarrow 'b) \Rightarrow 'b\ set \Rightarrow 'a\ set$ **where**
 $preImage\ f\ A = \{x . f\ x \in A\}$

definition
 $pre :: ('a \Rightarrow 'b) \Rightarrow 'b \Rightarrow 'a\ set$ **where**
 $pre\ f\ a = \{x . f\ x = a\}$

definition
 $equalOn :: ['a\ set, 'a \Rightarrow 'b, 'a \Rightarrow 'b] \Rightarrow bool$ **where**
 $equalOn\ A\ f\ g = (!x:A. f\ x = g\ x)$

lemma $preImage\ insert$: $preImage\ f\ (insert\ a\ A) = pre\ f\ a\ Un\ preImage\ f\ A$
by(*auto simp add: preImage-def pre-def*)

lemma $preImageI$: $f\ x : A \implies x : preImage\ f\ A$
by(*simp add: preImage-def*)

lemma $preImageE$: $x : preImage\ f\ A \implies f\ x : A$
by(*simp add: preImage-def*)

lemma $equalOn\ Un$: $equalOn\ (A \cup B)\ f\ g = (equalOn\ A\ f\ g \wedge equalOn\ B\ f\ g)$
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 $equalOn\ UnD$: $equalOn\ (A\ Un\ B)\ f\ g \implies equalOn\ A\ f\ g \ \&\ equalOn\ B\ f\ g$
by(*auto simp: equalOn-def*)

— FIXME move following elsewhere?

```

lemma inj-inv-singleton[simp]:  $\llbracket \text{inj } f; f z = y \rrbracket \implies \{x. f x = y\} = \{z\}$ 
apply rule
apply(auto simp: inj-on-def) done

lemma finite-pre[simp]:  $\text{inj } f \implies \text{finite } (\text{pre } f x)$ 
apply(simp add: pre-def)
apply (cases  $\exists y. f y = x$ , auto) done

lemma finite-preImage[simp]:  $\llbracket \text{finite } A; \text{inj } f \rrbracket \implies \text{finite } (\text{preImage } f A)$ 
apply(induct A rule: finite-induct)
apply(simp add: preImage-def)
apply(simp add: preImage-insert) done

```

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 deX (X n) = n

```

```

lemma X-deX[simp]:  $X (\text{deX } a) = a$ 
by(cases a) simp

```

```

definition zeroX = X 0

```

```

primrec
nextX :: vbl => vbl where
nextX (X n) = X (Suc n)

```

```

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

```

translations

case p of XCONST zeroX \Rightarrow a | XCONST nextX y \Rightarrow b == (CONST vblcase a (%y. b) p)

definition

*freshVar :: vbl set => vbl where
freshVar vs = X (LEAST n. n \notin deX ' vs)*

lemma *nextX-nextX[iff]: nextX x = nextX y = (x = y)*
by(cases x, cases y) *auto*

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

lemma *ind': P zeroX ==> (! v . P v --> P (nextX v)) ==> P v'*
apply (case-tac v', simp)
apply(induct-tac nat)
apply(simp add: zeroX-def)
apply (drule-tac x=X n in spec, simp)
done

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

lemma *zeroX-nextX[iff]: zeroX \sim nextX a — FIXME iff?*
apply(case-tac a)
apply(simp add: zeroX-def)
done

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

lemma *nextX: nextX (X n) = X (Suc n)*
apply simp **done**

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 | (? y . x = nextX y)*
apply(case-tac x, rename-tac m)
apply(case-tac m)
apply(simp add: zeroX-def)
apply(rule disjI2)
apply (rule-tac x=X nat in exI, simp)
done

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

```

apply(auto intro: vbl-cases[elim-format]) done

lemma comp-vblcase:  $f \circ \text{vblcase } a \ b = \text{vblcase } (f \ a) \ (f \circ \ b)$ 
apply(rule ext)
apply(rule-tac x = x in vbl-casesE)
apply(simp-all add: vblcase-zeroX vblcase-nextX)
done

lemma equalOn-vblcaseI':  $\text{equalOn } (\text{preImage } \text{nextX } A) \ f \ g \implies \text{equalOn } A \ (\text{vblcase } x \ f) \ (\text{vblcase } x \ g)$ 
apply(simp add: equalOn-def)
apply(rule+)
apply (case-tac xa rule: vbl-casesE, simp, simp)
apply(drule-tac x=y in bspec)
apply(simp add: preImage-def)
by assumption

lemma equalOn-vblcaseI:  $(\text{zeroX} : A \dashrightarrow x=y) \implies \text{equalOn } (\text{preImage } \text{nextX } A) \ f \ g \implies \text{equalOn } A \ (\text{vblcase } x \ f) \ (\text{vblcase } y \ g)$ 
apply (rule equalOnI, rule)
apply (case-tac xa rule: vbl-casesE, simp, simp)
apply(simp add: preImage-def equalOn-def)
done

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

lemma finiteFreshVar:  $\text{finite } A \implies \text{freshVar } A \ \sim : A$ 
apply(simp add: freshVar-def)
apply(simp add: X-deX-connection)
apply(rule-tac LeastI-ex)
apply(rule-tac x=(Suc (Max (deX ' A))) in exI)
apply(rule natset-finite-max)
by force

lemma freshVarI:  $[\text{finite } A; B \leq A] \implies \text{freshVar } A \ \sim : B$ 
apply(auto dest!: finiteFreshVar) done

lemma freshVarI2:  $\text{finite } A \implies !x . x \ \sim : A \dashrightarrow P \ x \implies P \ (\text{freshVar } A)$ 
apply(auto dest!: finiteFreshVar) done

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*: $\llbracket \text{signs} = \text{Neg} \implies P; \text{signs} = \text{Pos} \implies P \rrbracket \implies P$
apply(*cases signs, auto*) **done**

lemma *expand-signs-case*: $Q(\text{signs-case } v\text{pos } v\text{neg } F) = ($
 $(F = \text{Pos} \dashrightarrow Q(v\text{pos})) \ \&$
 $(F = \text{Neg} \dashrightarrow Q(v\text{neg}))$
 $)$
by(*induct F simp-all*)

primrec *sign* :: *signs* \Rightarrow *bool* \Rightarrow *bool*
where
 $\text{sign Pos } x = x$
 $\text{sign Neg } x = (\neg x)$

lemma *sign-arg-cong*: $x = y \implies \text{sign } z \ x = \text{sign } z \ y$ **by** *simp*

primrec *invSign* :: *signs* \Rightarrow *signs*
where
 $\text{invSign Pos} = \text{Neg}$
 $\text{invSign Neg} = \text{Pos}$

3.3 Formulas

datatype *formula* =
 $F\text{Atom } \text{signs } \text{predicate } (v\text{bl list})$
 $F\text{Conj } \text{signs } \text{formula } \text{formula}$
 $F\text{All } \text{signs } \text{formula}$

3.4 formula signs induct, formula signs cases

lemma *formula-signs-induct*: \llbracket
 $! p \text{ vs. } P (F\text{Atom Pos } p \text{ vs});$
 $! p \text{ vs. } P (F\text{Atom Neg } p \text{ vs});$
 $!! A B . \llbracket P A; P B \rrbracket \implies P (F\text{Conj Pos } A B);$
 $!! A B . \llbracket P A; P B \rrbracket \implies P (F\text{Conj Neg } A B);$
 $!! A . \llbracket P A \rrbracket \implies P (F\text{All Pos } A);$
 $!! A . \llbracket P A \rrbracket \implies P (F\text{All Neg } A)$
 \rrbracket
 $\implies P A$
apply(*induct-tac A*)
apply(*rule signs.induct, force, force*)
done

lemma *formula-signs-cases*: $!!P.$
 $\llbracket ! p \text{ vs} . P (F\text{Atom Pos } p \text{ vs});$
 $! p \text{ vs} . P (F\text{Atom Neg } p \text{ vs});$
 $!! f1 f2 . P (F\text{Conj Pos } f1 f2);$
 $!! f1 f2 . P (F\text{Conj Neg } f1 f2);$
 $!! f1 . P (F\text{All Pos } f1);$

```

!! f1 . P (FALL Neg f1) ||
==> P A
apply(induct-tac A)
apply(rule signs.induct, force, force)+
done

— induction using nat induction, not wellfounded induction
lemma strong-formula-induct': !A. (! B. size B < size A --> P B) --> P A
==> ! A. size A = n --> P (A::formula)
by (induct-tac n rule: nat-less-induct, blast)

lemma strong-formula-induct: (! A. (! B. size B < size A --> P B) --> P A)
==> P (A::formula)
by (rule strong-formula-induct'[rule-format], blast+)

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

lemma expand-formula-case:
Q(formula-case fatom fconj fall F) = (
(! z P vs . F = FAtom z P vs --> Q (fatom z P vs)) &
(! z A0 A1 . F = FConj z A0 A1 --> Q (fconj z A0 A1)) &
(! z A . F = FALL z A --> Q (fall z A))
)
apply(cases F) apply simp-all done

primrec FNot :: formula => 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 => signs
where
neg Pos = Neg
| neg Neg = Pos

primrec
dual :: [(signs => signs),(signs => signs),(signs => signs)] => formula =>
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*: $\text{dual } p \ q \ r \ o \ \text{dual } P \ Q \ R = \text{dual } (p \ o \ P) \ (q \ o \ Q) \ (r \ o \ R)$
apply(*rule ext*)
apply (*induct-tac x, auto*)
done

lemma *dualFNot'*: $\text{dual } \text{invSign } \text{invSign } \text{invSign} = \text{FNot}$
apply(*rule ext*)
apply(*induct-tac x*)
apply *auto*
done

lemma *dualFNot*: $\text{dual } \text{invSign } \text{id } \text{id} \ (\text{FNot } A) = \text{FNot} \ (\text{dual } \text{invSign } \text{id } \text{id} \ A)$
by(*induct A*) (*auto simp: id-def*)

lemma *dualId*: $\text{dual } \text{id } \text{id } \text{id} \ A = A$
by(*induct A*) (*auto simp: id-def*)

3.5 Frees

primrec *freeVarsF* :: *formula* => *vbl set*

where

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

definition

freeVarsFL :: *formula list* => *vbl set* **where**
freeVarsFL gamma = $\text{Union} \ (\text{freeVarsF } \text{' } \ (\text{set } \text{gamma}))$

lemma *freeVarsF-FNot[simp]*: $\text{freeVarsF} \ (\text{FNot } A) = \text{freeVarsF } A$
by(*induct A*) *auto*

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

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

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

by(*simp add: freeVarsFL-def*)

— FIXME not simp, since simp stops some later lemmas because the simpset isn't confluent

lemma *finite-freeVarsFL[simp]*: *finite* (*freeVarsFL gamma*)
by(*induct gamma*) (*auto simp: freeVarsFL-cons*)

lemma *freeVarsDual*: $\text{freeVarsF} \ (\text{dual } p \ q \ r \ A) = \text{freeVarsF } A$

by(*induct A*) *auto*

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)*

by(*induct A*) *auto*

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

by(*induct A*) *auto*

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

by(*induct A*) *auto*

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)*

by(*induct A*) (*auto simp: 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

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* ~ = {*}*) } **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 *evalP-arg2-cong*: *x* = *y* ==> *evalP* *M* *p* *x* = *evalP* *M* *p* *y* **by** *simp*

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

definition

valid :: *formula* => *bool* **where**
valid *F* ←→ (∀ *M* *phi*. *evalF* *M* *phi* *F* = *True*)

lemma *evalF-FAll*: *evalF* *M* *phi* (*FAll* *Pos* *A*) = (!*x*: (*objects* *M*). (*evalF* *M* (*vblcase*
x (%*v* . *phi* *v*)) *A*))
by *simp*

lemma *evalF-FEx*: *evalF* *M* *phi* (*FAll* *Neg* *A*) = (? *x*:(*objects* *M*). (*evalF* *M*
(*vblcase* *x* (%*v* . *phi* *v*)) *A*))
by *simp*

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

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

lemma *evalF-equiv*[*rule-format*]: ! *f* *g*. (*equalOn* (*freeVarsF* *A*) *f* *g*) → (*evalF* *M*
f *A* = *evalF* *M* *g* *A*)
apply(*induct* *A*)
apply (*force* *simp*:*equalOn-def* *cong*: *map-cong*, *clarify*) **apply** *simp* **apply**(*drule-tac*
equalOn-UnD) **apply** *force*

```

apply clarify apply simp apply(rule-tac f = sign signs in arg-cong) apply(rule
ball-cong) apply rule
apply(rule-tac f = sign signs in arg-cong) apply(force intro: equalOn-vblcaseI')
done

```

— FIXME tricky to automate cong args convincingly?

— composition of substitutions

```

lemma evalF-subF-eq: !phi theta. evalF M phi (subF theta A) = evalF M (phi o
theta) A

```

```

apply(induct-tac A)
apply(simp del: o-apply)
apply(simp del: o-apply)
apply(intro allI)
apply(simp del: o-apply)
apply(rule-tac f=sign signs in arg-cong)
apply(rule ball-cong) apply rule
apply(rule-tac f=sign signs in arg-cong)
apply(subgoal-tac (vblcase x phi o vblcase zeroX (λv. nextX (theta v))) = (vblcase
x (phi o theta)))
apply(simp del: o-apply)
apply(rule ext)
apply (case-tac xa rule: vbl-casesE, simp, simp)
done

```

```

lemma o-id'[simp]: f o (% x. x) = f
by (fold id-def, simp)

```

```

lemma evalF-instance: evalF M phi (instanceF u A) = evalF M (vblcase (phi u)
phi) A
apply(simp add: instanceF-def evalF-subF-eq vblsimps)
done

```

— FIXME move

```

lemma s[simp]: FConj signs formula1 formula2 ≠ formula1
apply(induct-tac formula1, auto) done

```

```

lemma s'[simp]: FConj signs formula1 formula2 ≠ formula2
apply(induct formula2, auto) done

```

```

lemma instanceF-E: instanceF g formula ≠ FAll signs formula
apply clarify
apply(subgoal-tac Suc (size (instanceF g formula)) = (size (FAll signs formula)))
apply force
apply(simp (no-asm) only: size-instance[rule-format])
apply simp done

```

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  $\longleftrightarrow$  (? f : set fs . evalF M phi f = True)
```

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

```
lemma evalS-cons[simp]: evalS M phi (A # Gamma) = (evalF M phi A | evalS
M phi Gamma)
by(simp add: evalS-def)
```

```
lemma evalS-append: evalS M phi (Gamma @ Delta) = (evalS M phi Gamma |
evalS M phi Delta)
by(force simp add: evalS-def)
```

```
lemma evalS-equiv[rule-format]: (equalOn (freeVarsFL Gamma) f g)  $\dashrightarrow$  (evalS
M f Gamma = evalS M g Gamma)
apply (induct Gamma, simp, rule)
apply(simp add: freeVarsFL-cons)
apply(drule-tac equalOn-UnD)
apply(blast dest: evalF-equiv)
done
```

definition

```
modelAssigns :: [model] => (vbl => object) set where
modelAssigns M = { phi . range phi <= objects M }
```

```
lemma modelAssignsI: range f <= objects M  $\implies$  f : modelAssigns M
by(simp add: modelAssigns-def)
```

```
lemma modelAssignsD: f : modelAssigns M  $\implies$  range f <= objects M
by(simp add: modelAssigns-def)
```

definition

```
validS :: formula list => bool where
validS fs  $\longleftrightarrow$  (! M . ! phi : modelAssigns M . evalS M phi fs = 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 = map f a; B = (map f) ‘ b |] ==> (A,B) = mapRule f (a,b)

by(*simp add: mapRule-def*)
 — FIXME tjr would like symmetric

4.2 Deductions

inductive-set

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

where

inferI: [| (conc,prems) : rules;
 prems : Pow(deductions(rules))
 |] ==> conc : deductions(rules)

monos *Pow-mono*

lemma *mono-deductions*: [| A <= B |] ==> deductions(A) <= deductions(B)

apply(*best intro: deductions.inferI elim: deductions.induct*) **done**

4.3 Basic Rule sets

definition

Axioms = { z. ? p vs. 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 Gamma. z = (FConj Neg A0 A1#Gamma,{A0#A1#Gamma}) }

definition

Alls = { z. ? A x Gamma. z = (FAll Pos A#Gamma,{instanceF x A#Gamma}) & x ~: freeVarsFL (FAll Pos A#Gamma) }

definition

Exs = { z. ? A x Gamma. z = (FAll Neg A#Gamma,{instanceF x A#Gamma}) }

definition

$WeakI = \{ z. ? A \quad Gamma. z = (A \# Gamma, \{ Gamma \}) \}$
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
 $Perms = \{ z. ? Gamma \ Gamma' \quad . z = (Gamma, \{ Gamma' \}) \ \& \ Gamma \ \langle \sim \sim \rangle \ Gamma' \}$
definition
 $DAxioms = \{ z. ? p \ vs. \quad z = ([FAtom \ Neg \ p \ vs, FAtom \ Pos \ p \ vs], \{ \}) \}$

lemma *AxiomI*: $[[\ Axioms \ \leq \ A \]] \implies [FAtom \ Pos \ p \ vs, FAtom \ Neg \ p \ vs] : \ deductions(A)$
apply(rule *deductions.inferI*)
apply(auto simp add: *Axioms-def*) **done**

lemma *DAxiomsI*: $[[\ DAxioms \ \leq \ A \]] \implies [FAtom \ Neg \ p \ vs, FAtom \ Pos \ p \ vs] : \ deductions(A)$
apply(rule *deductions.inferI*)
apply(auto simp add: *DAxioms-def*) **done**

lemma *DisjI*: $[[\ A0 \ \# \ A1 \ \# \ Gamma : \ deductions(A); \ Disjs \ \leq \ A \]] \implies (FConj \ Neg \ A0 \ A1 \ \# \ Gamma) : \ deductions(A)$
apply(rule *deductions.inferI*)
apply(auto simp add: *Disjs-def*) **done**

lemma *ConjI*: $[[\ (A0 \ \# \ Gamma) : \ deductions(A); \ (A1 \ \# \ Delta) : \ deductions(A); \ Conjs \ \leq \ A \]] \implies FConj \ Pos \ A0 \ A1 \ \# \ Gamma \ @ \ Delta : \ deductions(A)$
apply(rule-tac prems = $\{ A0 \ \# \ Gamma, A1 \ \# \ Delta \}$ **in** *deductions.inferI*)
apply(auto simp add: *Conjs-def*) **apply** force **done**

lemma *AllI*: $[[\ instanceF \ w \ A \ \# \ Gamma : \ deductions(R); \ w \ \sim : \ freeVarsFL \ (FAll \ Pos \ A \ \# \ Gamma); \ Alls \ \leq \ R \]] \implies (FAll \ Pos \ A \ \# \ Gamma) : \ deductions(R)$
apply(rule-tac prems = $\{ instanceF \ w \ A \ \# \ Gamma \}$ **in** *deductions.inferI*)
apply(auto simp add: *Alls-def*) **done**

lemma *ExI*: $[[\ instanceF \ w \ A \ \# \ Gamma : \ deductions(R); \ Exs \ \leq \ R \]] \implies (FAll \ Neg \ A \ \# \ Gamma) : \ deductions(R)$
apply(rule-tac prems = $\{ instanceF \ w \ A \ \# \ Gamma \}$ **in** *deductions.inferI*)
apply(auto simp add: *Exs-def*) **done**

lemma *WeakI*: $[[\ Gamma : \ deductions \ R; \ WeakI \ \leq \ R \]] \implies A \ \# \ Gamma : \ deductions(R)$
apply(rule-tac prems = $\{ Gamma \}$ **in** *deductions.inferI*)
apply(auto simp add: *WeakI-def*) **done**

lemma *ContrI*: $[[\ A \ \# \ A \ \# \ Gamma : \ deductions \ R; \ Contrs \ \leq \ R \]] \implies A \ \# \ Gamma$

: *deductions*(*R*)
apply(*rule-tac* *prems*={*A*#*A*#*Gamma*} **in** *deductions.inferI*)
apply(*auto simp add: Contrs-def*) **done**

lemma *PermI*: [| *Gamma'* : *deductions R*; *Gamma* <~> *Gamma'*; *Perms* <=
R |] ==> *Gamma* : *deductions*(*R*)
apply(*rule-tac* *prems*={*Gamma'*} **in** *deductions.inferI*)
apply(*auto simp add: Perms-def*) **done**

4.4 Derived Rules

lemma *WeakI1*: [| *Gamma* : *deductions*(*A*); *Weaks* <= *A* |] ==> (*Delta* @
Gamma) : *deductions*(*A*)
apply (*induct* *Delta, simp*)
apply(*auto intro: WeakI*) **done**

lemma *WeakI2*: [| *Gamma* : *deductions*(*A*); *Perms* <= *A*; *Weaks* <= *A* |] ==>
(*Gamma* @ *Delta*) : *deductions*(*A*)
apply(*blast intro: PermI perm-append-swap WeakI1*) **done**

lemma *SATAxiomI*: [| *Axioms* <= *A*; *Weaks* <= *A*; *Perms* <= *A*; *forms* =
[*FAtom Pos n vs, FAtom Neg n vs*] @ *Gamma* |] ==> *forms* : *deductions*(*A*)
apply(*simp only:*)
apply(*blast intro: WeakI2 AxiomI*)
done

lemma *DisjI1*: [| (*A1*#*Gamma*) : *deductions*(*A*); *Disjs* <= *A*; *Weaks* <= *A* |]
==> *FConj Neg A0 A1*#*Gamma* : *deductions*(*A*)
apply(*blast intro: DisjI WeakI*)
done

lemma *DisjI2*: !!*A*. [| (*A0*#*Gamma*) : *deductions*(*A*); *Disjs* <= *A*; *Weaks* <= *A*;
Perms <= *A* |] ==> *FConj Neg A0 A1*#*Gamma* : *deductions*(*A*)
apply(*rule DisjI*)
apply(*rule PermI[OF - perm.swap]*)
apply(*rule WeakI*)
.

— 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*: *Perms* \subseteq *R* ==> *A* @ (*a* # *list*) @ (*a* # *list*) : *deductions R*
==> (*a* # *a* # *A*) @ *list* @ *list* : *deductions R*
apply (*rule PermI, auto*)
apply(*simp add: perm-count-conv count-append*) **done**

lemma *weaken-append*[*rule-format*]: *Contrs* <= *R* ==> *Perms* <= *R* ==> !*A*.
A @ *Gamma* @ *Gamma* : *deductions*(*R*) --> *A* @ *Gamma* : *deductions*(*R*)

apply (*induct-tac* *Gamma*, *simp*, *rule*) **apply** *rule*
apply (*drule-tac* $x=a\#a\#A$ **in** *spec*)
apply (*erule-tac* *impE*)
apply (*rule perm-tmp4*) **apply** (*assumption*, *assumption*)
apply (*thin-tac* $A @ (a \# list) @ a \# list \in \text{deductions } R$)
apply *simp*
apply (*frule-tac* *ContrI*) **apply** *assumption*
apply (*thin-tac* $a \# a \# A @ list \in \text{deductions } R$)
apply (*rule PermI*) **apply** *assumption*
apply (*simp add: perm-count-conv count-append*)
by *assumption*
— FIXME horrible

lemma *ListWeakI*: $\text{Perms} \leq R \implies \text{Contrs} \leq R \implies x \# \text{Gamma} @ \text{Gamma} : \text{deductions}(R) \implies x \# \text{Gamma} : \text{deductions}(R)$
by (*rule weaken-append*[*of* $R [x] \text{Gamma}$, *simplified*])

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

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

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

lemma *rulesInPCs*: $\text{Axioms} \leq PC$ $\text{Axioms} \leq \text{CutFreePC}$

$\text{Conjs} \leq PC$ $\text{Conjs} \leq \text{CutFreePC}$
 $\text{Disjs} \leq PC$ $\text{Disjs} \leq \text{CutFreePC}$
 $\text{Alls} \leq PC$ $\text{Alls} \leq \text{CutFreePC}$
 $\text{Exs} \leq PC$ $\text{Exs} \leq \text{CutFreePC}$
 $\text{Weaks} \leq PC$ $\text{Weaks} \leq \text{CutFreePC}$
 $\text{Contrs} \leq PC$ $\text{Contrs} \leq \text{CutFreePC}$
 $\text{Perms} \leq PC$ $\text{Perms} \leq \text{CutFreePC}$
 $\text{Cuts} \leq PC$
 $\text{CutFreePC} \leq PC$
by (*auto simp: PC-def CutFreePC-def*)

4.6 Monotonicity for CutFreePC deductions

- these lemmas can be used to replace complicated permutation reasoning above
- essentially if x is a deduction, and $x \subseteq y$, then y is a deduction

definition

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

lemma perm: ! xs ys. xs <~~> ys --> (inDed xs = inDed ys)
apply(subgoal-tac ! xs ys. xs <~~> ys --> inDed xs --> inDed ys)
apply (blast intro: perm-sym, clarify)
apply(simp add: inDed-def)
apply (rule PermI, assumption)
apply(rule perm-sym) **apply** assumption
by(blast intro!: rulesInPCs)

lemma contr: ! x xs. inDed (x#x#xs) --> inDed (x#xs)
apply(simp add: inDed-def)
apply(blast intro!: ContrI rulesInPCs)
done

lemma weak: ! x xs. inDed xs --> inDed (x#xs)
apply(simp add: inDed-def)
apply(blast intro!: WeakI rulesInPCs)
done

lemma inDed-mono'[simplified inDed-def]: set x <= set y ==> inDed x ==>
inDed y
using perm-weak-contr-mono[OF perm contr weak] .

lemma inDed-mono[simplified inDed-def]: inDed x ==> set x <= set y ==>
inDed y
using perm-weak-contr-mono[OF perm contr weak] .

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*: $(0, y) : \text{tree subs } \gamma = (y = \gamma)$
apply (*rule iffI*)
apply (*erule tree.cases, auto*)
done

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

$\forall Y. (Suc\ n, Y) \in \text{tree subs } \gamma = (\exists \sigma \in \text{subs } \gamma . (n, Y) \in \text{tree subs } \sigma)$
by (*induct n*) (*blast, force*)
— moving down a tree

definition

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

lemma *injIncLevel*: *inj incLevel*
apply (*simp add: incLevel-def*)
apply (*rule inj-onI*)
apply *auto*
done

lemma *treeEquation*: $\text{tree subs } \gamma = \text{insert } (0, \gamma) (UN\ \delta : \text{subs } \gamma . \text{incLevel } \delta \text{ tree subs } \delta)$
apply (*rule set-ext*)
apply (*simp add: split-paired-all*)
apply (*case-tac a*)
apply (*force simp add: tree0Eq incLevel-def*)
apply (*force simp add: tree1Eq incLevel-def*)
done

definition

fans :: $['a \Rightarrow 'a\ \text{set}] \Rightarrow \text{bool}$ **where**
fans *subs* $\longleftrightarrow (!x. \text{finite } (\text{subs } x))$

lemma *fansD*: *fans* *subs* $\implies \text{finite } (\text{subs } A)$
by (*simp add: fans-def*)

lemma *fansI*: $(!A. \text{finite } (\text{subs } A)) \implies \text{fans } \text{subs}$
by (*simp add: fans-def*)

4.8 Terminal

definition

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

lemma *terminalD*: *terminal* *subs* *Gamma* $\implies x \sim : \text{subs } \text{Gamma}$

by(*simp add: terminal-def*)
 — not a good dest rule

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

4.9 Inherited

definition

inherited :: [$'a \implies 'a \text{ set}, (\text{nat} * 'a) \text{ set} \implies \text{bool}$] $\implies \text{bool}$ **where**
inherited subs $P \iff (!A B. (P A \& P B) = P (A \text{ Un } B))$
 $\& (!A. P A = P (\text{incLevel } 'A))$
 $\& (!n \Gamma A. \sim(\text{terminal subs } \Gamma) \dashrightarrow P A = P (\text{insert } (n, \Gamma) A))$
 $\& (P \{\})$

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

lemma *inheritedUn*[*rule-format*]: *inherited subs* $P \dashrightarrow P A \dashrightarrow P B \dashrightarrow P (A \text{ Un } B)$
by (*auto simp add: inherited-def*)

lemma *inheritedIncLevel*[*rule-format*]: *inherited subs* $P \dashrightarrow P A \dashrightarrow P (\text{incLevel } 'A)$
by (*auto simp add: inherited-def*)

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

lemma *inheritedInsert*[*rule-format*]:
inherited subs $P \dashrightarrow \sim(\text{terminal subs } \Gamma) \dashrightarrow P A \dashrightarrow P (\text{insert } (n, \Gamma) A)$
by (*auto simp add: inherited-def*)

lemma *inheritedI*[*rule-format*]: $[[\forall A B. (P A \& P B) = P (A \text{ Un } B);$
 $\forall A. P A = P (\text{incLevel } 'A);$
 $\forall n \Gamma A. \sim(\text{terminal subs } \Gamma) \dashrightarrow P A = P (\text{insert } (n, \Gamma) A);$
 $P \{\}] \implies \text{inherited subs } P$
by (*simp add: inherited-def*)

lemma *inheritedUnEq*[*rule-format, symmetric*]: *inherited subs* $P \dashrightarrow (P A \& P B) = P (A \text{ Un } B)$
by (*auto simp add: inherited-def*)

lemma *inheritedIncLevelEq*[*rule-format, symmetric*]: *inherited subs* $P \dashrightarrow P A = P$ (*incLevel* ' A)
by (*auto simp add: inherited-def*)

lemma *inheritedInsertEq*[*rule-format, symmetric*]: *inherited subs* $P \dashrightarrow \sim$ (*terminal subs* Γ) $\dashrightarrow P A = P$ (*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*[*rule-format*]:
finite $A \dashrightarrow$ *inherited subs* $P \dashrightarrow (!x:A. P (B x)) = P$ (*UN* $a:A. B a$)
apply(*intro impI*)
apply(*erule finite-induct*)
apply *simp*
apply(*simp add: inheritedEmpty*)
apply(*force dest: inheritedUnEq*)
done

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 (*terminal subs* δ)
shows P (*tree subs* δ) = (! σ :*subs* δ . P (*tree subs* σ))
apply(*insert fansD*[*OF b*])
apply(*subst treeEquation* [*of* - δ])
using *assms*
apply(*simp add: inheritedInsertEq inheritedUNEq*[*symmetric*] *inheritedIncLevelEq*)
done

lemma *inheritedPropagate*:
[[$\sim P$ (*tree subs* δ); *inherited subs* P ; *fans subs*; \sim (*terminal subs* δ)]]
 $\implies \exists \sigma \in$ *subs* δ . $\sim P$ (*tree subs* σ)
by(*simp add: inheritedPropagateEq*)

lemma *inheritedViaSub*: [[*inherited subs* P ; *fans subs*; P (*tree subs* δ); $\sigma \in$ *subs* δ]]
 $\implies P$ (*tree subs* σ)
apply(*frule-tac terminalI*)
apply(*simp add: inheritedPropagateEq*)
done

lemma *inheritedJoin*[*rule-format*]:
(inherited subs P & inherited subs Q) --> inherited subs (%x. P x & Q x)
by(*blast intro!*: *inheritedI*
dest: inheritedUnEq inheritedIncLevelEq inheritedInsertEq inheritedEmpty)

lemma *inheritedJoinI*[*rule-format*]: *[| inherited subs P; inherited subs Q; R = (% x . P x & Q x) |] ==> inherited subs R*
by(*blast intro!*:*inheritedI* *dest: inheritedUnEq inheritedIncLevelEq inheritedInsertEq inheritedEmpty*)

4.10 bounded, boundedBy

definition

boundedBy :: *nat => (nat * 'a) set => bool* **where**
boundedBy N A \longleftrightarrow $(\forall (n, delta) \in A. n < N)$

definition

bounded :: *(nat * 'a) set => bool* **where**
bounded A \longleftrightarrow $(\exists N . boundedBy N A)$

lemma *boundedByEmpty*[*simp*]: *boundedBy N {}*
by(*simp add: boundedBy-def*)

lemma *boundedByInsert*: *boundedBy N (insert (n, delta) B)* = *(n < N & 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* \implies *boundedBy (N+M) B*
by(*auto simp add: boundedBy-def*)

lemma *boundedByAdd2*: *boundedBy M B* \implies *boundedBy (N+M) B*
by(*auto simp add: boundedBy-def*)

lemma *boundedByMono*: *boundedBy m B* \implies *m < M* \implies *boundedBy M B*
by(*auto simp: boundedBy-def*)

lemmas *boundedByMonoD* = *boundedByMono*

lemma *boundedBy0*: *boundedBy 0 A* = *(A = {})*
apply(*simp add: boundedBy-def*)

apply(*auto simp add: boundedBy-def*)
done

lemma *boundedBySuc'*: $\text{boundedBy } N \ A \implies \text{boundedBy } (\text{Suc } N) \ A$
by (*auto simp add: boundedBy-def*)

lemma *boundedByIncLevel*: $\text{boundedBy } n \ (\text{incLevel } ' \ (\text{tree subs } \text{gamma})) = (\exists m . n = \text{Suc } m \ \& \ \text{boundedBy } m \ (\text{tree subs } \text{gamma}))$
apply(*cases n*)
apply(*force simp add: boundedBy0 tree0*)
apply(*force simp add: treeEquation [of - gamma] incLevel-def boundedBy-def*)
done

lemma *boundedByUN*: $\text{boundedBy } N \ (\text{UN } x:A. \ B \ x) = (!x:A. \ \text{boundedBy } N \ (B \ x))$
by(*simp add: boundedBy-def*)

lemma *boundedBySuc[rule-format]*: $\text{sigma} \in \text{subs } \text{Gamma} \implies \text{boundedBy } (\text{Suc } n) \ (\text{tree subs } \text{Gamma}) \longrightarrow \text{boundedBy } n \ (\text{tree subs } \text{sigma})$
apply(*subst treeEquation [of - Gamma]*)
apply *rule*
apply(*simp add: boundedByInsert*)
apply(*simp add: boundedByUN*)
apply(*drule-tac x=sigma in bspec*) **apply** *assumption*
apply(*simp add: boundedByIncLevel*)
done

4.11 Inherited Properties- bounded

lemma *boundedEmpty*: $\text{bounded } \{\}$
by(*simp add: bounded-def*)

lemma *boundedUn*: $\text{bounded } (A \ \text{Un } B) = (\text{bounded } A \ \& \ \text{bounded } B)$
apply(*auto simp add: bounded-def boundedByUn*)
apply(*rule-tac x=N+Na in exI*)
apply(*blast intro: boundedByAdd1 boundedByAdd2*)
done

lemma *boundedIncLevel*: $\text{bounded } (\text{incLevel } ' \ A) = (\text{bounded } A)$
apply (*simp add: bounded-def, rule*)
apply(*erule exE*)
apply(*rule-tac x=N in exI*)
apply (*simp add: boundedBy-def incLevel-def, force*)
apply(*erule exE*)
apply(*rule-tac x=Suc N in exI*)
apply (*simp add: boundedBy-def incLevel-def, force*)
done

lemma *boundedInsert*: $\text{bounded } (\text{insert } a \ B) = (\text{bounded } B)$

```

apply(case-tac a)
apply (simp add: bounded-def boundedByInsert, rule) apply blast
apply(erule exE)
apply(rule-tac x=Suc(aa+N) in exI)
apply(force intro:boundedByMono)
done

```

```

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;  $\forall x. P x --> Q x$  |] ==> founded
subs Q A
  by (auto simp: founded-def)

```

```

lemma foundedSubs: founded subs P (tree subs Gamma) ==> sigma  $\in$  subs Gamma
==> founded subs P (tree subs sigma)
  apply(simp add: founded-def)
  apply(intro ballI impI)
  apply (case-tac x, simp, rule)
  apply(drule-tac x=(Suc a, b) in bspec)
  apply(subst treeEquation)
  apply (force simp: incLevel-def, simp)
done

```

4.13 Inherited Properties- founded

```

lemma foundedInsert[rule-format]:  $\sim$  terminal subs delta --> founded subs P
(insert (n,delta) B) = (founded subs P B)
  apply(simp add: terminal-def founded-def) done

```

```

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

```

```

lemma foundedIncLevel: founded subs P (incLevel ' A) = (founded subs P A)
  apply (simp add: founded-def incLevel-def, auto) done

```

```

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

```

lemma *inheritedFounded*: *inherited subs (founded subs P)*
by(*blast intro!*: *inheritedI foundedUn[symmetric] foundedIncLevel[symmetric]*
foundedInsert[symmetric] foundedEmpty)

4.14 Inherited Properties- finite

lemmas *finiteInsert = finite-insert*

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

lemma *finiteIncLevel*: *finite (incLevel ' A) = finite A*
apply (*insert injIncLevel, rule*)
apply (*frule finite-imageD*)
apply (*blast intro: subset-inj-on, assumption*)
apply (*rule finite-imageI*)
by *assumption*
— FIXME often have *injOn f A, finite f ' A*, to show *A finite*

lemma *finiteEmpty*: *finite {}* **by** *auto*

lemma *inheritedFinite*: *inherited subs (%A. finite A)*
apply(*blast intro!*: *inheritedI finiteUn[symmetric] finiteIncLevel[symmetric] finiteInsert[symmetric] finiteEmpty*)
done

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)))*
apply (*simp add: failingSub-def*)
apply (*drule inheritedPropagate*) **apply** (*assumption+*)
apply (*erule bexE*)
apply (*rule someI2, auto*)
done

lemma *failingSubFailsI*: [*inherited subs P; ~P (tree subs gamma); ~(terminal subs gamma); fans subs*]
=> *~(P (tree subs (failingSub subs P gamma)))*
apply (*rule conjunct2[OF failingSubProps]*) .

lemmas *failingSubFailsE = failingSubFailsI[THEN notE]*

lemma *failingSubSubs*: [| *inherited subs P*; $\sim P$ (*tree subs gamma*); \sim (*terminal subs gamma*); *fans subs* |]
 \implies *failingSub subs P gamma* \in *subs gamma*
apply(*rule conjunct1*[*OF failingSubProps*]) .

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*; $\sim P$ (*tree subs gamma*) |]
 \implies \sim (*P (tree subs (path subs gamma P n))*)
apply (*induct-tac n, simp, simp*)
apply *rule*
apply(*rule failingSubFailsI*) **apply**(*assumption+*)
done

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)*
apply (*induct-tac n, simp-all*) **done**

lemma *pathStarts*: *path subs gamma P 0* = *gamma*
by *simp*

lemma *pathSubs*: [| *inherited subs P*; *fans subs*; $\sim P$ (*tree subs gamma*); \sim (*terminal subs (path subs gamma P n)*) |]
 \implies *path subs gamma P (Suc n)* \in *subs (path subs gamma P n)*
apply *simp*
apply (*rule failingSubSubs, assumption*)
apply(*rule pathFailsP*)
apply(*assumption+*)
done

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* \implies *'a set*, *'a*, *nat* \implies *'a*] \implies *bool* **where**
branch subs Gamma f \longleftrightarrow *f 0* = *Gamma*

$\& (!n . \text{terminal subs } (f n) \dashrightarrow f (Suc n) = f n)$
 $\& (!n . \sim \text{terminal subs } (f n) \dashrightarrow f (Suc n) \in \text{subs } (f n))$

lemma *branch0*: *branch subs Gamma f ==> f 0 = Gamma*
by (*simp add: branch-def*)

lemma *branchStops*: *branch subs Gamma f ==> terminal subs (f n) ==> f (Suc n) = f n*
by (*simp add: branch-def*)

lemma *branchSubs*: *branch subs Gamma f ==> ~ terminal subs (f n) ==> f (Suc n) \in subs (f n)*
by (*simp add: branch-def*)

lemma *branchI*: $[[(f 0 = Gamma);$
 $!n . \text{terminal subs } (f n) \dashrightarrow f (Suc n) = f n;$
 $!n . \sim \text{terminal subs } (f n) \dashrightarrow f (Suc n) \in \text{subs } (f n)]]$ $==>$ *branch subs Gamma f*
by (*simp add: branch-def*)

lemma *branchTerminalPropagates*: *branch subs Gamma f ==> terminal subs (f m) ==> terminal subs (f (m + n))*
apply (*induct-tac n, simp*)
by(*simp add: branchStops*)

lemma *branchTerminalMono*: *branch subs Gamma f ==> m < n ==> terminal subs (f m) ==> terminal subs (f n)*
apply(*subgoal-tac terminal subs (f (m+(n-m)))*) **apply** *force*
apply(*rule branchTerminalPropagates*)
by *auto*

lemma *branchPath*:
 $[[\text{inherited subs } P; \text{fans subs}; \sim P(\text{tree subs gamma})]]$
 $==>$ *branch subs gamma (path subs gamma P)*
by(*auto intro!: branchI pathStarts pathSubs pathStops*)

4.17 failing branch property: abstracts path defn

lemma *failingBranchExistence*: $!!\text{subs}.$
 $[[\text{inherited subs } P; \text{fans subs}; \sim P(\text{tree subs gamma})]]$
 $==>$ $\exists f . \text{branch subs gamma } f \ \& \ (\forall n . \sim P(\text{tree subs } (f n)))$
apply(*rule-tac x=path subs gamma P in exI*)
apply(*rule conjI*)
apply(*force intro!: branchPath*)
apply(*intro allI*)
apply(*rule pathFailsP*)
by *auto*

definition

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

lemma *infBranchI*: [*(f 0 = Gamma); !n . f (Suc n) \in subs (f n)*] ==> *infBranch subs Gamma f*
by (*simp add: infBranch-def*)

4.18 Tree induction principles

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

lemma *boundedTreeInduction'*:

\llbracket *fans subs;*
 $\forall \text{delta. } \sim \text{terminal subs delta} \dashrightarrow (\forall \text{sigma} \in \text{subs delta. } P \text{ sigma}) \dashrightarrow P \text{ delta}$
 $\rrbracket \implies \forall \text{Gamma. boundedBy } m \text{ (tree subs Gamma)} \longrightarrow \text{founded subs } P \text{ (tree subs Gamma)} \longrightarrow P \text{ Gamma}$
apply(*induct-tac m*)
apply(*intro impI allI*)
apply(*simp add: boundedBy0*)
apply(*subgoal-tac (0, Gamma) \in tree subs Gamma*) **apply** *blast* **apply**(*rule tree0*)
apply(*intro impI allI*)
apply(*drule-tac x=Gamma in spec*)
apply (*case-tac terminal subs Gamma, simp*)
apply(*drule-tac foundedD*) **apply** *assumption* **apply** *assumption*
apply (*erule impE, assumption*)
apply (*erule impE, rule*)
apply(*drule-tac x=sigma in spec*)
apply(*erule impE*)
apply(*rule boundedBySuc*) **apply** *assumption* **apply** *assumption*
apply(*erule impE*)
apply(*rule foundedSubs*) **apply** *assumption* **apply** *assumption*
apply *assumption*
apply *assumption*
done

— tjr tidied and introduced new lemmas

lemma *boundedTreeInduction*:

\llbracket *fans subs;*
 $\text{bounded (tree subs Gamma); founded subs } P \text{ (tree subs Gamma);}$
 $\forall \text{delta. } \sim \text{terminal subs delta} \dashrightarrow (\forall \text{sigma} \in \text{subs delta. } P \text{ sigma}) \dashrightarrow P \text{ delta}$
 $\rrbracket \implies P \text{ Gamma}$
apply(*unfold bounded-def*)
apply(*erule exE*)
apply(*frule-tac boundedTreeInduction'*) **apply** *assumption*
apply *force*
done

```

lemma boundedTreeInduction2':
  [| fans subs;
     $\forall \text{delta. } (\forall \text{sigma} \in \text{subs delta. } P \text{ sigma}) \dashrightarrow P \text{ delta}$  |]
  ==>  $\forall \text{Gamma. boundedBy } m \text{ (tree subs Gamma)} \longrightarrow P \text{ Gamma}$ 
  apply (induct-tac m)
  apply (intro impI allI)
  apply (simp (no-asm-use) add: boundedBy0)
  apply (subgoal-tac (0, Gamma)  $\in$  tree subs Gamma) apply blast apply (rule
tree0)
  apply (intro impI allI)
  apply (drule-tac x=Gamma in spec)
  apply (erule impE, rule)
  apply (drule-tac x=sigma in spec)
  apply (erule impE)
  apply (rule boundedBySuc) apply assumption apply assumption
  apply assumption
  apply assumption
  done

```

```

lemma boundedTreeInduction2:
  [| fans subs; boundedBy m (tree subs Gamma);
     $\forall \text{delta. } (\forall \text{sigma} \in \text{subs delta. } P \text{ sigma}) \dashrightarrow P \text{ delta}$  |]
  ==>  $P \text{ Gamma}$ 
  by (frule-tac boundedTreeInduction2', assumption, blast)

```

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 $atoms :: pseq \Rightarrow atom\ list$ **where** $atoms = fst$
definition $nforms :: pseq \Rightarrow nform\ list$ **where** $nforms = snd$

5.2 subs: SATAxiom

definition

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

5.3 subs: a CutFreePC justifiable backwards proof step

definition

$subsFAtom :: [atom\ list, (nat * formula)\ list, signs, predicate, vbl\ list] \Rightarrow pseq\ set$
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

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

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)

apply(*simp add: subs-def subsFAtom-def subsFConj-def subsFAll-def Let-def split-beta split: if-splits list.split formula.split signs.split*)
done

lemma fansSubs: fans subs
apply(*rule fansI*) **apply**(*rule, rule finite-subs*) **done**

lemma subs-def2:

!!*gamma*.

$\sim \text{SATAxiom (sequent gamma)} \implies$

$\text{subs gamma} = (\text{case nforms gamma of}$

$\quad [] \implies \{\}$

$\quad | \text{nA\#nAs} \implies \text{let } (n,A) = \text{nA}$

$\quad \text{in (case A of}$

$\quad \quad \text{FAtom } z \text{ P vs} \implies \text{subsFAtom (atoms gamma)}$

$\text{nAs } z \text{ P vs}$

$\quad | \text{FConj } z \text{ A0 A1} \implies \text{subsFConj (atoms gamma)}$

$\text{nAs } z \text{ A0 A1}$

$\quad | \text{FAll } z \text{ A} \implies \text{subsFAll (atoms gamma) nAs n}$

$\text{z A (freeVarsFL (sequent gamma)))}$)

apply(*simp add: subs-def Let-def nforms-def atoms-def split-beta split: list.split*)

done

5.9 inherited: proofTree

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

apply(*rule ext*)

apply(*simp add: proofTree-def*) **done**

lemma inheritedProofTree: inherited subs proofTree

apply(*simp add: proofTree-def2*)

apply(*auto intro: inheritedJoinI inheritedBounded inheritedFounded*)

done

lemma proofTreeI: [] bounded A; founded subs (SATAxiom o sequent) A [] \implies proofTree A

apply(*simp add: proofTree-def*) **done**

lemma proofTreeBounded: proofTree A \implies bounded A

apply(*simp add: proofTree-def*) **done**

lemma proofTreeTerminal: proofTree A \implies (n,delta) : A \implies terminal subs delta \implies SATAxiom (sequent delta)

apply(*simp add: proofTree-def founded-def*) **apply blast** **done**

5.10 pseq: lemma

lemma snd-o-Pair: (snd o (Pair x)) = (% x. x)

apply(*rule ext*)

by *simp*

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) --> terminal subs gamma*
apply(*simp add: subs-def proofTree-def terminal-def founded-def bounded-def*)
done

lemma *SATAxiomBounded*:*SATAxiom (sequent gamma) ==> bounded (tree subs gamma)*
apply(*frule SATAxiomTerminal*)
apply(*subst treeEquation*)
apply(*simp add: subs-def proofTree-def terminal-def founded-def bounded-def*)
apply(*force simp add: boundedByInsert boundedByEmpty*)
done

lemma *SATAxiomFounded*: *SATAxiom (sequent gamma) ==> founded subs (SATAxiom o sequent) (tree subs gamma)*
apply(*frule SATAxiomTerminal*)
apply(*subst treeEquation*)
apply(*simp add: subs-def proofTree-def terminal-def founded-def bounded-def*)
done

lemma *SATAxiomProofTree*[*rule-format*]: *SATAxiom (sequent gamma) --> proofTree (tree subs gamma)*
apply(*blast intro: proofTreeI SATAxiomBounded SATAxiomFounded*)
done

lemma *SATAxiomEq*: *(proofTree (tree subs gamma) & terminal subs gamma) = SATAxiom (sequent gamma)*
apply(*blast intro: SATAxiomProofTree proofTreeTerminal tree.tree0 SATAxiomTerminal*)
done

— FIXME tjr blast sensitive to obj imp vs meta imp for pTT

5.12 SATAxioms are deductions: - needed

lemma *SAT-deduction*: *SATAxiom x ==> x : deductions CutFreePC*
apply(*simp add: SATAxiom-def*)
apply(*elim exE*)
apply(*rule-tac x=[FAtom Pos n vs,FAtom Neg n vs] in inDed-mono*)
apply (*blast intro!: SATAxiomI rulesInPCs, force*)
done

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 $\neg \text{SATAxiom } (\text{sequent } (ats, (n, f) \# list)) \dashrightarrow \neg \text{terminal subs } (ats, (n, f) \# list)$
 $\dashrightarrow (\forall \text{sigma} \in \text{subs } (ats, (n, f) \# list). \text{sequent } \text{sigma} \in \text{deductions } \text{CutFreePC})$
 $\dashrightarrow \text{sequent } (ats, (n, f) \# list) \in \text{deductions } \text{CutFreePC}$
apply (rule-tac $A=f$ in formula-signs-cases, clarify) **apply**(simp add: *ss*)
apply(rule *PermI*) **apply** assumption **apply**(rule perm-append-Cons) **apply**
(rule rulesInPCs, clarify) **apply**(simp add: *ss*)
apply(rule *PermI*) **apply** assumption **apply**(rule perm-append-Cons) **apply**
(rule rulesInPCs, clarify) **apply**(simp add: *ss*) **apply**(elim conjE)
apply(rule *ConjI'*) **apply** assumption **apply** assumption **apply**(rule rulesIn-
PCs)+
apply clarify **apply**(simp add: *ss*)
apply(rule *DisjI*) **apply** assumption **apply**(rule rulesInPCs)+
apply clarify **apply**(simp add: *ss*)
apply(rule *AllI*) **apply** assumption **apply**(rule finiteFreshVar) **apply**(rule
finite-freeVarsFL) **apply** (rule rulesInPCs, clarify) **apply**(simp add: *ss*)
apply(rule *ContrI*) —!
apply(rule-tac $w = X n$ in *ExI*) **apply**(rule inDed-mono) **apply** assumption
apply force **apply**(rule rulesInPCs)+
done

lemma *subsJustified: !! gamma. ~ terminal subs gamma*

$\implies ! \text{sigma} : \text{subs } \text{gamma} . \text{sequent } \text{sigma} : \text{deductions } (\text{CutFreePC})$
 $\implies \text{sequent } \text{gamma} : \text{deductions } (\text{CutFreePC})$
apply(case-tac *SATAxiom* (sequent gamma))
apply(erule *SAT-deduction*)
apply(case-tac gamma) **apply**(rename-tac *ats nfs*)
apply(case-tac *nfs*)
apply(simp add: terminal-def subs-def sequent-def *Let-def*)
apply(case-tac *a*)
apply(blast intro: *subsJustified'*[rule-format])
done

5.14 proofTrees are deductions: instance of boundedTreeInduction

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

lemma *proofTreeDeductionD*[rule-format]: *proofTree(tree subs gamma) \implies sequent gamma : deductions (CutFreePC)*
apply(rule boundedTreeInduction[*OF fansSubs*])
apply(erule *proofTreeBounded*)
apply(rule *foundedMono*)

```

  apply (force dest: proofTreeD, simp)
  apply (blast intro: SAT-deduction foundedMono subsJustified)
  apply (blast intro: subsJustified)
  done

```

5.15 contains, considers:

```

lemma contains-def2: contains f iA n = (iA : set (nforms (f n)))
  apply (simp add: contains-def nforms-def) done

```

```

lemma considers-def2: considers f iA n = (? nAs . nforms (f n) = iA#nAs)
  apply (simp add: considers-def nforms-def split: list.split) done

```

```

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
  apply (simp add: contains-def2) done
  — FIXME tjr assumptions not required

```

```

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

```

```

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)
  apply (subgoal-tac f (Suc n) = f n)
  apply simp
  apply (blast intro: branchStops nformsTerminal)
  done

```

5.17 path: cases

```

lemma terminalNFormCases: !!f. terminal subs (f n) | (? i A nAs . nforms (f n)
= (i,A)#nAs)
  apply (rule disjCI, simp)
  apply (rule nformsTerminal)
  apply (case-tac nforms (f n))
  apply simp
  apply force
  done

```

```

lemma cases[elim-format]: terminal subs (f n) | (~ (terminal subs (f n) ^ (? i A
nAs . nforms (f n) = (i,A)#nAs))
  apply (auto elim: terminalNFormCases[elim-format])
  done

```

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))
apply(case-tac SATAxiom (sequent (f n)))
apply(blast dest: SATAxiomEq[THEN iffD2])
apply(drule-tac x=n in spec)
apply (simp add: subs-def subs-def subsFAtom-def subsFConj-def subsFAll-def Let-def contains-def terminal-def nforms-def split-beta branch-def split: list.split signs.split expand-formula-case, force)
done

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
apply(frule-tac containsNotTerminal) **apply** force **apply** force
apply(frule-tac branchSubs) **apply** assumption
apply(case-tac considers f iA n) **apply** simp
apply simp
apply(simp add: contains-def) **apply**(case-tac f n) **apply** simp **apply**(drule split-list) **apply**(elim exE conjE) **apply** simp
apply(case-tac ys)
apply (simp add: considers-def, simp)
apply(case-tac SATAxiom (sequent (f n))) **apply**(blast dest: iffD2[OF SATAxiomEq])
apply(simp add: subs-def2 nforms-def Let-def)
apply (case-tac aa, simp)
apply(case-tac ba)
apply(simp add: subsFAtom-def)
apply(case-tac signs) **apply**(simp add: subsFConj-def) **apply** force **apply**(simp add: subsFConj-def)
apply(case-tac signs) **apply**(simp add: subsFAll-def Let-def) **apply**(simp add: subsFAll-def Let-def)
done

5.19 path: no consider lemmas

lemma *noConsidersD*: !!f. ~ considers f iA n ==> nforms (f n) = x#xs ==> iA ~ = x
by(simp add: considers-def2)

lemma *considersD*: !!f. considers f iA n ==> ? xs . nforms (f n) = iA#xs
by(simp add: considers-def2)

5.20 path: contains initially

lemma *contains-initially*:
branch subs (pseq gamma) f ==> A : set gamma ==> (contains f (0,A) 0)
apply(drule branch0)

apply(*simp add: contains-def pseq-def*) **done**

lemma *contains-initialEVs*:

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

apply(*simp add: EV-def*)

apply(*fast dest: contains-initially*) **done**

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

by (*blast intro: r' [of P]*)

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

$! n. P n \longrightarrow (\sim(P (Suc n)) \mid (P (Suc n) \ \& \ msrFn (Suc n) < (msrFn::nat \Rightarrow nat) n)) \implies P m \longrightarrow (? n . P n \ \& \ \sim(P (Suc n)))$

(*is - \implies ?P m*)

apply (*rule r''[of msrFn ?P m], blast*)

done

— FIXME ugly

5.22 costBarrier: lemmas

5.23 costBarrier: exp3 lemmas - bit specific...

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

by (*induct a, simp, simp*)

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

using *exp3Min[of A] exp3Min[of B]*

apply(*case-tac exp 3 A*) **apply**(*simp add: exp3Min*)

apply(*case-tac exp 3 B*) **apply** (*simp add: exp3Min, simp*)

done

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

apply(*subgoal-tac exp 3 (A) < 3 * ((exp 3 A) * (exp 3 B))*)

apply *arith*

apply(*case-tac exp 3 A*) **using** *exp3Min[of A]* **apply** *arith*

apply(*case-tac exp 3 B*) **using** *exp3Min[of B]* **apply** *arith*

apply *simp*

done

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

using *exp3Min[of B]*

apply *arith*

done

lemma *expSum*: $\text{exp } x (a+b) = (\text{exp } x a) * (\text{exp } x b)$
apply(*induct a, auto*) **done**

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 $\sim \text{SATAxiom (sequent (a, (num, fm) \# list))} \rightarrow iA \sim = (num, fm) \rightarrow \neg \text{proofTree (tree subs (a, (num, fm) \# list))} \rightarrow f\text{Sucn} : \text{subs (a, (num, fm) \# list)} \rightarrow iA \in \text{set list} \rightarrow \text{costBarrier } iA (f\text{Sucn}) < \text{costBarrier } iA (a, (num, fm) \# \text{list})$

apply(*rule-tac A=fm in formula-signs-cases*)

— atoms

apply(*simp add: ss*)

apply(*simp add: ss*)

— conj

apply *clarify*

apply(*simp add: ss*)

apply(*erule disjE*)

apply(*simp add: ss exp2*)

apply(*simp add: ss exp2*)

— disj

apply *clarify*

apply(*simp add: ss exp1 exp1'*)

— all

apply *clarify*

apply(*simp add: ss size-instance*)

— ex

apply *clarify*

apply(*simp add: ss size-instance*)

done

lemma *costBarrierDecreases*:

[| *branch subs gamma f*;

!n . \sim proofTree (tree subs (f n));

contains f iA n;

\sim (considers f iA) n

] ==> *costBarrier iA (f (Suc n)) < costBarrier iA (f n)*

apply(*subgoal-tac \neg terminal subs (f n)*)

apply(*subgoal-tac \neg SATAxiom (sequent (f n))*)

apply(*subgoal-tac f (Suc n) \in subs (f n)*)

apply(*frule-tac x=n in spec*)

apply(*case-tac f n, simp*)

apply(*case-tac b, simp*)

apply(*simp add: contains-def*)

apply(*case-tac aa*) **apply**(*rename-tac num fm*) **apply** *simp* **apply**(*simp add:*

contains-def considers-def)

apply(*rule costBarrierDecreases'[rule-format]*) **apply** *force+*

```

  apply(rule branchSubs) apply assumption apply assumption
  apply(blast dest: SATAxiomTerminal)
  apply(blast dest: containsNotTerminal)
  done
— FIXME boring splitting etc.

```

5.25 path: EV contains implies EV considers

```

lemma considersContains: considers f iA n ==> contains f iA n
  apply(simp add: considers-def contains-def)
  apply(cases snd (f n), auto) done

```

```

lemma containsConsiders: [| branch subs gamma f; !n . ~ proofTree (tree subs (f
n));
  EV (contains f iA) |]
==> EV (considers f iA)
  apply(simp add: EV-def)
  apply(erule exE)
  apply(case-tac considers f iA n) apply force
  apply(subgoal-tac  $\exists n. (contains f iA n \wedge \neg considers f iA n) \wedge$ 
 $\neg (contains f iA (Suc n) \wedge \neg considers f iA (Suc n))$ )
  apply(erule exE)
  apply(simp, clarify)
  apply(case-tac contains f iA (Suc na))
  apply force apply(force dest!: containsPropagates)
  apply(rule-tac msrFn = %n. costBarrier iA (f n) and P = %n. contains f iA n
& ~ considers f iA n in terminationRule)
  prefer 2 apply force
  apply(case-tac (contains f iA (Suc na)  $\wedge \neg considers f iA (Suc na)$ ))
  apply simp apply(erule costBarrierDecreases) apply simp-all
  done

```

5.26 EV contains: common lemma

```

lemma lemmaA:
  [| branch subs gamma f; ! n. ~ proofTree (tree subs (f n));
  EV (contains f (i,A)) |]
==> ? n nAs. ~ SATAxiom (sequent (f n)) & (nforms (f n) = (i,A) # nAs & f
(Suc n) : subs (f n))
  apply(frule containsConsiders) apply(assumption+)
  apply(unfold EV-def)
  apply(erule exE, frule considersContains)
  apply(unfold considers-def)
  apply(case-tac snd (f n))
  apply force
  apply simp
  apply(rule-tac x=n in exI)
  apply (intro conjI, rule)
  apply(blast dest!: SATAxiomEq[THEN iffD2])
  apply(rule-tac x=list in exI) apply(simp add: nforms-def)

```

```

apply(frule containsNotTerminal) apply force apply(assumption+)
apply(blast dest!: branchSubs)
done

```

5.27 EV contains: FConj,FDisj,FAI

```

lemma EV-disj: (EV  $P \mid Q$ ) = EV ( $\lambda n. P \ n \mid Q \ n$ )
apply (unfold EV-def, force) done

```

```

lemma evContainsConj: [| EV (contains f (i,FConj Pos A0 A1));
  branch subs gamma f; !n . ~ proofTree (tree subs (f n))
  |] ==> EV (contains f (0,A0)) | EV (contains f (0,A1))
apply(drule lemmA) apply(assumption+)
apply(subgoal-tac EV (\lambda n. contains f (0,A0) n | contains f (0,A1) n))
apply(simp add: EV-disj)
apply(unfold EV-def)
apply clarify
apply(rename-tac n n' nAs, rule-tac x=Suc n' in exI)
apply(simp add: subs-def2 Let-def)
apply(simp add: subsFConj-def)
apply (simp add: contains-def nforms-def, auto)
done

```

```

lemma evContainsDisj: [| EV (contains f (i,FConj Neg A0 A1));
  branch subs gamma f; !n . ~ proofTree (tree subs (f n))
  |] ==> EV (contains f (0,A0)) & EV (contains f (0,A1))
apply(drule lemmA) apply(assumption+)
apply(rule conjI)
apply(unfold EV-def)
apply(erule exE) back
apply(rule-tac x=Suc n in exI)
apply(clarify, simp add: subs-def2 Let-def)
apply(simp add: subsFConj-def)
apply(simp add: contains-def nforms-def)
apply(erule exE) back
apply(rule-tac x=Suc n in exI)
apply(clarify, simp add: subs-def2 Let-def)
apply(simp add: subsFConj-def)
apply(simp add: contains-def nforms-def)
done

```

```

lemma evContainsAll:
  [| EV (contains f (i,FAI Pos body)); branch subs gamma f; !n . ~ proofTree (tree
subs (f n))
  |]
  ==> ? v . EV (contains f (0,instanceF v body))
apply(drule lemmA) apply(assumption+)
apply(erule exE)
apply(rule-tac x=freshVar(freeVarsFL (sequent (f n))) in exI)

```

```

apply(unfold EV-def)
apply(rule-tac x=Suc n in exI)
apply(clarify, simp add: subs-def2 Let-def)
apply(simp add: subsFAll-def Let-def)
apply(simp add: contains-def nforms-def)
done

```

lemma *evContainsEx-instance:*

```

[[ EV (contains f (i,FAll Neg body)); branch subs gamma f; !n . ~ proofTree (tree
subs (f n))
]]
==> EV (contains f (0,instanceF (X i) body))
apply(drule lemmA) apply(assumption+)
apply(erule exE)
apply(unfold EV-def)
apply(rule-tac x=Suc n in exI)
apply(clarify, simp add: subs-def2 Let-def)
apply(simp add: subsFAll-def Let-def)
apply(simp add: contains-def nforms-def)
done

```

lemma *evContainsEx-repeat:*

```

[[ branch subs gamma f; !n . ~ proofTree (tree subs (f n));
EV (contains f (i,FAll Neg body)) ]]
==> EV (contains f (Suc i,FAll Neg body))
apply(drule lemmA) apply(assumption+)
apply(erule exE)
apply(unfold EV-def)
apply(rule-tac x=Suc n in exI)
apply(clarify, simp add: subs-def2 Let-def)
apply(simp add: subsFAll-def Let-def)
apply(simp add: contains-def nforms-def)
done

```

5.28 EV contains: lemmas (temporal related)

lemma *lemma1:* [[*P A n ; !A n. P A n --> P A (Suc n)*]]
==> *P A (n + m)*
apply (*induct m, simp, simp*)
done

lemma *lemma2:*

```

[[ P A n ; P B m ; ! A n. P A n --> P A (Suc n) ]]
==> ? n . P A n & P B n
apply (rule exI[of - n+m], rule)
apply(blast intro!: lemma1)
apply(rule subst[OF add-commute])
apply(blast intro!: lemma1)
done

```

5.29 EV contains: FAtoms

lemma *notTerminalNotSATAxiom*: \neg terminal subs gamma \implies \neg SATAxiom
(sequent gamma)

apply(erule contrapos-nn) **apply**(erule SATAxiomTerminal) **done**

lemma *notTerminalNforms*: \neg terminal subs (f n) \implies nforms (f n) \neq []

apply(erule contrapos-nn) **apply**(erule nformsTerminal) **done**

lemma *atomsPropagate*: [| branch subs gamma f |]
 \implies $x : \text{set} (\text{atoms} (f n)) \dashrightarrow x : \text{set} (\text{atoms} (f (\text{Suc } n)))$
apply(cases terminal subs (f n))
apply(drule branchStops) **apply** assumption **apply** simp
apply(drule branchSubs) **apply** assumption
apply rule **apply**(frule notTerminalNotSATAxiom)
apply(frule notTerminalNforms)
apply(simp add: subs-def2)
apply(cases nforms (f n)) **apply** simp
apply(simp add: Let-def)
apply(case-tac a, auto)
apply(case-tac ba, auto)
apply(simp add: subsFAtom-def atoms-def)
apply(simp add: subsFConj-def atoms-def)
apply(case-tac signs) **apply** force **apply** force
apply(simp add: subsFAll-def atoms-def)
apply(case-tac signs) **apply**(force simp: Let-def) **apply** force
done

5.30 EV contains: FEx cases

lemma *evContainsEx0-allRepeats*:

[| branch subs gamma f; !n . \sim proofTree (tree subs (f n));

EV (contains f (0, Fall Neg A)) |]

\implies EV (contains f (i, Fall Neg A))

apply (induct i, simp)

apply(blast dest!: evContainsEx-repeat)

done

lemma *evContainsEx0-allInstances*:

[| branch subs gamma f; !n . \sim proofTree (tree subs (f n));

EV (contains f (0, Fall Neg A)) |]

\implies EV (contains f (0, instanceF (X i) A))

apply(blast dest!: evContainsEx0-allRepeats intro!: evContainsEx-instance)

done

5.31 pseq: creates initial pseq

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

apply(drule branch0)

apply (simp add: pseq-def contains-def, blast)

done

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

lemma *claim*: $(A \mid B \mid C) = (\sim C \dashrightarrow \sim B \dashrightarrow A)$ **by** *auto*

lemma *natPredCases*: $(!n. P n) \mid (\sim P 0) \mid (? n . P n \ \& \ \sim P (Suc n))$
apply(*rule claim*[*THEN iffD2*])
apply(*intro impI*) **apply** *simp*
apply *rule* **apply**(*induct-tac n*) **apply** *auto*
done

lemma *containsNotTerminal'*:
 $\llbracket \text{branch subs } \gamma f; !n . \sim \text{proofTree } (\text{tree subs } (f n)); \text{contains } f \text{ iA } n \rrbracket \implies$
 $\sim (\text{terminal subs } (f n))$
apply (*rule containsNotTerminal, auto*)
done

lemma *notTerminalSucNotTerminal*: $\llbracket \neg \text{terminal subs } (f (Suc n)); \text{branch subs } \gamma f \rrbracket \implies \neg \text{terminal subs } (f n)$
apply(*erule contrapos-nn*)
apply(*rule-tac branchTerminalPropagates*[*of - - - 1, simplified*])
apply(*assumption+*) **done**
— FIXME move to Tree?

lemma *evContainsExSuc-containsEx*:
 $\llbracket \text{branch subs } (pseq fs) f; !n . \sim \text{proofTree } (\text{tree subs } (f n));$
 $\text{EV } (\text{contains } f (Suc i, Fall Neg body)) \rrbracket$
 $\implies \text{EV } (\text{contains } f (i, Fall Neg body))$
apply(*cut-tac P=%n. \sim contains f (Suc i, Fall Neg body) n in natPredCases*)
apply *simp* **apply**(*erule disjE*)
apply(*simp add: EV-def*)
apply(*erule disjE*)
apply(*blast dest!: containsPSeq0D*)
apply(*thin-tac EV ?x*)
apply(*erule exE*)
apply(*erule conjE*)
apply(*unfold EV-def*) **apply**(*rule-tac x=n in exI*)
apply(*rule considersContains*)
apply(*frule containsNotTerminal'*) **apply**(*assumption+*)
apply(*frule notTerminalSucNotTerminal*) **apply** *assumption*
apply(*thin-tac \neg terminal ?x ?y*)
apply(*frule branchSubs*) **apply** *assumption*
apply(*frule notTerminalNforms*)
apply(*case-tac SATAxiom (sequent (f n))*)
apply(*drule SATAxiomTerminal*) **apply** *simp*
apply(*subgoal-tac* $(\exists i A nAs. nforms (f n) = (i, A) \# nAs)$)
prefer 2 **apply**(*rule-tac f=f and n=n in cases*) **apply** *simp*

```

apply(case-tac nforms (f n)) apply simp apply(case-tac a, force)
apply(erule exE)+
apply(unfold considers-def) apply(simp add: nforms-def)
— shift A into succedent
apply(rule-tac P=snd (f n) = (ia, A) # nAs in rev-mp) apply assumption
apply(thin-tac snd (f n) = (ia, A) # nAs)
apply(rule-tac A=A in formula-signs-cases)
apply(auto simp add: subs-def2 nforms-def Let-def subsFAtom-def subsFConj-def
subsFAll-def contains-def2 Let-def)
done
— phew, bit precarious, but not too much going on besides unfolding defns. and
computing a bit. Need to set_simps up

```

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)) |]
  ==> EV (contains f (i,FAll Neg A)) -->
      EV (contains f (0,FAll Neg A))
apply (induct i, simp)
apply(blast dest!: evContainsExSuc-containsEx)
done

```

```

lemma evContainsExval:
  [| EV (contains f (i,FAll Neg body)); branch subs (pseq fs) f; !n . ~ proofTree
(tree subs (f n))
  |]
  ==> ! v . EV (contains f (0,instanceF v body))
apply rule apply(induct-tac v)
apply(blast intro!: evContainsEx0-allInstances dest!: evContainsEx-containsEx0)
done

```

5.34 EV contains: atoms

```

lemma atomsInSequentI[rule-format]: (z,P,vs) : set (fst ps) -->
  FAtom z P vs : set (sequent ps)
apply(simp add: sequent-def)
apply(cases ps, force)
done

```

```

lemma evContainsAtom1:
  [| branch subs (pseq fs) f; !n . ~ proofTree (tree subs (f n));
   EV (contains f (i,FAtom z P vs)) |]
  ==> ? n . (z,P,vs) : set (fst (f n))
apply(drule lemmA) apply(assumption+)
apply(erule exE) apply(rule-tac x=Suc n in exI)
apply(simp add: subs-def2) apply clarify apply(simp add: subs-def2 Let-def)
apply(simp add: subsFAtom-def) done

```

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

lemma *evContainsAtom*:
 [| *branch subs (pseq fs) f; !n . ~ proofTree (tree subs (f n));*
EV (contains f (i,FAtom z P vs)) |]
 ==> ? *n . (! m . FAtom z P vs : set (sequent (f (n + m))))*
apply(*frule evContainsAtom1*) **apply**(*assumption+*)
apply(*erule exE*)
apply (*rule-tac x=n in exI, rule*)
apply(*rule atomsInSequentI*)
apply (*induct-tac m, simp, simp*)
apply(*rule atomsPropagate'*) **apply**(*assumption+*) **done**

lemma *notEvContainsBothAtoms*:
 [| *branch subs (pseq fs) f; !n . ~ proofTree (tree subs (f n))* |]
 ==> ~ *EV (contains f (i,FAtom Pos p vs))* |
 ~ *EV (contains f (j,FAtom Neg p vs))*
apply *clarify*
apply(*frule evContainsAtom*) **apply**(*assumption+*) **apply**(*thin-tac EV (contains*
f (j, FAtom Neg p vs)))
apply(*frule evContainsAtom*) **apply**(*assumption+*) **apply**(*thin-tac EV (contains*
f (i, FAtom Pos p vs)))
apply(*erule-tac exE*)
apply(*drule-tac x=na in spec*) **back**
apply(*drule-tac x=n in spec*) **back**
apply(*simp add: add-ac*)
apply(*subgoal-tac SATAxiom (sequent (f (n+na)))*)
apply(*force dest: SATAxiomProofTree*)
apply(*force simp add: SATAxiom-def*) **done**

5.35 counterModel: lemmas

lemma *counterModelInRepn*: (*counterM f, counterEvalP f*) : *model*
apply(*simp add: model-def counterM-def*) **done**

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

lemma *inv-obj-obj*: *inv obj (obj n) = n*
using *inj-obj* **apply** *simp* **done**

lemma *map-X-map-counterAssign*: *map X (map (inv obj) (map counterAssign*
xs)) = xs
apply(*simp*)
apply(*subgoal-tac (X o (inv obj o counterAssign)) = (% x . x)*)
apply *simp*
apply(*rule ext*)
apply(*case-tac x*)

apply(*simp add: inv-obj-obj*)
done

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

lemma *inCounterM*: *counterAssign v : objects (counterModel f)*
apply(*induct v*)
apply(*simp add: objectsCounterModel*)
by *blast*

lemma *counterAssign-eqI*[*rule-format*]: *x : objects (counterModel f) --> z = X*
(*inv obj x*) --> *counterAssign z = x*
apply(*force simp: objectsCounterModel inj-obj*) **done**

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

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

lemma *path-evalF'*:
notes *ss = evalPCounterModel counterEvalP-def map-X-map-counterAssign map-map[symmetric]*
and *ss1 = instanceF-def evalF-subF-eq comp-vblcase id-def[symmetric]*
shows [] *branch subs (pseq fs) f;*
!n . ~ proofTree (tree subs (f n))
[] ==> (*? i . EV (contains f (i,A))*) \longrightarrow \sim (*evalF (counterModel f) counterAssign A*)
apply (*rule-tac strong-formula-induct, rule*)
apply(*rule formula-signs-cases*)
— *atom*
apply(*simp add: ss del: map-map*)
apply(*rule, rule*)
apply(*simp add: ss del: map-map*)
apply(*force dest: notEvContainsBothAtoms*)
— *conj*
apply(*force dest: evContainsConj*)
— *disj*
apply(*force dest: evContainsDisj*)
— *all*
apply(*rule, rule*)
apply(*erule exE*)
apply(*drule-tac evContainsAll*) **apply** *assumption* **apply** *assumption*
apply(*erule exE*)
apply(*drule-tac x=(instanceF v f1) in spec*)

```

apply(erule impE, force simp: size-instance)+
apply simp
apply(simp add: ss1)
apply(rule-tac x=counterAssign v in bezI) apply simp apply(simp add: in-
CounterM)
— ex
apply(rule, rule)
apply(erule exE)
apply(drule-tac evContainsEval) apply assumption apply assumption
apply simp
apply rule
apply(simp add: objectsCounterModel) apply(erule exE)
apply(drule-tac x=X i in spec)
apply(drule-tac x=(instanceF (X i) f1) in spec)
apply(erule impE, force simp: size-instance)+
apply(simp add: ss1)
done

```

lemmas path-evalF'' = mp[OF path-evalF']

5.37 adequacy

lemma counterAssignModelAssign: counterAssign : modelAssigns (counterModel gamma)

```

apply (simp add: modelAssigns-def, rule)
apply (erule rangeE, simp)
apply(rule inCounterM)
done

```

lemma branch-contains-initially: branch subs (pseq fs) f \implies x : set fs \implies contains f (0,x) 0

```

apply(simp add: contains-def branch0 pseq-def)
done

```

lemma path-evalF:

```

[| branch subs (pseq fs) f;
 $\forall n. \neg$  proofTree (tree subs (f n));
x  $\in$  set fs
|]  $\implies$   $\neg$  evalF (counterModel f) counterAssign x
apply (rule path-evalF'', assumption, assumption)
apply(rule-tac x=0 in exI) apply(simp add: EV-def)
apply(rule-tac x=0 in exI) apply(simp add: branch-contains-initially)
done

```

lemma validProofTree: \sim proofTree (tree subs (pseq fs)) \implies \sim (validS fs)

```

apply(simp add: validS-def evalS-def)
apply(subgoal-tac  $\exists$ f. branch subs (pseq fs) f  $\wedge$  ( $\forall n. \neg$  proofTree (tree subs (f n))))
apply(elim exE conjE)

```

```

apply(rule-tac x=counterModel f in exI)
apply(rule-tac x=counterAssign in beXI)
  apply(force dest!: path-evalF)
apply(rule counterAssignModelAssign)
apply(rule failingBranchExistence)
  apply(rule inheritedProofTree)
apply (rule fansSubs, assumption)
done

```

```

lemma adequacy[simplified sequent-pseq]: validS fs ==> (sequent (pseq fs)) : de-
ductions CutFreePC
  apply(rule proofTreeDeductionD)
  apply(rule ccontr)
  apply(force dest!: validProofTree)
done

```

end

6 Soundness

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

```

lemma permutation-validS: fs <~~> gs --> (validS fs = validS gs)
  apply(simp add: validS-def)
  apply(simp add: evalS-def)
  apply(simp add: perm-set-eq)
done

```

```

lemma modelAssigns-vblcase: phi ∈ modelAssigns M ==> x ∈ objects M ==>
vblcase x phi ∈ modelAssigns M
  apply (simp add: modelAssigns-def, rule)
  apply(erule-tac rangeE)
  apply(case-tac xaa rule: vbl-casesE)
  apply(simp add: vblsimps)
  apply(simp add: vblsimps) apply force
done

```

```

lemma tmp: (! x : A. P x | Q) ==> (! x : A. P x) | Q by blast

```

```

lemma soundnessFAll: !!Gamma.
  [| u ~: freeVarsFL (FAll Pos A # Gamma);
  validS (instanceF u A # Gamma) |]
  ==> validS (FAll Pos A # Gamma)
  apply (simp add: validS-def, rule)
  apply (drule-tac x=M in spec, rule)
  apply(simp add: evalF-instance)
  apply (rule tmp, rule)
  apply(drule-tac x=% y. if y = u then x else phi y in bspec)

```

```

apply(simp add: modelAssigns-def) apply force
apply(erule disjE)
apply (rule disjI1, simp)
apply(subgoal-tac evalF M (vblcase x ( $\lambda y$ . if  $y = u$  then  $x$  else  $\phi y$ )) A =
evalF M (vblcase x  $\phi$ ) A)
apply force
apply(rule evalF-equiv)
apply(rule equalOn-vblcaseI)
apply(rule,rule)
apply(simp add: freeVarsFL-nil freeVarsFL-cons)
apply (rule equalOnI, force)
apply(rule disjI2)
apply(subgoal-tac evalS M ( $\lambda y$ . if  $y = u$  then  $x$  else  $\phi y$ ) Gamma = evalS M
 $\phi$  Gamma)
apply force
apply(rule evalS-equiv)
apply(rule equalOnI)
apply(force simp: freeVarsFL-nil freeVarsFL-cons)
done

```

lemma soundnessFEx: validS (instanceF x A # Gamma) ==> validS (FAll Neg A # Gamma)

```

apply(simp add: validS-def)
apply (simp add: evalF-instance, rule, rule)
apply(drule-tac x=M in spec)
apply (drule-tac x= $\phi$  in bspec, assumption)
apply(erule disjE)
apply(rule disjI1)
apply (rule-tac x= $\phi$  x in bexI, assumption)
apply(force dest: modelAssignsD subsetD)
apply (rule disjI2, assumption)
done

```

lemma soundnessFCut: [| validS (C # Gamma); validS (FNot C # Delta) |] ==> validS (Gamma @ Delta)

```

apply (simp add: validS-def, rule, rule)
apply(drule-tac x=M in spec)
apply(drule-tac x=M in spec)
apply(drule-tac x= $\phi$  in bspec) apply assumption
apply(drule-tac x= $\phi$  in bspec) apply assumption
apply (simp add: evalS-append evalS-cons evalF-FNot, blast)
done

```

lemma soundness: fs : deductions(PC) ==> (validS fs)

```

apply(erule-tac deductions.induct)
apply(drule-tac PowD)
apply(subgoal-tac prems  $\subseteq$  {x. validS x}) prefer 2 apply force apply(thin-tac
prems  $\subseteq$  deductions PC  $\cap$  {x. validS x})

```

```

apply(simp add: subset-eq)
apply(simp add: PC-def)
apply(elim disjE)
  apply(force simp add: Perms-def permutation-validS)
  apply(force simp: Axioms-def validS-def evalS-def)
  apply(force simp: Conjs-def validS-def evalS-def)
  apply(force simp: Disjs-def validS-def evalS-def)
  apply(simp add: Alls-def)
  apply(force intro: soundnessFAll)
  apply(simp add: Exs-def)
  apply(force intro: soundnessFEx)
  apply(force simp: Weaks-def validS-def evalS-def)
  apply(force simp: Contrs-def validS-def evalS-def)
apply(force simp: Cuts-def intro: soundnessFCut)
done

```

```

lemma completeness: fs : deductions (PC) = validS fs
apply rule
  apply(rule soundness) apply assumption
apply(subgoal-tac fs : deductions CutFreePC)
  apply(rule subsetD) prefer 2 apply assumption
  apply(rule mono-deductions)
  apply(simp add: PC-def CutFreePC-def) apply blast
apply(rule adequacy)
by assumption

```

end