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*›
⟨*proof*⟩

lemma *perm-count*: *mset A* = *mset B* ⟹ (∀ *x*. *count-list A x* = *count-list B x*)
⟨*proof*⟩

lemma *count-0*: (∀ *x*. *count-list B x* = *0*) = (*B* = [])
⟨*proof*⟩

lemma *count-Suc*: *count-list B a* = *Suc m* ⟹ *a* : *set B*
⟨*proof*⟩

lemma *count-perm*: !! *B*. (∀ *x*. *count-list A x* = *count-list B x*) ⟹ *mset A* = *mset B*
⟨*proof*⟩

lemma *perm-count-conv*: *mset A* = *mset B* ⟷ (∀ *x*. *count-list A x* = *count-list B x*)
⟨*proof*⟩

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*)

⟨*proof*⟩

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*))
⟨*proof*⟩

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*))
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *length-rem*: *length* (*rem x xs*) <= *length xs*
⟨*proof*⟩

lemma *rem-notin*: *x* ~: *set xs* ==> *rem x xs* = *xs*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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*⟩

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*⟩
lemma *not-finite-univ*: ~ *finite* (*UNIV*::*nat set*)
⟨*proof*⟩
lemma *LeastI-ex*: (∃ *x*. *P* (*x*::'*a*::*wellorder*)) ⟹ *P* (*LEAST x*. *P x*)
⟨*proof*⟩

2.2 Summation

primrec *summation* :: (*nat* ⇒ *nat*) ⇒ *nat* ⇒ *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*] ⇒ *nat*
where
exp x 0 = *1*
| *exp x* (*Suc m*) = *x* ∗ *exp x m*

primrec *sumList* :: *nat list* ⇒ *nat*
where
sumList [] = *0*
| *sumList* (*x*#*xs*) = *x* + *sumList xs*

2.4 Functions

definition
preImage :: ('*a* ⇒ '*b*) ⇒ '*b set* ⇒ '*a set* **where**
preImage f A = { *x* . *f x* ∈ *A*}

definition
pre :: ('*a* ⇒ '*b*) => '*b* ⇒ '*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*
⟨*proof*⟩

lemma *preImageI*: *f x* : *A* ==> *x* : *preImage f A*
⟨*proof*⟩

lemma *preImageE*: *x* : *preImage f A* ==> *f x* : *A*
⟨*proof*⟩

lemma *equalOn-Un*: *equalOn* (*A* ∪ *B*) *f g* = (*equalOn A f g* ∧ *equalOn B f g*)
⟨*proof*⟩

lemma *equalOnD*: *equalOn A f g* ⟹ (∀ *x* ∈ *A* . *f x* = *g x*)
⟨*proof*⟩

lemma *equalOnI*:(∀ *x* ∈ *A* . *f x* = *g x*) ⟹ *equalOn A f g*
⟨*proof*⟩

lemma *equalOn-UnD*: *equalOn* (*A Un B*) *f g* ==> *equalOn A f g* & *equalOn B f g*
⟨*proof*⟩
lemma *inj-inv-singleton*[*simp*]: ⟦ *inj f*; *f z* = *y* ⟧ ⟹ {*x*. *f x* = *y*} = {*z*}
⟨*proof*⟩

lemma *finite-pre*[*simp*]: *inj f* ⟹ *finite* (*pre f x*)
⟨*proof*⟩

lemma *finite-preImage*[*simp*]: ⟦ *finite A*; *inj f* ⟧ ⟹ *finite* (*preImage f A*)
⟨*proof*⟩

end

3 Formula

theory *Formula*
imports *Base*
begin

3.1 Variables

datatype *vbl* = *X nat*
— FIXME there's a lot of stuff about this datatype that is really just a lifting

from nat (what else could it be). Makes me wonder whether things wouldn't be clearer is we just identified vbls with nats

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

lemma *X-deX*[*simp*]: *X* (*deX a*) = *a*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *inj-nextX*: *inj nextX*
⟨*proof*⟩

lemma *ind'*: *P zeroX* ==> (! *v* . *P v* --> *P* (*nextX v*)) ==> *P v'*
⟨*proof*⟩

lemma *ind*: ⟦ *P zeroX*; ⋀ *v*. *P v* ⟹ *P* (*nextX v*) ⟧ ⟹ *P v'*
⟨*proof*⟩

lemma *zeroX-nextX*[*iff*]: *zeroX* ~= *nextX a* — FIXME iff?
⟨*proof*⟩

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

lemma *nextX*: *nextX* (*X n*) = *X* (*Suc n*)
⟨*proof*⟩

lemma *vblcase-zeroX*[*simp*]: *vblcase a b zeroX* = *a*
⟨*proof*⟩

lemma *vblcase-nextX*[*simp*]: *vblcase a b* (*nextX n*) = *b n*

⟨*proof*⟩

lemma *vbl-cases*: $x = zeroX \mid (?\ y\ .\ x = nextX\ y)$
⟨*proof*⟩

lemma *vbl-casesE*: $[\![\ x = zeroX \Longrightarrow P;\ \bigwedge y.\ x = nextX\ y \Longrightarrow P\ ]\!] \Longrightarrow P$
⟨*proof*⟩

lemma *comp-vblcase*: $f\ o\ vblcase\ a\ b = vblcase\ (f\ a)\ (f\ o\ b)$
⟨*proof*⟩

lemma *equalOn-vblcaseI'*: *equalOn* (*preImage nextX A*) *f g* ==> *equalOn A* (*vblcase x f*) (*vblcase x g*)
⟨*proof*⟩

lemma *equalOn-vblcaseI*: (*zeroX* : *A* −−> *x*=*y*) ==> *equalOn* (*preImage nextX A*) *f g* ==> *equalOn A* (*vblcase x f*) (*vblcase y g*)
⟨*proof*⟩

lemma *X-deX-connection*: *X n* : *A* = (*n* : (*deX* ‘ *A*))
⟨*proof*⟩

lemma *finiteFreshVar*: *finite A* ==> *freshVar A* ~: *A*
⟨*proof*⟩

lemma *freshVarI*: [| *finite A*; *B* <= *A* |] ==> *freshVar A* ~: *B*
⟨*proof*⟩

lemma *freshVarI2*: *finite A* ==> !*x* . *x* ~: *A* −−> *P x* ==> *P* (*freshVar A*)
⟨*proof*⟩

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

3.2 Predicates

datatype *predicate* = *Predicate nat*

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

lemma *signsE*: $[\![\ signs = Neg \Longrightarrow P;\ signs = Pos \Longrightarrow P\ ]\!] \Longrightarrow P$
⟨*proof*⟩

lemma *expand-case-signs*: *Q*(*case-signs vpos vneg F*) = (
(*F* = *Pos* −−> *Q* (*vpos*)) &
(*F* = *Neg* −−> *Q* (*vneg*))
)
⟨*proof*⟩

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* ⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩
lemma *strong-formula-induct'*: !*A*. (! *B*. *size B* < *size A* −−> *P B*) −−> *P A*
==> ! *A*. *size A* = *n* −−> *P* (*A*::*formula*)
⟨*proof*⟩

lemma *strong-formula-induct*: (! *A*. (! *B*. *size B* < *size A* −−> *P B*) −−> *P A*)
==> *P* (*A*::*formula*)
⟨*proof*⟩

lemma *sizelemmas*: *size A* < *size* (*FConj z A B*)

size B < size (FConj z A B)
size A < size (FAll z A)
⟨*proof*⟩

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*))
)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *dualFNot'*: *dual invSign invSign invSign* = *FNot*
⟨*proof*⟩

lemma *dualFNot*: *dual invSign id id* (*FNot A*) = *FNot* (*dual invSign id id A*)
⟨*proof*⟩

lemma *dualId*: *dual id id id A* = *A*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *finite-freeVarsF*[*simp*]: *finite* (*freeVarsF A*)
⟨*proof*⟩

lemma *freeVarsFL-nil*[*simp*]: *freeVarsFL* ([]) = {}
⟨*proof*⟩

lemma *freeVarsFL-cons*: *freeVarsFL* (*A*#*Gamma*) = *freeVarsF A* ∪ *freeVarsFL Gamma*
⟨*proof*⟩

lemma *finite-freeVarsFL*[*simp*]: *finite* (*freeVarsFL gamma*)
⟨*proof*⟩

lemma *freeVarsDual*: *freeVarsF* (*dual p q r A*) = *freeVarsF A*
⟨*proof*⟩

3.6 Substitutions

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

lemma *size-subF*: !!*theta*. *size* (*subF theta A*) = *size* (*A*::*formula*)
⟨*proof*⟩

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

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

definition

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

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

lemma *instanceFDual*: *instanceF u* (*dual p q r A*) = *dual p q r* (*instanceF u A*)
⟨*proof*⟩

3.7 Models

typedecl
object

axiomatization *obj* :: *nat* => *object*
where *inj-obj*: *inj obj*

3.8 model, non empty set and positive atom valuation

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

typedef *model* = *model*
⟨*proof*⟩

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

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

lemma *evalP-arg2-cong*: *x* = *y* ==> *evalP M p x* = *evalP M p y* ⟨*proof*⟩

lemma *objectsNonEmpty*: *objects M* ≠ {}
⟨*proof*⟩

lemma *modelsNonEmptyI*: *fst* (*Rep-model M*) ≠ {}
⟨*proof*⟩

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*))
⟨*proof*⟩

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

lemma *evalF-arg2-cong*: *x* = *y* ==> *evalF M p x* = *evalF M p y* ⟨*proof*⟩

lemma *evalF-FNot*: !!*phi*. *evalF M phi* (*FNot A*) = (¬ *evalF M phi A*)
⟨*proof*⟩

lemma *evalF-equiv*[*rule-format*]: ! *f g*. (*equalOn* (*freeVarsF A*) *f g*) ⟶ (*evalF M f A* = *evalF M g A*)
⟨*proof*⟩
lemma *evalF-subF-eq*: !*phi theta*. *evalF M phi* (*subF theta A*) = *evalF M* (*phi o theta*) *A*
⟨*proof*⟩

lemma *o-id'*[*simp*]: *f o* (% *x*. *x*) = *f*
⟨*proof*⟩

lemma *evalF-instance*: *evalF M phi* (*instanceF u A*) = *evalF M* (*vblcase* (*phi u*) *phi*) *A*
⟨*proof*⟩

lemma *s*[*simp*]: *FConj signs formula1 formula2* ≠ *formula1*
⟨*proof*⟩

lemma *s'*[*simp*]: *FConj signs formula1 formula2* ≠ *formula2*
⟨*proof*⟩

lemma *instanceF-E*: *instanceF g formula* ≠ *FAll signs formula*
⟨*proof*⟩

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 ⟷ (*? f* : *set fs* . *evalF M phi f* = *True*)

lemma *evalS-nil*[*simp*]: *evalS M phi* [] = *False*
⟨*proof*⟩

lemma *evalS-cons*[*simp*]: *evalS M phi* (*A* # *Gamma*) = (*evalF M phi A* | *evalS M phi Gamma*)
⟨*proof*⟩

lemma *evalS-append*: *evalS M phi* (*Gamma* @ *Delta*) = (*evalS M phi Gamma* | *evalS M phi Delta*)
⟨*proof*⟩

lemma *evalS-equiv*[*rule-format*]: (*equalOn* (*freeVarsFL Gamma*) *f g*) −−> (*evalS M f Gamma* = *evalS M g Gamma*)
⟨*proof*⟩

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

lemma *modelAssignsI*: *range f* <= *objects M* ⟹ *f* : *modelAssigns M*
⟨*proof*⟩

lemma *modelAssignsD*: *f* : *modelAssigns M* ⟹ *range f* <= *objects M*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)

⟨*proof*⟩

lemma *DAxiomsI*: [| *DAxioms* <= *A* |] ==> [*FAtom Neg p vs*,*FAtom Pos p vs*] : *deductions*(*A*)

⟨*proof*⟩

lemma *DisjI*: [| *A0*#*A1*#*Gamma* : *deductions*(*A*); *Disjs* <= *A* |] ==> (*FConj Neg A0 A1*#*Gamma*) : *deductions*(*A*)

⟨*proof*⟩

lemma *ConjI*: [| (*A0*#*Gamma*) : *deductions*(*A*); (*A1*#*Delta*) : *deductions*(*A*); *Conjs* <= *A* |] ==> *FConj Pos A0 A1*#*Gamma* @ *Delta* : *deductions*(*A*)

⟨*proof*⟩

lemma *AllI*: [| *instanceF w A*#*Gamma* : *deductions*(*R*); *w* ~: *freeVarsFL* (*FAll Pos A*#*Gamma*); *Alls* <= *R* |] ==> (*FAll Pos A*#*Gamma*) : *deductions*(*R*)

⟨*proof*⟩

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

⟨*proof*⟩

lemma *WeakI*: [| *Gamma* : *deductions R*; *Weaks* <= *R* |] ==> *A*#*Gamma* : *deductions*(*R*)

⟨*proof*⟩

lemma *ContrI*: [| *A*#*A*#*Gamma* : *deductions R*; *Contrs* <= *R* |] ==> *A*#*Gamma* : *deductions*(*R*)

⟨*proof*⟩

lemma *PermI*: [| *Gamma′* : *deductions R*; *mset Gamma* = *mset Gamma′*; *Perms* <= *R* |] ==> *Gamma* : *deductions*(*R*)

⟨*proof*⟩

4.4 Derived Rules

lemma *WeakI1*: [| *Gamma* : *deductions*(*A*); *Weaks* <= *A* |] ==> (*Delta* @ *Gamma*) : *deductions*(*A*)

⟨*proof*⟩

lemma *WeakI2*: [| *Gamma* : *deductions*(*A*); *Perms* <= *A*; *Weaks* <= *A* |] ==> (*Gamma* @ *Delta*) : *deductions*(*A*)
⟨*proof*⟩

lemma *SATAxiomI*: [| *Axioms* <= *A*; *Weaks* <= *A*; *Perms* <= *A*; *forms* = [*FAtom Pos n vs*,*FAtom Neg n vs*] @ *Gamma* |] ==> *forms* : *deductions*(*A*)
⟨*proof*⟩

lemma *DisjI1*: [| (*A1*#*Gamma*) : *deductions*(*A*); *Disjs* <= *A*; *Weaks* <= *A* |] ==> *FConj Neg A0 A1*#*Gamma* : *deductions*(*A*)
⟨*proof*⟩

lemma *DisjI2*: !!*A*. [| (*A0*#*Gamma*) : *deductions*(*A*); *Disjs* <= *A*; *Weaks* <= *A*; *Perms* <= *A* |] ==> *FConj Neg A0 A1*#*Gamma* : *deductions*(*A*)
⟨*proof*⟩
lemma *perm-tmp4*: *Perms* ⊆ *R* ⟹ *A* @ (*a* # *list*) @ (*a* # *list*) : *deductions R* ⟹ (*a* # *a* # *A*) @ *list* @ *list* : *deductions R*
⟨*proof*⟩

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

lemma *ListWeakI*: *Perms* <= *R* ==> *Contrs* <= *R* ==> *x* # *Gamma* @ *Gamma* : *deductions*(*R*) ==> *x* # *Gamma* : *deductions*(*R*)
⟨*proof*⟩

lemma *ConjI'*: [| (*A0*#*Gamma*) : *deductions*(*A*); (*A1*#*Gamma*) : *deductions*(*A*); *Contrs* <= *A*; *Conjs* <= *A*; *Perms* <= *A* |] ==> *FConj Pos A0 A1*#*Gamma* : *deductions*(*A*)
⟨*proof*⟩

4.5 Standard Rule Sets For Predicate Calculus

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

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

lemma *rulesInPCs*: *Axioms* <= *PC Axioms* <= *CutFreePC*
Conjs <= *PC Conjs* <= *CutFreePC*
Disjs <= *PC Disjs* <= *CutFreePC*
Alls <= *PC Alls* <= *CutFreePC*
Exs <= *PC Exs* <= *CutFreePC*

Weaks <= *PC Weaks* <= *CutFreePC*
Contrs <= *PC Contrs* <= *CutFreePC*
Perms <= *PC Perms* <= *CutFreePC*
Cuts <= *PC*
CutFreePC <= *PC*
⟨*proof*⟩

4.6 Monotonicity for CutFreePC deductions

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

lemma *perm*: ! *xs ys*. *mset xs* = *mset ys* −−> (*inDed xs* = *inDed ys*)
⟨*proof*⟩

lemma *contr*: ! *x xs*. *inDed* (*x*#*x*#*xs*) −−> *inDed* (*x*#*xs*)
⟨*proof*⟩

lemma *weak*: ! *x xs*. *inDed xs* −−> *inDed* (*x*#*xs*)
⟨*proof*⟩

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

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

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

4.7 Tree

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

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

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

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

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

⟨*proof*⟩

lemma *tree1Eq* [*rule-format*]:
∀ *Y*. (*Suc n*, *Y*) ∈ *tree subs gamma* = (∃ *sigma* ∈ *subs gamma* . (*n*, *Y*) ∈ *tree subs sigma*)
⟨*proof*⟩

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

lemma *injIncLevel*: *inj incLevel*
⟨*proof*⟩

lemma *treeEquation*: *tree subs gamma* = *insert* (*0*,*gamma*) (*UN delta*:*subs gamma* . *incLevel* ‘ *tree subs delta*)
⟨*proof*⟩

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

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

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

4.8 Terminal

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

lemma *terminalD*: *terminal subs Gamma* ==> *x* ~: *subs Gamma*
⟨*proof*⟩

lemma *terminalI*: *x* ∈ *subs Gamma* ==> ~ *terminal subs Gamma*
⟨*proof*⟩

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)*
⟨*proof*⟩

lemma *inheritedIncLevel*[*rule-format*]: *inherited subs P −−> P A −−> P (incLevel ‘ A)*
⟨*proof*⟩

lemma *inheritedEmpty*[*rule-format*]: *inherited subs P −−> P {}*
⟨*proof*⟩

lemma *inheritedInsert*[*rule-format*]:
inherited subs P −−> ~(terminal subs Gamma) −−> P A −−> P (insert (n,Gamma) A)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *inheritedUnEq*[*rule-format*, *symmetric*]: *inherited subs P −−> (P A & P B) = P (A Un B)*
⟨*proof*⟩

lemma *inheritedIncLevelEq*[*rule-format*, *symmetric*]: *inherited subs P −−> P A = P (incLevel ‘ A)*
⟨*proof*⟩

lemma *inheritedInsertEq*[*rule-format*, *symmetric*]: *inherited subs P −−> ~(terminal subs Gamma) −−> P A = P (insert (n,Gamma) A)*
⟨*proof*⟩

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

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

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

lemma *inheritedUNEq*[*rule-format*]:
finite A −−> inherited subs P −−> (!x:A. P (B x)) = P (UN a:A. B a)
⟨*proof*⟩

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

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

lemma *inheritedPropagateEq*[*rule-format*]: **assumes** *a*: *inherited subs P*
and *b*: *fans subs*
and *c*: ~(*terminal subs delta*)
shows *P*(*tree subs delta*) = (!*sigma*:*subs delta*. *P*(*tree subs sigma*))
⟨*proof*⟩

lemma *inheritedPropagate*:
[| ~*P*(*tree subs delta*); *inherited subs P*; *fans subs*; ~(*terminal subs delta*)|]
==> ∃ *sigma* ∈ *subs delta* . ~*P*(*tree subs sigma*)
⟨*proof*⟩

lemma *inheritedViaSub*: [| *inherited subs P*; *fans subs*; *P*(*tree subs delta*); *sigma* ∈ *subs delta* |]
==> *P*(*tree subs sigma*)
⟨*proof*⟩

lemma *inheritedJoin*[*rule-format*]:
(*inherited subs P* & *inherited subs Q*) −−> *inherited subs* (%*x*. *P x* & *Q x*)
⟨*proof*⟩

lemma *inheritedJoinI*[*rule-format*]: [| *inherited subs P*; *inherited subs Q*; *R* = (% *x* . *P x* & *Q x*) |] ==> *inherited subs R*
⟨*proof*⟩

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* {}
⟨*proof*⟩

lemma *boundedByInsert*: *boundedBy N* (*insert* (*n*,*delta*) *B*) = (*n* < *N* & *boundedBy N B*)
⟨*proof*⟩

lemma *boundedByUn*: *boundedBy N* (*A Un B*) = (*boundedBy N A* & *boundedBy N B*)
⟨*proof*⟩

lemma *boundedByIncLevel'*: *boundedBy* (*Suc N*) (*incLevel* ‘ *A*) = *boundedBy N A*
⟨*proof*⟩

lemma *boundedByAdd1*: *boundedBy N B* ⟹ *boundedBy* (*N*+*M*) *B*
⟨*proof*⟩

lemma *boundedByAdd2*: *boundedBy M B* ⟹ *boundedBy* (*N*+*M*) *B*
⟨*proof*⟩

lemma *boundedByMono*: *boundedBy m B* ⟹ *m* < *M* ⟹ *boundedBy M B*
⟨*proof*⟩

lemmas *boundedByMonoD* = *boundedByMono*

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

lemma *boundedBySuc'*: *boundedBy N A* ⟹ *boundedBy* (*Suc N*) *A*
⟨*proof*⟩

lemma *boundedByIncLevel*: *boundedBy n* (*incLevel* ‘ (*tree subs gamma*)) = (∃ *m* . *n* = *Suc m* & *boundedBy m* (*tree subs gamma*))
⟨*proof*⟩

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

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

4.11 Inherited Properties- bounded

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

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

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

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

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

4.12 founded

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

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

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

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

4.13 Inherited Properties- founded

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

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

lemma *foundedIncLevel*: *founded subs P* (*incLevel* ‘ *A*) = (*founded subs P A*)
⟨*proof*⟩

lemma *foundedEmpty*: *founded subs P* {}
⟨*proof*⟩

lemma *inheritedFounded*: *inherited subs* (*founded subs P*)
⟨*proof*⟩

4.14 Inherited Properties- finite

lemmas *finiteInsert* = *finite-insert*

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

lemma *finiteIncLevel*: *finite* (*incLevel* ‘ *A*) = *finite A*
⟨*proof*⟩

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

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

⟨*proof*⟩

4.15 path: follows a failing inherited property through tree

definition
failingSub :: ['*a* => '*a* *set*,(*nat* * '*a*) *set* => *bool*,'*a*] => '*a* **where**
failingSub subs P gamma = (*SOME sigma*. (*sigma*:*subs gamma* & ~*P*(*tree subs sigma*)))

lemma *failingSubProps*: [| *inherited subs P*; ~*P* (*tree subs gamma*); ~(*terminal subs gamma*); *fans subs* |]
==> *failingSub subs P gamma* ∈ *subs gamma* & ~(*P* (*tree subs* (*failingSub subs P gamma*)))
⟨*proof*⟩

lemma *failingSubFailsI*: [| *inherited subs P*; ~*P* (*tree subs gamma*); ~(*terminal subs gamma*); *fans subs* |]
==> ~(*P* (*tree subs* (*failingSub subs P gamma*)))
⟨*proof*⟩

lemmas *failingSubFailsE* = *failingSubFailsI*[*THEN notE*]

lemma *failingSubSubs*: [| *inherited subs P*; ~*P* (*tree subs gamma*); ~(*terminal subs gamma*); *fans subs* |]
==> *failingSub subs P gamma* ∈ *subs gamma*
⟨*proof*⟩

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*)))
⟨*proof*⟩

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

lemma *pathTerminal*[*rule-format*]: [| *inherited subs P*; *fans subs*; *terminal subs gamma* |]
==> *terminal subs* (*path subs gamma P n*)
⟨*proof*⟩

lemma *pathStarts*: *path subs gamma P 0* = *gamma*
⟨*proof*⟩

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*)
  ⟨*proof*⟩

lemma *pathStops*: *terminal subs* (*path subs gamma P n*) ==> *path subs gamma P* (*Suc n*) = *path subs gamma P n*
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

lemma *branchStops*: *branch subs Gamma f* ==> *terminal subs* (*f n*) ==> *f* (*Suc n*) = *f n*
  ⟨*proof*⟩

lemma *branchSubs*: *branch subs Gamma f* ==> ~ *terminal subs* (*f n*) ==> *f* (*Suc n*) ∈ *subs* (*f n*)
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

lemma *branchTerminalPropagates*: *branch subs Gamma f* ==> *terminal subs* (*f m*) ==> *terminal subs* (*f* (*m* + *n*))
  ⟨*proof*⟩

lemma *branchTerminalMono*: *branch subs Gamma f* ==> *m* < *n* ==> *terminal subs* (*f m*) ==> *terminal subs* (*f n*)
  ⟨*proof*⟩

lemma *branchPath*:
      [| *inherited subs P*; *fans subs*; ~*P*(*tree subs gamma*) |]
       ==> *branch subs gamma* (*path subs gamma P*)
  ⟨*proof*⟩

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*)))
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *boundedTreeInduction*:
⟦*fans subs*;
bounded (*tree subs Gamma*); *founded subs P* (*tree subs Gamma*);
∀ *delta*. ~ *terminal subs delta* −−> (∀ *sigma* ∈ *subs delta*. *P sigma*) −−> *P delta* ⟧ ⟹ *P Gamma*
⟨*proof*⟩

lemma *boundedTreeInduction2'*:
[| *fans subs*;
∀ *delta*. (∀ *sigma* ∈ *subs delta*. *P sigma*) −−> *P delta* |]
==> ∀ *Gamma*. *boundedBy m* (*tree subs Gamma*) ⟶ *P Gamma*
⟨*proof*⟩

lemma *boundedTreeInduction2*:
[| *fans subs*; *boundedBy m* (*tree subs Gamma*);
∀ *delta*. (∀ *sigma* ∈ *subs delta*. *P sigma*) −−> *P delta* |]
==> *P Gamma*
⟨*proof*⟩

end

5 Completeness

theory *Completeness*
imports *Tree Sequents*
begin

5.1 pseq: type represents a processed sequent

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*)
⟨*proof*⟩

lemma *fansSubs*: *fans subs*
⟨*proof*⟩

lemma *subs-def2*:
!!*gamma*.
~ *SATAxiom* (*sequent gamma*) ==>
subs gamma = (*case nforms gamma of*
[] => {}
| *nA*#*nAs* => *let* (*n*,*A*) = *nA*
in (*case A of*
FAtom z P vs => *subsFAtom* (*atoms gamma*) *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*))))
⟨*proof*⟩

5.9 inherited: proofTree

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

lemma *inheritedProofTree*: *inherited subs proofTree*
⟨*proof*⟩

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

lemma *proofTreeBounded*: *proofTree A* ==> *bounded A*
⟨*proof*⟩

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

5.10 pseq: lemma

lemma *snd-o-Pair*: *(snd o (Pair x)) = (% x. x)*
⟨*proof*⟩

lemma *sequent-pseq*: *sequent (pseq fs) = fs*
⟨*proof*⟩

5.11 SATAxiom: proofTree

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

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

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

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

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

5.12 SATAxioms are deductions: - needed

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

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

lemma *subsJustified′*:
notes *ss = subs-def2 nforms-def Let-def atoms-def sequent-def subsFAtom-def subsFConj-def subsFAll-def*
shows ¬ *SATAxiom (sequent (ats, (n, f) # list)) −−> ¬ terminal subs (ats, (n, f) # list)*
−−> (∀ sigma∈subs (ats, (n, f) # list). sequent sigma ∈ deductions CutFreePC)

−−> sequent (ats, (n, f) # list) ∈ deductions CutFreePC

⟨*proof*⟩

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

5.14 proofTrees are deductions: instance of boundedTreeInduction

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

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

5.15 contains, considers:

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

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

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

5.16 path: nforms = [] implies

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

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

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

5.17 path: cases

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

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

5.18 path: contains not terminal and propagate condition

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

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

5.19 path: no consider lemmas

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

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

5.20 path: contains initially

lemma *contains-initially*:
branch subs (*pseq gamma*) *f* ⟹ *A* : *set gamma* ⟹ (*contains f* (*0*,*A*) *0*)
⟨*proof*⟩

lemma *contains-initialEVs*:
branch subs (*pseq gamma*) *f* ⟹ *A* : *set gamma* ⟹ *EV* (*contains f* (*0*,*A*))
⟨*proof*⟩

5.21 termination: (for EV contains implies EV considers)

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

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*)
⟨*proof*⟩

5.22 costBarrier: lemmas

5.23 costBarrier: exp3 lemmas - bit specific...

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

⟨*proof*⟩

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

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

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

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

5.24 costBarrier: decreases whilst contains and unconsiders

lemma *costBarrierDecreases'*:
notes *ss = subs-def2 nforms-def Let-def subsFAtom-def subsFConj-def subsFAll-def costBarrier-def atoms-def exp3Min expSum*
shows *~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)*
⟨*proof*⟩

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)*
⟨*proof*⟩

5.25 path: EV contains implies EV considers

lemma *considersContains*: *considers f iA n ⟹ contains f iA n*
⟨*proof*⟩

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

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.* $\sim$*SATAxiom (sequent (f n)) & (nforms (f n) = (i,A) # nAs & f (Suc n) : subs (f n))*
⟨*proof*⟩

5.27 EV contains: FConj,FDisj,FAll

lemma *EV-disj*: *(EV P | EV Q) = EV (λn. P n | Q n)*
⟨*proof*⟩

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))*
⟨*proof*⟩

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))*
⟨*proof*⟩

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))
⟨*proof*⟩

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))
⟨*proof*⟩

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))
⟨*proof*⟩

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)*
⟨*proof*⟩

lemma *lemma2*:
[| P A n ; P B m ; ! A n. P A n --> P A (Suc n) |] ==> ? n . P A n & P B n
⟨*proof*⟩

5.29 EV contains: FAtoms

lemma *notTerminalNotSATAxiom*: $\neg$ *terminal subs gamma* $\Longrightarrow$ $\neg$ *SATAxiom* (*sequent gamma*)
⟨*proof*⟩

lemma *notTerminalNforms*: $\neg$ *terminal subs* (*f n*) $\Longrightarrow$ *nforms* (*f n*) $\neq$ []
⟨*proof*⟩

lemma *atomsPropagate*: [| *branch subs gamma f* |]
==> *x* : *set* (*atoms* (*f n*)) −−> *x* : *set* (*atoms* (*f* (*Suc n*)))
⟨*proof*⟩

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*))
⟨*proof*⟩

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*))
⟨*proof*⟩

5.31 pseq: creates initial pseq

lemma *containsPSeq0D*: *branch subs* (*pseq fs*) *f* $\Longrightarrow$ *contains f* (*i*,*A*) *0* $\Longrightarrow$ *i=0*
⟨*proof*⟩

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

lemma *claim*: (*A* | *B* | *C*) = (~*C* −−> ~*B* −−> *A*) ⟨*proof*⟩

lemma *natPredCases*: (!*n*. *P n*) | (~*P 0*) | (*?* *n* . *P n* & ~ *P* (*Suc n*))
⟨*proof*⟩

lemma *containsNotTerminal′*:
⟦ *branch subs gamma f*; !*n* . ~*proofTree* (*tree subs* (*f n*)); *contains f iA n* ⟧ $\Longrightarrow$ ~ (*terminal subs* (*f n*))
⟨*proof*⟩

lemma *notTerminalSucNotTerminal*: ⟦ $\neg$ *terminal subs* (*f* (*Suc n*)); *branch subs gamma f* ⟧ $\Longrightarrow$ $\neg$ *terminal subs* (*f n*)
⟨*proof*⟩

lemma *evContainsExSuc-containsEx*:

[| *branch subs* (*pseq fs*) *f*; !*n* . ~ *proofTree* (*tree subs* (*f n*));
EV (*contains f* (*Suc i*,*FAll Neg body*)) |]
==> *EV* (*contains f* (*i*,*FAll Neg body*))
⟨*proof*⟩

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

lemma *evContainsEx-containsEx0*:
[| *branch subs* (*pseq fs*) *f*; !*n* . ~ *proofTree* (*tree subs* (*f n*)) |]
==> *EV* (*contains f* (*i*,*FAll Neg A*)) −−>
EV (*contains f* (*0*,*FAll Neg A*))
⟨*proof*⟩

lemma *evContainsExval*:
[| *EV* (*contains f* (*i*,*FAll Neg body*)); *branch subs* (*pseq fs*) *f*; !*n* . ~ *proofTree* (*tree subs* (*f n*))
|]
==> ! *v* . *EV* (*contains f* (*0*,*instanceF v body*))
⟨*proof*⟩

5.34 EV contains: atoms

lemma *atomsInSequentI*[*rule-format*]: (*z*,*P*,*vs*) : *set* (*fst ps*) −−>
FAtom z P vs : *set* (*sequent ps*)
⟨*proof*⟩

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*))
⟨*proof*⟩

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*))))
⟨*proof*⟩

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*))
⟨*proof*⟩

5.35 counterModel: lemmas

lemma *counterModelInRepn*: (*counterM f*,*counterEvalP f*) : *model*
⟨*proof*⟩

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

lemma *inv-obj-obj*: *inv obj* (*obj n*) = *n*
⟨*proof*⟩

lemma *map-X-map-counterAssign*: *map X* (*map* (*inv obj*) (*map counterAssign xs*))
= *xs*
⟨*proof*⟩

lemma *objectsCounterModel*: *objects* (*counterModel f*) = { *z* . *? i* . *z* = *obj i* }
⟨*proof*⟩

lemma *inCounterM*: *counterAssign v* : *objects* (*counterModel f*)
⟨*proof*⟩

lemma *counterAssign-eqI*[*rule-format*]: *x* : *objects* (*counterModel f*) −−> *z* = *X*
(*inv obj x*) −−> *counterAssign z* = *x*
⟨*proof*⟩

lemma *evalPCounterModel*: *M* = *counterModel f* ==> *evalP M* = *counterEvalP*
f
⟨*proof*⟩

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)
⟨*proof*⟩

lemmas *path-evalF″* = *mp*[*OF path-evalF′*]

5.37 adequacy

lemma *counterAssignModelAssign*: *counterAssign* : *modelAssigns* (*counterModel*
gamma)
⟨*proof*⟩

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

⟨*proof*⟩

lemma *path-evalF*:
[| *branch subs* (*pseq fs*) *f*;
∀ *n*. ¬ *proofTree* (*tree subs* (*f n*));
x ∈ *set fs*
|] ==> ¬ *evalF* (*counterModel f*) *counterAssign x*
⟨*proof*⟩

lemma *validProofTree*: ~*proofTree* (*tree subs* (*pseq fs*)) ==> ~(*validS fs*)
⟨*proof*⟩

lemma *adequacy*[*simplified sequent-pseq*]: *validS fs* ==> (*sequent* (*pseq fs*)) : *deductions CutFreePC*
⟨*proof*⟩

end

6 Soundness

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

lemma *permutation-validS*: *mset fs* = *mset gs* −−> (*validS fs* = *validS gs*)
⟨*proof*⟩

lemma *modelAssigns-vblcase*: *phi* ∈ *modelAssigns M* ⟹ *x* ∈ *objects M* ⟹ *vblcase x phi* ∈ *modelAssigns M*
⟨*proof*⟩

lemma *tmp*: (!*x* : *A*. *P x* | *Q*) ==> (! *x* : *A*. *P x*) | *Q* ⟨*proof*⟩

lemma *soundnessFAll*: !!*Gamma*.
[| *u* ~: *freeVarsFL* (*FAll Pos A* # *Gamma*);
validS (*instanceF u A* # *Gamma*) |]
==> *validS* (*FAll Pos A* # *Gamma*)
⟨*proof*⟩

lemma *soundnessFEx*: *validS* (*instanceF x A* # *Gamma*) ==> *validS* (*FAll Neg A* # *Gamma*)
⟨*proof*⟩

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

⟨*proof*⟩

lemma *soundness*: *fs* : *deductions*(*PC*) ==> (*validS fs*)
⟨*proof*⟩

lemma *completeness*: $fs : deductions\ (PC) = validS\ fs$
$\langle proof \rangle$

end