Completeness for FOL

James Margetson, ported by Tom Ridge

September 13, 2023

Contents

1 Permutation Lemmas

theory *PermutationLemmas*
imports *HOL−Library.Multiset*
begin

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

1.1 perm, count equivalence

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

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

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

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

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

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

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

lemma *remdups-append*: *y* : *set ys* −−> *remdups* (*ws*@*y*#*ys*) = *remdups* (*ws*@*ys*)

apply (*induct ws*, *simp*)
apply (*case-tac y = a*, *simp*, *simp*)
done

lemma *perm-contr'*: **assumes** *perm*[*rule-format*]: ! *xs ys*. *mset xs = mset ys* −−> (*P xs = P ys*)
and *contr'*[*rule-format*]: ! *x xs*. *P*(*x*#*x*#*xs*) = *P* (*x*#*xs*)
shows ! *xs*. *length xs = n* −−> (*P xs = P* (*remdups xs*))
apply(*induct n rule*: *nat-less-induct*)
proof (*safe*)
fix *xs* :: *'a list*
assume *a*[*rule-format*]: ∀ *m*<*length xs*. ∀ *ys*. *length ys = m* ⟶ *P ys = P* (*remdups ys*)
show *P xs = P* (*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 ... = *P* ([*y*,*y*]@*ws*@*ys*@*zs*)
apply(*rule perm*) **apply**(*rule iffD2*[*OF perm-count-conv*]) **apply** *rule* **apply**(*simp*) **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* **apply**(*simp*) **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*. *mset xs = mset 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*. *mset xs* = *mset 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*)
proof (*rule allI*, *rule impI*)
fix *ys*
define *zs* **where** ‹*zs* = *filter* (*Not* ∘ *Q*) *xs*›
assume ‹*P* (*ys* @ *filter Q xs*)›
then have ‹*P* (*filter Q xs* @ *ys*)›
apply (*subst perm*) **defer apply** *assumption* **apply** *simp* **done**
then have ‹*P* (*zs* @ *filter Q xs* @ *ys*)›
apply (*induction zs*)
apply (*simp-all add*: *weak*)
done
with *zs-def* **show** ‹*P* (*ys* @ *xs*)›
apply (*subst perm*) **defer apply** *assumption* **apply** *simp* **done**
qed

lemma *perm-weak-filter*: **assumes** *perm*: ! *xs ys*. *mset xs* = *mset 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*. *mset xs* = *mset 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*

define *y′* **where** *y′* = *filter* (% *z*. *z* : *set x*) *y*
from *xy* **have** *set x* = *set y′* **apply**(*simp add*: *y′-def*) **apply** *blast* **done**
hence *rxry′*: *mset* (*remdups x*) = *mset* (*remdups y′*)
using *set-eq-iff-mset-remdups-eq* **by** *auto*

from *Px perm-contr*[*OF perm contr′*] **have** *Prx*: *P* (*remdups x*) **by** *simp*
with *rxry′* **have** *P* (*remdups y′*) **apply** (*subst perm*) **defer apply** *assumption*
apply *simp* **done**

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

end

2 Base

theory *Base*
imports *PermutationLemmas*
begin

2.1 Integrate with Isabelle libraries?

lemma *natset-finite-max*: **assumes** *a*: *finite A*
shows *Suc* (*Max A*) ∉ *A*
proof (*cases A* = {})
case *True*
thus *?thesis* **by** *auto*
next
case *False*
with *a* **have** *Max A* ∈ *A* ∧ (∀ *s* ∈ *A*. *s* ≤ *Max A*) **by** *simp*
thus *?thesis* **by** *auto*
qed

— not used
lemma *not-finite-univ*: ~ *finite* (*UNIV*::*nat set*)
apply *rule*
apply(*drule-tac natset-finite-max*)
by *force*

— FIXME should be in main lib
lemma *LeastI-ex*: (∃ *x*. *P* (*x*::*′a*::*wellorder*)) ⟹ *P* (*LEAST x*. *P x*)
by(*blast intro*: *LeastI*)

2.2 Summation

primrec *summation* :: $(nat \Rightarrow nat) \Rightarrow nat \Rightarrow nat$
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
  $exp\ x\ 0 \quad = 1$
| $exp\ x\ (Suc\ m) = x * exp\ x\ m$

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

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

definition
  equalOn :: $['a\ set,'a => 'b,'a => 'b] => bool$ **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*: $f\ x : A ==> x : preImage\ f\ A$
  by(*simp add*: *preImage-def*)

lemma *preImageE*: $x : preImage\ f\ A ==> f\ x : A$
  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*: $equalOn\ A\ f\ g \Longrightarrow (\forall\ x \in A\ .\ f\ x = g\ x)$
  by(*simp add*: *equalOn-def*)

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

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

— FIXME move following elsewhere?
lemma *inj-inv-singleton*[*simp*]: ⟦ *inj f*; *f z* = *y* ⟧ ⟹ {*x*. *f x* = *y*} = {*z*}
apply *rule*
apply(*auto simp*: *inj-on-def*) **done**

lemma *finite-pre*[*simp*]: *inj f* ⟹ *finite* (*pre f x*)
apply(*simp add*: *pre-def*)
apply (*cases* ∃ *y*. *f y* = *x*, *auto*) **done**

lemma *finite-preImage*[*simp*]: ⟦ *finite A*; *inj f* ⟧ ⟹ *finite* (*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 is we just identified vbls with nats

primrec *deX* :: *vbl* => *nat* **where** *deX* (*X n*) = *n*

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

definition *zeroX* = *X 0*

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

definition

vblcase :: ['*a*,*vbl* => '*a*,*vbl*] => '*a* **where**
vblcase a f n = (@*z*. (*n*=*zeroX* ⟶ *z*=*a*) ∧ (!*x*. *n*=*nextX x* ⟶ *z*=*f*(*x*)))

declare [[*case-translation vblcase zeroX nextX*]]

definition
freshVar :: *vbl set* => *vbl* **where**
freshVar vs = *X* (*LEAST n*. *n* ∉ *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(*rename-tac nat*)
apply(*induct-tac nat*)
apply(*simp add*: *zeroX-def*)
apply(*rename-tac n*)
apply (*drule-tac x*=*X n* **in** *spec*, *simp*)
done

lemma *ind*: ⟦ *P zeroX*; ⋀ *v*. *P v* ⟹ *P* (*nextX v*) ⟧ ⟹ *P v'*
apply(*rule ind'*) **by** *auto*

lemma *zeroX-nextX*[*iff*]: *zeroX* ~= *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*: $[\![x = zeroX \Longrightarrow P; \bigwedge y.\ x = nextX\ y \Longrightarrow P]\!] \Longrightarrow P$
apply(*auto intro*: *vbl-cases*[*elim-format*]) **done**

lemma *comp-vblcase*: *f o vblcase a b* = *vblcase* (*f a*) (*f o b*)
apply(*rule ext*)
apply(*rule-tac x* = *x* **in** *vbl-casesE*)
apply(*simp-all add*: *vblcase-zeroX vblcase-nextX*)
done

lemma *equalOn-vblcaseI'*: *equalOn* (*preImage nextX A*) *f g* ==> *equalOn A* (*vblcase x f*) (*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*: (*zeroX* : *A* −−> *x*=*y*) ==> *equalOn* (*preImage nextX A*) *f g* ==> *equalOn A* (*vblcase x f*) (*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* : (*deX* ‘ *A*))
by *force*

lemma *finiteFreshVar*: *finite A* ==> *freshVar A* ~: *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*: [| *finite A*; *B* <= *A* |] ==> *freshVar A* ~: *B*
apply(*auto dest*!: *finiteFreshVar*) **done**

lemma *freshVarI2*: *finite A* ==> !*x* . *x* ~: *A* −−> *P x* ==> *P* (*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*: ⟦ *signs* = *Neg* ⟹ *P*; *signs* = *Pos* ⟹ *P* ⟧ ⟹ *P*
apply(*cases signs*, *auto*) **done**

lemma *expand-case-signs*: *Q*(*case-signs vpos vneg F*) = (
(*F* = *Pos* −−> *Q* (*vpos*)) &
(*F* = *Neg* −−> *Q* (*vneg*))
)
by(*induct F*) *simp-all*

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

lemma *sign-arg-cong*: *x* = *y* ==> *sign z x* = *sign z y* **by** *simp*

primrec *invSign* :: *signs* ⇒ *signs*
where
invSign Pos = *Neg*
| *invSign Neg* = *Pos*

3.3 Formulas

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

3.4 formula signs induct, formula signs cases

lemma *formula-signs-induct*: [|
! *p vs*. *P* (*FAtom Pos p vs*);
! *p vs*. *P* (*FAtom Neg p vs*);
!! *A B* . [| *P A*; *P B* |] ==> *P* (*FConj Pos A B*);
!! *A B* . [| *P A*; *P B* |] ==> *P* (*FConj Neg A B*);
!! *A* . [| *P A* |] ==> *P* (*FAll Pos A*);
!! *A* . [| *P A* |] ==> *P* (*FAll Neg A*)
|]
==> *P A*
apply(*induct-tac A*)
apply(*rule signs.induct*, *force*, *force*)+
done

lemma *formula-signs-cases*: !!*P*.

```
  [| !! p vs . P (FAtom Pos p vs);
  !! p vs . P (FAtom Neg p vs);
  !! f1 f2  . P (FConj Pos f1 f2);
  !! f1 f2  . P (FConj Neg f1 f2);
  !! f1     . P (FAll  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-case-formula:
  Q(case-formula 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*: *dual p q r o dual P Q R* = *dual* (*p o P*) (*q o Q*) (*r o R*)
apply(*rule ext*)
apply (*induct-tac x*, *auto*)
done

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

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

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

3.5 Frees

primrec *freeVarsF* :: *formula* => *vbl set*
where
freeVarsFAtom: *freeVarsF* (*FAtom z P vs*) = *set vs*
| *freeVarsFConj*: *freeVarsF* (*FConj z A0 A1*) = (*freeVarsF A0*) *Un* (*freeVarsF A1*)

| *freeVarsFAll*: *freeVarsF* (*FAll z body*) = *preImage nextX* (*freeVarsF body*)

definition
freeVarsFL :: *formula list* => *vbl set* **where**
freeVarsFL gamma = *Union* (*freeVarsF* ' (*set gamma*))

lemma *freeVarsF-FNot*[*simp*]: *freeVarsF* (*FNot A*) = *freeVarsF A*
by(*induct A*) *auto*

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

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

lemma *freeVarsFL-cons*: *freeVarsFL* (*A*#*Gamma*) = *freeVarsF A* ∪ *freeVarsFL 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*: *freeVarsF* (*dual p q r A*) = *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

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

typedef *model* = *model*
unfolding *model-def* **by** *auto*

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

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

lemma *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**(*rename-tac signs A f g*) **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(*rename-tac signs phi0 phi theta*)
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* ∘ *vblcase zeroX* (λ*v*. *nextX* (*theta v*))) = (*vblcase x* (*phi* ∘ *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* $\neq$ *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

type-synonym *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*) −−> (*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* $\Longrightarrow$ *f* : *modelAssigns M*
by(*simp add*: *modelAssigns-def*)

lemma *modelAssignsD*: *f* : *modelAssigns M* ⟹ *range f* <= *objects M*
by(*simp add*: *modelAssigns-def*)

definition
validS :: *formula list* => *bool* **where**
validS fs ⟷ (! *M* . ! *phi* : *modelAssigns M* . *evalS M phi fs* = *True*)

4.1 Rules

type-synonym *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

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

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

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

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
Weaks = { *z*. *? A* *Gamma*. *z* = (*A#Gamma*,{*Gamma*})}
definition
Contrs = { *z*. *? A* *Gamma*. *z* = (*A#Gamma*,{*A#A#Gamma*})}
definition
Cuts = { *z*. *? C Delta* *Gamma*. *z* = (*Gamma* @ *Delta*,{*C#Gamma*,*FNot C#Delta*})}
definition
Perms = { *z*. *? Gamma Gamma'* . *z* = (*Gamma*,{*Gamma'*}) & *mset Gamma* = *mset Gamma'*}
definition
DAxioms = { *z*. *? p vs*. *z* = ([*FAtom Neg p vs*,*FAtom Pos p vs*],{}) }

lemma *AxiomI*: [| *Axioms* <= *A* |] ==> [*FAtom Pos p vs*,*FAtom Neg p vs*] : *deductions*(*A*)
apply(*rule deductions.inferI*)
apply(*auto simp add*: *Axioms-def*) **done**

lemma *DAxiomsI*: [| *DAxioms* <= *A* |] ==> [*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* <= *A* |] ==> (*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* <= *A* |] ==> *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* ~: *freeVarsFL* (*FAll Pos A#Gamma*); *Alls* <= *R* |] ==> (*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* <= *R* |] ==> (*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*; *Weaks* <= *R* |] ==> *A#Gamma* : *deductions*(*R*)
apply(*rule-tac prems*={*Gamma*} **in** *deductions.inferI*)
apply(*auto simp add*: *Weaks-def*) **done**

lemma *ContrI*: [| *A#A#Gamma* : *deductions R*; *Contrs* <= *R* |] ==> *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*; *mset Gamma* = *mset 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 (*auto intro*: *PermI* [*of* ‹*Delta* @ *Gamma*›] *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* ‹*A1 # A0 # Gamma*›])
apply *simp-all*

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* $\Longrightarrow$ *A* @ (*a* # *list*) @ (*a* # *list*) : *deductions R* $\Longrightarrow$ (*a* # *a* # *A*) @ *list* @ *list* : *deductions R*
apply (*rule PermI*, *auto*)
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* ∈ *deductions R*)
apply *simp*
apply(*frule-tac ContrI*) **apply** *assumption*
apply(*thin-tac a* # *a* # *A* @ *list* ∈ *deductions R*)
apply(*rule PermI*) **apply** *assumption*
apply(*simp add*: *perm-count-conv*)
by *assumption*
— FIXME horrible

lemma *ListWeakI*: *Perms* <= *R* ==> *Contrs* <= *R* ==> *x* # *Gamma* @ *Gamma* : *deductions*(*R*) ==> *x* # *Gamma* : *deductions*(*R*)
by(*rule weaken-append*[*of R* [*x*] *Gamma*, *simplified*])

lemma *ConjI'*: [| (*A0*#*Gamma*) : *deductions*(*A*); (*A1*#*Gamma*) : *deductions*(*A*); *Contrs* <= *A*; *Conjs* <= *A*; *Perms* <= *A* |] ==> *FConj Pos A0 A1*#*Gamma* : *deductions*(*A*)
apply(*rule ListWeakI*, *assumption*, *assumption*)
apply(*rule ConjI*) .

4.5 Standard Rule Sets For Predicate Calculus

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

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

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

Disjs <= *PC Disjs* <= *CutFreePC*
Alls <= *PC Alls* <= *CutFreePC*
Exs <= *PC Exs* <= *CutFreePC*
Weaks <= *PC Weaks* <= *CutFreePC*
Contrs <= *PC Contrs* <= *CutFreePC*
Perms <= *PC Perms* <= *CutFreePC*
Cuts <= *PC*
CutFreePC <= *PC*
by(*auto simp*: *PC-def CutFreePC-def*)

4.6 Monotonicity for CutFreePC deductions

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

lemma *perm*: ! *xs ys*. *mset xs* = *mset ys* −−> (*inDed xs* = *inDed ys*)
by (*metis PermI inDed-def rulesInPCs*(*16*))

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*) : *tree subs gamma* = (*y* = *gamma*)
apply(*rule iffI*)
apply (*erule tree.cases*, *auto*)
done

lemma *tree1Eq* [*rule-format*]:
∀ *Y*. (*Suc n*, *Y*) ∈ *tree subs gamma* = (∃ *sigma* ∈ *subs gamma* . (*n*, *Y*) ∈ *tree subs sigma*)
by (*induct n*) (*blast*, *force*)
— moving down a tree

definition
incLevel :: *nat* ∗ *'a* => *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*: *tree subs gamma* = *insert* (*0*,*gamma*) (*UN delta*:*subs gamma* . *incLevel* ‘ *tree subs delta*)
apply(*rule set-eqI*)
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* => *'a set*] => *bool* **where**
fans subs ⟷ (!*x*. *finite* (*subs x*))

lemma *fansD*: *fans subs* ==> *finite* (*subs A*)
by(*simp add*: *fans-def*)

lemma *fansI*: (!*A*. *finite* (*subs A*)) ==> *fans subs*
by(*simp add*: *fans-def*)

4.8 Terminal

definition

terminal :: [*'a* => *'a set*,*'a*] => *bool* **where**
terminal subs delta ⟷ *subs*(*delta*) = {}

lemma *terminalD*: *terminal subs Gamma* ==> *x* ~: *subs Gamma*
by(*simp add*: *terminal-def*)
— not a good dest rule

lemma *terminalI*: *x* ∈ *subs Gamma* ==> ~ *terminal subs Gamma*
by(*auto simp add*: *terminal-def*)
— not a good intro rule

4.9 Inherited

definition
inherited :: [*'a* => *'a set*,(*nat* ∗ *'a*) *set* => *bool*] => *bool* **where**
inherited subs P ⟷ (!*A B*. (*P A* & *P B*) = *P* (*A Un B*))
& (!*A*. *P A* = *P* (*incLevel* ‘ *A*))
& (!*n Gamma A*. ~(*terminal subs Gamma*) −−> *P A* = *P* (*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* −−> *P A* −−> *P B* −−> *P* (*A Un B*)
by (*auto simp add*: *inherited-def*)

lemma *inheritedIncLevel*[*rule-format*]: *inherited subs P* −−> *P A* −−> *P* (*incLevel* ‘ *A*)
by (*auto simp add*: *inherited-def*)

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

lemma *inheritedInsert*[*rule-format*]:
inherited subs P −−> ~(*terminal subs Gamma*) −−> *P A* −−> *P* (*insert* (*n*,*Gamma*) *A*)
by (*auto simp add*: *inherited-def*)

lemma *inheritedI*[*rule-format*]: [| ∀ *A B* . (*P A* & *P B*) = *P* (*A Un B*);
∀ *A* . *P A* = *P* (*incLevel* ‘ *A*);
∀ *n Gamma A* . ~(*terminal subs Gamma*) −−> *P A* = *P* (*insert* (*n*,*Gamma*) *A*);
P {} |] ==> *inherited subs P*
by (*simp add*: *inherited-def*)

lemma *inheritedUnEq*[*rule-format*, *symmetric*]: *inherited subs P* −−> (*P A* & *P*

$B) = P\ (A\ Un\ B)$
by (*auto simp add*: *inherited-def*)

lemma *inheritedIncLevelEq*[*rule-format*, *symmetric*]: *inherited subs P* −−> *P A* = *P* (*incLevel* ‘ *A*)
by (*auto simp add*: *inherited-def*)

lemma *inheritedInsertEq*[*rule-format*, *symmetric*]: *inherited subs P* −−> ~(*terminal subs Gamma*) −−> *P A* = *P* (*insert* (*n*,*Gamma*) *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 −−> *inherited subs P* −−> (!*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*: ~(*terminal subs delta*)
shows *P*(*tree subs delta*) = (!*sigma*:*subs delta*. *P*(*tree subs sigma*))
apply(*insert fansD*[*OF b*])
apply(*subst treeEquation* [*of* - *delta*])
using *assms*
apply(*simp add*: *inheritedInsertEq inheritedUNEq*[*symmetric*] *inheritedIncLevelEq*)
done

lemma *inheritedPropagate*:
[| ~*P*(*tree subs delta*); *inherited subs P*; *fans subs*; ~(*terminal subs delta*)|]
==> ∃ *sigma* ∈ *subs delta* . ~*P*(*tree subs sigma*)
by(*simp add*: *inheritedPropagateEq*)

lemma *inheritedViaSub*: [| *inherited subs P*; *fans subs*; *P*(*tree subs delta*); *sigma* ∈ *subs delta* |]
==> *P*(*tree subs sigma*)
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 ⟷ (∀ (*n*,*delta*) ∈ *A*. *n* < *N*)

definition
bounded :: (*nat* ∗ *'a*) *set* => *bool* **where**
bounded A ⟷ (∃ *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* ⟹ *boundedBy* (*N*+*M*) *B*
by(*auto simp add*: *boundedBy-def*)

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

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

lemmas *boundedByMonoD* = *boundedByMono*

lemma *boundedBy0*: *boundedBy 0 A* = (*A* = {})

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

lemma *boundedBySuc'*: *boundedBy N A* $\Longrightarrow$ *boundedBy* (*Suc N*) *A*
by (*auto simp add*: *boundedBy-def*)

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

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

lemma *boundedBySuc*[*rule-format*]: *sigma* $\in$ *subs Gamma* $\Longrightarrow$ *boundedBy* (*Suc n*) (*tree subs Gamma*) $\longrightarrow$ *boundedBy n* (*tree subs 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*: *bounded* {}
by(*simp add*: *bounded-def*)

lemma *boundedUn*: *bounded* (*A Un B*) = (*bounded A* & *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*: *bounded* (*incLevel*‘ *A*) = (*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*: *bounded* (*insert a B*) = (*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*; ∀ *x*. *P x −−> Q x* |] *==> founded subs Q A*
by (*auto simp*: *founded-def*)

lemma *foundedSubs*: *founded subs P* (*tree subs Gamma*) ⟹ *sigma* ∈ *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*]: ~ *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*; ~*P* (*tree subs gamma*); ~(*terminal subs gamma*); *fans subs* |]
==> *failingSub subs P gamma* ∈ *subs gamma*
apply(*rule conjunct1*[*OF failingSubProps*]) **.**

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

lemma *pathFailsP*: [| *inherited subs P*; *fans subs*; ~*P*(*tree subs gamma*) |]
==> ~(*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* |]
==> *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*; ~*P*(*tree subs gamma*); ~ (*terminal subs* (*path subs gamma P n*)) |]
==> *path subs gamma P* (*Suc n*) ∈ *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*) ==> *path subs gamma P* (*Suc n*) = *path subs gamma P n*
by *simp*

4.16 Branch

definition
branch :: ['*a* => '*a set*,'*a*,*nat* => '*a*] => *bool* **where**
branch subs Gamma f ⟷ *f 0* = *Gamma*

& (!n . *terminal subs* (f n) −−> f (*Suc* n) = f n)
& (!n . ~ *terminal subs* (f n) −−> f (*Suc* n) ∈ *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*) ∈ *subs* (*f n*)
by (*simp add*: *branch-def*)

lemma *branchI*: [| (*f 0* = *Gamma*);
!*n* . *terminal subs* (*f n*) −−> *f* (*Suc n*) = *f n*;
!*n* . ~ *terminal subs* (*f n*) −−> *f* (*Suc n*) ∈ *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*)
.

lemma *branchPath*:
[| *inherited subs P*; *fans subs*; ~*P*(*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*: !!*subs*.
[| *inherited subs P*; *fans subs*; ~*P*(*tree subs gamma*) |]
==> ∃*f* . *branch subs gamma f* & (∀ *n* . ~*P*(*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 ⟷ *f 0* = *Gamma* & (∀ *n*. *f* (*Suc n*) ∈ *subs* (*f n*))

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

4.18 Tree induction principles

lemma *boundedTreeInduction'*:
⟦ *fans subs*;
∀ *delta*. ~ *terminal subs delta* −−> (∀ *sigma* ∈ *subs delta*. *P sigma*) −−> *P delta* ⟧
⟹ ∀ *Gamma*. *boundedBy m* (*tree subs Gamma*) ⟶ *founded subs P* (*tree subs Gamma*) ⟶ *P Gamma*
apply(*induct-tac m*)
apply(*intro impI allI*)
apply(*simp add*: *boundedBy0*)
apply(*subgoal-tac* (*0*,*Gamma*) ∈ *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*:
⟦*fans subs*;
bounded (*tree subs Gamma*); *founded subs P* (*tree subs Gamma*);
∀ *delta*. ~ *terminal subs delta* −−> (∀ *sigma* ∈ *subs delta*. *P sigma*) −−> *P delta* ⟧ ⟹ *P Gamma*
apply(*unfold bounded-def*)
apply(*erule exE*)
apply(*frule-tac boundedTreeInduction'*) **apply** *assumption*
apply *force*
done

lemma *boundedTreeInduction2'*:
[| *fans subs*;

```
    ∀ delta. (∀ sigma ∈ subs delta. P sigma) −−> P delta |]
  ==> ∀ Gamma. boundedBy m (tree subs Gamma) ⟶ P Gamma
  apply(induct-tac m)
   apply(intro impI allI)
   apply(simp (no-asm-use) add: boundedBy0)
    apply(subgoal-tac (0,Gamma) ∈ 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);
          ∀ delta. (∀ sigma ∈ subs delta. P sigma) −−> P delta |]
      ==> P Gamma
  by (frule-tac boundedTreeInduction2', assumption, blast)

end
```

5 Completeness

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

5.1 pseq: type represents a processed sequent

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

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

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

definition atoms :: pseq => atom list where atoms = fst
definition nforms :: pseq => nform list where nforms = snd
```

5.2 subs: SATAxiom

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

5.3 subs: a CutFreePC justifiable backwards proof step

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

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

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

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

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

definition
proofTree :: (*nat* ∗ *pseq*) *set* => *bool* **where**
proofTree A ⟷ *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 ⟷ *nA* : *set* (*snd* (*f n*))

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

costBarrier nA = (%(*atms*,*nAs*).
let barrier = *takeWhile* (%*x*. *nA* ~= *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* . ~(*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*.
~ *SATAxiom* (*sequent gamma*) ==>
subs gamma = (*case nforms gamma of*
[] => {}
| *nA*#*nAs* => *let* (*n*,*A*) = *nA*
in (*case A of*
FAtom z P vs => *subsFAtom* (*atoms gamma*) *nAs z P vs*
| *FConj z A0 A1* => *subsFConj* (*atoms gamma*) *nAs z A0 A1*
| *FAll z A* => *subsFAll* (*atoms gamma*) *nAs n 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* |] ==> *proofTree A*
apply(*simp add*: *proofTree-def*) **done**

lemma *proofTreeBounded*: *proofTree A* ==> *bounded A*
apply(*simp add*: *proofTree-def*) **done**

lemma *proofTreeTerminal*: *proofTree A* ==> (*n*,*delta*) : *A* ==> *terminal subs delta* ==> *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 ¬ *SATAxiom* (*sequent* (*ats*, (*n*, *f*) # *list*)) −−> ¬ *terminal subs* (*ats*, (*n*, *f*) # *list*)
−−> (∀ *sigma*∈*subs* (*ats*, (*n*, *f*) # *list*). *sequent sigma* ∈ *deductions CutFreePC*)

−−> *sequent* (*ats*, (*n*, *f*) # *list*) ∈ *deductions CutFreePC*
apply (*rule-tac A=f* **in** *formula-signs-cases*, *clarify*) **apply**(*simp add*: *ss*)
apply(*rule PermI*) **apply** *assumption* **apply** *simp-all*
apply (*rule rulesInPCs*, *clarify*) **apply**(*simp add*: *ss*)
apply(*rule PermI*) **apply** *assumption* **apply** *simp-all*
apply (*rule rulesInPCs*, *clarify*) **apply**(*simp add*: *ss*) **apply**(*elim conjE*)
apply(*rule ConjI'*) **apply** *assumption* **apply** *assumption* **apply**(*rule rulesInPCs*)+
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*
==> ! *sigma* : *subs gamma* . *sequent sigma* : *deductions* (*CutFreePC*)
==> *sequent gamma* : *deductions* (*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*) ⟹ *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-case-formula*, *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(*rename-tac signs a b*)
apply(*case-tac signs*) **apply**(*simp add*: *subsFConj-def*) **apply** *force* **apply**(*simp add*: *subsFConj-def*)
apply(*rename-tac signs a*)
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* $\Longrightarrow$ *A* : *set gamma* $\Longrightarrow$ *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*] **for** *msrFn*
lemmas *r'* = *r*[*simplified measure-def inv-image-def less-than-def less-eq mem-Collect-eq*]

lemma *r''*: (∀ *x*. (∀ *y*. (((*msrFn*::'*a* ⇒ *nat*) *y*) < ((*msrFn* :: '*a* ⇒ *nat*) *x*)) ⟶ *P y*) ⟶ *P x*) ⟹ *P a*
by (*blast intro*: *r'* [*of* - *P*])

lemma *terminationRule* [*rule-format*]:
! *n*. *P n* −−> (~(*P* (*Suc n*)) | (*P* (*Suc n*) & *msrFn* (*Suc n*) < (*msrFn*::*nat* => *nat*) *n*)) ==> *P m* −−> (*? n* . *P n* & ~(*P* (*Suc n*)))
(**is** - ⟹ *?P m*)
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*: *exp x* (*a*+*b*) = (*exp x a*) ∗ (*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 ~*SATAxiom* (*sequent*
(*a*, (*num*,*fm*) # *list*)) −−> *iA* ~= (*num*, *fm*) −−> ¬ *proofTree* (*tree subs* (*a*, (*num*, *fm*) # *list*)) −−> *fSucn* : *subs* (*a*, (*num*, *fm*) # *list*) −−> *iA* ∈ *set list* −−> *costBarrier iA* (*fSucn*) < *costBarrier iA* (*a*, (*num*, *fm*) # *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* . ~*proofTree* (*tree subs* (*f n*));
contains f iA n;
~(*considers f iA*) *n*
|] ==> *costBarrier iA* (*f* (*Suc n*)) < *costBarrier iA* (*f n*)
apply(*subgoal-tac* ¬ *terminal subs* (*f n*))
apply(*subgoal-tac* ¬ *SATAxiom* (*sequent* (*f n*)))
apply(*subgoal-tac f* (*Suc n*) ∈ *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* $\Longrightarrow$ *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 *lemmA*:
[| *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,FAll

lemma *EV-disj*: (*EV P* | *EV Q*) = *EV* (λ*n*. *P n* | *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* (λ *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*,*FAll 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*: ¬ *terminal subs gamma* ⟹ ¬ *SATAxiom* (*sequent gamma*)
apply(*erule contrapos-nn*) **apply**(*erule SATAxiomTerminal*) **done**

lemma *notTerminalNforms*: ¬ *terminal subs* (*f n*) ⟹ *nforms* (*f n*) ≠ []
apply(*erule contrapos-nn*) **apply**(*erule nformsTerminal*) **done**

lemma *atomsPropagate*: [| *branch subs gamma f* |]
==> *x* : *set* (*atoms* (*f n*)) −−> *x* : *set* (*atoms* (*f* (*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(*rename-tac signs a b*)
apply(*case-tac signs*) **apply** *force* **apply** *force*
apply(*simp add*: *subsFAll-def atoms-def*)
apply(*rename-tac signs a*)
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* . ~ *proofTree* (*tree subs* (*f n*));
EV (*contains f* (*0*,*FAll Neg A*)) |]
==> *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* . ~ *proofTree* (*tree subs* (*f n*));
EV (*contains f* (*0*,*FAll Neg A*)) |]
==> *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* ⟹ *contains f* (*i*,*A*) *0* ⟹ *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'*:
$[\![\ branch\ subs\ gamma\ f;\ !n\ .\ \sim proofTree\ (tree\ subs\ (f\ n));\ contains\ f\ iA\ n\ ]\!] \Longrightarrow \sim (terminal\ subs\ (f\ n))$
apply (*rule containsNotTerminal*, *auto*)
done

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

lemma *evContainsExSuc-containsEx*:
[| *branch subs* (*pseq fs*) *f*; !*n* . ~ *proofTree* (*tree subs* (*f n*));
EV (*contains f* (*Suc i*,*FAll Neg body*)) |]
==> *EV* (*contains f* (*i*,*FAll Neg body*))
apply(*cut-tac P*=%*n*. ~ *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* -)
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* ¬ *terminal x y* **for** *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* (∃ *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*: *ac-simps*)
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* ∘ (*inv obj* ∘ *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*))) ⟶ ~(*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** *bexI*) **apply** *simp* **apply**(*simp add*: *inCounterM*)
— ex
apply(*rule*, *rule*)
apply(*erule exE*)
apply(*drule-tac evContainsExval*) **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* ⟹ *x* : *set fs* ⟹ *contains f* (*0*,*x*) *0*
apply(*simp add*: *contains-def branch0 pseq-def*)
done

lemma *path-evalF*:
[| *branch subs* (*pseq fs*) *f*;
∀ *n*. ¬ *proofTree* (*tree subs* (*f n*));
x ∈ *set fs*
|] ==> ¬ *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*: ~*proofTree* (*tree subs* (*pseq fs*)) ==> ~(*validS fs*)
apply(*simp add*: *validS-def evalS-def*)
apply(*subgoal-tac* ∃ *f*. *branch subs* (*pseq fs*) *f* ∧ (∀ *n*. ¬ *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)) : deductions CutFreePC*
apply(*rule proofTreeDeductionD*)
apply(*rule ccontr*)
apply(*force dest*!: *validProofTree*)
done

end

6 Soundness

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

lemma *permutation-validS*: *mset fs = mset gs −−> (validS fs = validS gs)*
apply(*simp add*: *validS-def*)
apply(*simp add*: *evalS-def*)
using *mset-eq-setD* **apply** *blast*
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*, *auto*)
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* (λ*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-cons*)
apply (*rule equalOnI*, *force*)
apply(*rule disjI2*)
apply(*subgoal-tac evalS M* (λ*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-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 evalF-FNot*, *blast*)
done

lemma *soundness*: *fs* : *deductions*(*PC*) ==> (*validS fs*)
apply(*erule-tac deductions.induct*)
apply(*drule-tac PowD*)
apply(*subgoal-tac prems* ⊆ {*x*. *validS x*}) **prefer** *2* **apply** *force* **apply**(*thin-tac prems* ⊆ *deductions PC* ∩ {*x*. *validS x*})
apply(*simp add*: *subset-eq*)
apply(*simp add*: *PC-def*)
apply(*elim disjE*)
apply (*auto simp add*: *Perms-def*)
apply (*subst permutation-validS*)
defer

```
        apply assumption
       apply simp-all
      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
```