A Sequent Calculus Prover for First-Order Logic with Functions

Asta Halkjær From Frederik Krogsdal Jacobsen

March 17, 2025

Abstract

We formalize an automated theorem prover for first-order logic with functions. The proof search procedure is based on sequent calculus and we verify its soundness and completeness using the Abstract Soundness and Abstract Completeness theories. Our analytic completeness proof covers both open and closed formulas. Since our deterministic prover considers only the subset of terms relevant to proving a given sequent, we do so as well when building a countermodel from a failed proof. We formally connect our prover with the proof system and semantics of the existing SeCaV system. In particular, the prover's output can be post-processed in Haskell to generate human-readable SeCaV proofs which are also machine-verifiable proof certificates.

Contents

Chapter 1

SeCaV

1.1 Sequent Calculus Verifier (SeCaV)

theory *SeCaV* **imports** *Main* **begin**

1.2 Syntax: Terms / Formulas

datatype *tm = Fun nat ‹tm list› | Var nat*

datatype *fm = Pre nat ‹tm list› | Imp fm fm | Dis fm fm | Con fm fm | Exi fm | Uni fm | Neg fm*

1.3 Semantics: Terms / Formulas

definition ‹*shift e v x* ≡ λ*n. if n < v then e n else if n = v then x else e (n − 1)*›

primrec *semantics-term* **and** *semantics-list* **where**
‹*semantics-term e f (Var n) = e n*› |
‹*semantics-term e f (Fun i l) = f i (semantics-list e f l)*› |
‹*semantics-list e f [] = []*› |
‹*semantics-list e f (t # l) = semantics-term e f t # semantics-list e f l*›

primrec *semantics* **where**
‹*semantics e f g (Pre i l) = g i (semantics-list e f l)*› |
‹*semantics e f g (Imp p q) = (semantics e f g p* ⟶ *semantics e f g q)*› |
‹*semantics e f g (Dis p q) = (semantics e f g p* ∨ *semantics e f g q)*› |
‹*semantics e f g (Con p q) = (semantics e f g p* ∧ *semantics e f g q)*› |
‹*semantics e f g (Exi p) = (*∃ *x. semantics (shift e 0 x) f g p)*› |
‹*semantics e f g (Uni p) = (*∀ *x. semantics (shift e 0 x) f g p)*› |
‹*semantics e f g (Neg p) = (*¬ *semantics e f g p)*›

— Test

corollary ‹*semantics e f g (Imp (Pre 0 []) (Pre 0 []))*›

by *simp*

lemma ‹¬ *semantics e f g* (*Neg* (*Imp* (*Pre 0* []) (*Pre 0* [])))›
by *simp*

1.4 Auxiliary Functions

primrec *new-term* **and** *new-list* **where**
‹*new-term c* (*Var n*) = *True*› |
‹*new-term c* (*Fun i l*) = (*if i* = *c then False else new-list c l*)› |
‹*new-list c* [] = *True*› |
‹*new-list c* (*t* # *l*) = (*if new-term c t then new-list c l else False*)›

primrec *new* **where**
‹*new c* (*Pre i l*) = *new-list c l*› |
‹*new c* (*Imp p q*) = (*if new c p then new c q else False*)› |
‹*new c* (*Dis p q*) = (*if new c p then new c q else False*)› |
‹*new c* (*Con p q*) = (*if new c p then new c q else False*)› |
‹*new c* (*Exi p*) = *new c p*› |
‹*new c* (*Uni p*) = *new c p*› |
‹*new c* (*Neg p*) = *new c p*›

primrec *news* **where**
‹*news c* [] = *True*› |
‹*news c* (*p* # *z*) = (*if new c p then news c z else False*)›

primrec *inc-term* **and** *inc-list* **where**
‹*inc-term* (*Var n*) = *Var* (*n* + *1*)› |
‹*inc-term* (*Fun i l*) = *Fun i* (*inc-list l*)› |
‹*inc-list* [] = []› |
‹*inc-list* (*t* # *l*) = *inc-term t* # *inc-list l*›

primrec *sub-term* **and** *sub-list* **where**
‹*sub-term v s* (*Var n*) = (*if n* < *v then Var n else if n* = *v then s else Var* (*n* − *1*))› |
‹*sub-term v s* (*Fun i l*) = *Fun i* (*sub-list v s l*)› |
‹*sub-list v s* [] = []› |
‹*sub-list v s* (*t* # *l*) = *sub-term v s t* # *sub-list v s l*›

primrec *sub* **where**
‹*sub v s* (*Pre i l*) = *Pre i* (*sub-list v s l*)› |
‹*sub v s* (*Imp p q*) = *Imp* (*sub v s p*) (*sub v s q*)› |
‹*sub v s* (*Dis p q*) = *Dis* (*sub v s p*) (*sub v s q*)› |
‹*sub v s* (*Con p q*) = *Con* (*sub v s p*) (*sub v s q*)› |
‹*sub v s* (*Exi p*) = *Exi* (*sub* (*v* + *1*) (*inc-term s*) *p*)› |
‹*sub v s* (*Uni p*) = *Uni* (*sub* (*v* + *1*) (*inc-term s*) *p*)› |
‹*sub v s* (*Neg p*) = *Neg* (*sub v s p*)›

primrec *member* **where**

‹member p [] = False› |
‹member p (q # z) = (if p = q then True else member p z)›

primrec *ext* **where**
‹ext y [] = True› |
‹ext y (p # z) = (if member p y then ext y z else False)›

— Simplifications

lemma *member* [*iff*]: ‹member p z ⟷ p ∈ set z›
by (*induct z*) *simp-all*

lemma *ext* [*iff*]: ‹ext y z ⟷ set z ⊆ set y›
by (*induct z*) *simp-all*

1.5 Sequent Calculus

inductive *sequent-calculus* (‹⊩ -› *0*) **where**
‹⊩ p # z› **if** ‹member (Neg p) z› |
‹⊩ Dis p q # z› **if** ‹⊩ p # q # z› |
‹⊩ Imp p q # z› **if** ‹⊩ Neg p # q # z› |
‹⊩ Neg (Con p q) # z› **if** ‹⊩ Neg p # Neg q # z› |
‹⊩ Con p q # z› **if** ‹⊩ p # z› **and** ‹⊩ q # z› |
‹⊩ Neg (Imp p q) # z› **if** ‹⊩ p # z› **and** ‹⊩ Neg q # z› |
‹⊩ Neg (Dis p q) # z› **if** ‹⊩ Neg p # z› **and** ‹⊩ Neg q # z› |
‹⊩ Exi p # z› **if** ‹⊩ sub 0 t p # z› |
‹⊩ Neg (Uni p) # z› **if** ‹⊩ Neg (sub 0 t p) # z› |
‹⊩ Uni p # z› **if** ‹⊩ sub 0 (Fun i []) p # z› **and** ‹news i (p # z)› |
‹⊩ Neg (Exi p) # z› **if** ‹⊩ Neg (sub 0 (Fun i []) p) # z› **and** ‹news i (p # z)› |
‹⊩ Neg (Neg p) # z› **if** ‹⊩ p # z› |
‹⊩ y› **if** ‹⊩ z› **and** ‹ext y z›

— Test

corollary ‹⊩ [Imp (Pre 0 []) (Pre 0 [])]›
using *sequent-calculus.intros(1,3,13) ext.simps member.simps(2)* **by** *metis*

1.6 Shorthands

lemmas *Basic* = *sequent-calculus.intros(1)*

lemmas *AlphaDis* = *sequent-calculus.intros(2)*
lemmas *AlphaImp* = *sequent-calculus.intros(3)*
lemmas *AlphaCon* = *sequent-calculus.intros(4)*

lemmas *BetaCon* = *sequent-calculus.intros(5)*
lemmas *BetaImp* = *sequent-calculus.intros(6)*
lemmas *BetaDis* = *sequent-calculus.intros(7)*

```
lemmas GammaExi = sequent-calculus.intros(8)
lemmas GammaUni = sequent-calculus.intros(9)

lemmas DeltaUni = sequent-calculus.intros(10)
lemmas DeltaExi = sequent-calculus.intros(11)

lemmas Neg = sequent-calculus.intros(12)

lemmas Ext = sequent-calculus.intros(13)

— Test

lemma ‹⊩
  [
    Imp (Pre 0 []) (Pre 0 [])
  ]
  ›
proof −
  from AlphaImp have ?thesis if ‹⊩
    [
      Neg (Pre 0 []),
      Pre 0 []
    ]
    ›
    using that by simp
  with Ext have ?thesis if ‹⊩
    [
      Pre 0 [],
      Neg (Pre 0 [])
    ]
    ›
    using that by simp
  with Basic show ?thesis
    by simp
qed
```

1.7 Appendix: Soundness

1.7.1 Increment Function

```
primrec liftt :: ‹tm ⇒ tm› and liftts :: ‹tm list ⇒ tm list› where
  ‹liftt (Var i) = Var (Suc i)› |
  ‹liftt (Fun a ts) = Fun a (liftts ts)› |
  ‹liftts [] = []› |
  ‹liftts (t # ts) = liftt t # liftts ts›
```

1.7.2 Parameters: Terms

primrec *paramst* :: ‹*tm* ⇒ *nat set*› **and** *paramsts* :: ‹*tm list* ⇒ *nat set*› **where**
‹*paramst* (*Var n*) = {}› |
‹*paramst* (*Fun a ts*) = {*a*} ∪ *paramsts ts*› |
‹*paramsts* [] = {}› |
‹*paramsts* (*t* # *ts*) = (*paramst t* ∪ *paramsts ts*)›

lemma *p0* [*simp*]: ‹*paramsts ts* = ⋃(*set* (*map paramst ts*))›
by (*induct ts*) *simp-all*

primrec *paramst′* :: ‹*tm* ⇒ *nat set*› **where**
‹*paramst′* (*Var n*) = {}› |
‹*paramst′* (*Fun a ts*) = {*a*} ∪ ⋃(*set* (*map paramst′ ts*))›

lemma *p1* [*simp*]: ‹*paramst′ t* = *paramst t*›
by (*induct t*) *simp-all*

1.7.3 Parameters: Formulas

primrec *params* :: ‹*fm* ⇒ *nat set*› **where**
‹*params* (*Pre b ts*) = *paramsts ts*› |
‹*params* (*Imp p q*) = *params p* ∪ *params q*› |
‹*params* (*Dis p q*) = *params p* ∪ *params q*› |
‹*params* (*Con p q*) = *params p* ∪ *params q*› |
‹*params* (*Exi p*) = *params p*› |
‹*params* (*Uni p*) = *params p*› |
‹*params* (*Neg p*) = *params p*›

primrec *params′* :: ‹*fm* ⇒ *nat set*› **where**
‹*params′* (*Pre b ts*) = ⋃(*set* (*map paramst′ ts*))› |
‹*params′* (*Imp p q*) = *params′ p* ∪ *params′ q*› |
‹*params′* (*Dis p q*) = *params′ p* ∪ *params′ q*› |
‹*params′* (*Con p q*) = *params′ p* ∪ *params′ q*› |
‹*params′* (*Exi p*) = *params′ p*› |
‹*params′* (*Uni p*) = *params′ p*› |
‹*params′* (*Neg p*) = *params′ p*›

lemma *p2* [*simp*]: ‹*params′ p* = *params p*›
by (*induct p*) *simp-all*

fun *paramst″* :: ‹*tm* ⇒ *nat set*› **where**
‹*paramst″* (*Var n*) = {}› |
‹*paramst″* (*Fun a ts*) = {*a*} ∪ (⋃*t* ∈ *set ts*. *paramst″ t*)›

lemma *p1′* [*simp*]: ‹*paramst″ t* = *paramst t*›
by (*induct t*) *simp-all*

fun *params″* :: ‹*fm* ⇒ *nat set*› **where**
‹*params″* (*Pre b ts*) = (⋃*t* ∈ *set ts*. *paramst″ t*)› |

‹params'' (Imp p q) = params'' p ∪ params'' q› |
‹params'' (Dis p q) = params'' p ∪ params'' q› |
‹params'' (Con p q) = params'' p ∪ params'' q› |
‹params'' (Exi p) = params'' p› |
‹params'' (Uni p) = params'' p› |
‹params'' (Neg p) = params'' p›

lemma *p2'* [*simp*]: ‹params'' p = params p›
by (*induct p*) *simp-all*

1.7.4 Update Lemmas

lemma *upd-lemma'* [*simp*]:
‹n ∉ paramst t ⟹ semantics-term e (f(n := z)) t = semantics-term e f t›
‹n ∉ paramsts ts ⟹ semantics-list e (f(n := z)) ts = semantics-list e f ts›
by (*induct t* **and** *ts rule*: *paramst.induct paramsts.induct*) *auto*

lemma *upd-lemma* [*iff*]: ‹n ∉ params p ⟹ semantics e (f(n := z)) g p ⟷ semantics e f g p›
by (*induct p arbitrary*: *e*) *simp-all*

1.7.5 Substitution

primrec *substt* :: ‹tm ⇒ tm ⇒ nat ⇒ tm› **and** *substts* :: ‹tm list ⇒ tm ⇒ nat ⇒ tm list› **where**
‹substt (Var i) s k = (if k < i then Var (i − 1) else if i = k then s else Var i)› |
‹substt (Fun a ts) s k = Fun a (substts ts s k)› |
‹substts [] s k = []› |
‹substts (t # ts) s k = substt t s k # substts ts s k›

primrec *subst* :: ‹fm ⇒ tm ⇒ nat ⇒ fm› **where**
‹subst (Pre b ts) s k = Pre b (substts ts s k)› |
‹subst (Imp p q) s k = Imp (subst p s k) (subst q s k)› |
‹subst (Dis p q) s k = Dis (subst p s k) (subst q s k)› |
‹subst (Con p q) s k = Con (subst p s k) (subst q s k)› |
‹subst (Exi p) s k = Exi (subst p (liftt s) (Suc k))› |
‹subst (Uni p) s k = Uni (subst p (liftt s) (Suc k))› |
‹subst (Neg p) s k = Neg (subst p s k)›

lemma *shift-eq* [*simp*]: ‹i = j ⟹ (shift e i T) j = T› **for** *i* :: *nat*
unfolding *shift-def* **by** *simp*

lemma *shift-gt* [*simp*]: ‹j < i ⟹ (shift e i T) j = e j› **for** *i* :: *nat*
unfolding *shift-def* **by** *simp*

lemma *shift-lt* [*simp*]: ‹i < j ⟹ (shift e i T) j = e (j − 1)› **for** *i* :: *nat*
unfolding *shift-def* **by** *simp*

lemma *shift-commute* [*simp*]: ‹shift (shift e i U) 0 T = shift (shift e 0 T) (Suc i) U›

unfolding *shift-def* **by** *force*

lemma *subst-lemma′* [*simp*]:
‹*semantics-term e f* (*substt t u i*) = *semantics-term* (*shift e i* (*semantics-term e f u*)) *f t*›
‹*semantics-list e f* (*substts ts u i*) = *semantics-list* (*shift e i* (*semantics-term e f u*)) *f ts*›
by (*induct t* **and** *ts rule*: *substt.induct substts.induct*) *simp-all*

lemma *lift-lemma* [*simp*]:
‹*semantics-term* (*shift e 0 x*) *f* (*liftt t*) = *semantics-term e f t*›
‹*semantics-list* (*shift e 0 x*) *f* (*liftts ts*) = *semantics-list e f ts*›
by (*induct t* **and** *ts rule*: *liftt.induct liftts.induct*) *simp-all*

lemma *subst-lemma* [*iff*]:
‹*semantics e f g* (*subst a t i*) ⟷ *semantics* (*shift e i* (*semantics-term e f t*)) *f g a*›
by (*induct a arbitrary*: *e i t*) *simp-all*

1.7.6 Auxiliary Lemmas

lemma *s1* [*iff*]: ‹*new-term c t* ⟷ (*c* ∉ *paramst t*)› ‹*new-list c l* ⟷ (*c* ∉ *paramsts l*)›
by (*induct t* **and** *l rule*: *new-term.induct new-list.induct*) *simp-all*

lemma *s2* [*iff*]: ‹*new c p* ⟷ (*c* ∉ *params p*)›
by (*induct p*) *simp-all*

lemma *s3* [*iff*]: ‹*news c z* ⟷ *list-all* (λ*p*. *c* ∉ *params p*) *z*›
by (*induct z*) *simp-all*

lemma *s4* [*simp*]: ‹*inc-term t* = *liftt t*› ‹*inc-list l* = *liftts l*›
by (*induct t* **and** *l rule*: *inc-term.induct inc-list.induct*) *simp-all*

lemma *s5* [*simp*]: ‹*sub-term v s t* = *substt t s v*› ‹*sub-list v s l* = *substts l s v*›
by (*induct t* **and** *l rule*: *inc-term.induct inc-list.induct*) *simp-all*

lemma *s6* [*simp*]: ‹*sub v s p* = *subst p s v*›
by (*induct p arbitrary*: *v s*) *simp-all*

1.7.7 Soundness

theorem *sound*: ‹⊩ *z* ⟹ ∃ *p* ∈ *set z*. *semantics e f g p*›
proof (*induct arbitrary*: *f rule*: *sequent-calculus.induct*)
case (*10 i p z*)
then show *?case*
proof (*cases* ‹∀ *x*. *semantics e* (*f*(*i* := λ-. *x*)) *g* (*sub 0* (*Fun i* []) *p*)›)
case *False*
moreover have ‹*list-all* (λ*p*. *i* ∉ *params p*) *z*›
using *10* **by** *simp*

ultimately show *?thesis*
using *10 Ball-set insert-iff list.set*(*2*) *upd-lemma* **by** *metis*
qed *simp*
next
case (*11 i p z*)
then show *?case*
proof (*cases* ‹$\forall x$. *semantics e* ($f(i := \lambda$-. x)) *g* (*Neg* (*sub 0* (*Fun i* []) *p*))›)
case *False*
moreover have ‹*list-all* (λp. $i \notin$ *params p*) *z*›
using *11* **by** *simp*
ultimately show *?thesis*
using *11 Ball-set insert-iff list.set*(*2*) *upd-lemma* **by** *metis*
qed *simp*
qed *force*+

corollary ‹⊩ $z \Longrightarrow \exists p$. *member p z* $\wedge$ *semantics e f g p*›
using *sound* **by** *force*

corollary ‹⊩ [*p*] $\Longrightarrow$ *semantics e f g p*›
using *sound* **by** *force*

corollary ‹¬ (⊩ [])›
using *sound* **by** *force*

1.8 Reference

Mordechai Ben-Ari (Springer 2012): Mathematical Logic for Computer Science (Third Edition)

end
theory *Sequent1* **imports** *FOL-Seq-Calc1.Sequent*
begin

This theory exists exclusively as a shim to link the AFP theory imported here to the *Sequent-Calculus-Verifier* theory.

end

1.9 Appendix: Completeness

theory *Sequent-Calculus-Verifier* **imports** *Sequent1 SeCaV* **begin**

primrec *from-tm* **and** *from-tm-list* **where**
‹*from-tm* (*Var n*) = *FOL-Fitting.Var n*› |
‹*from-tm* (*Fun a ts*) = *App a* (*from-tm-list ts*)› |
‹*from-tm-list* [] = []› |
‹*from-tm-list* (*t* # *ts*) = *from-tm t* # *from-tm-list ts*›

primrec *from-fm* **where**

‹*from-fm* (*Pre b ts*) = *Pred b* (*from-tm-list ts*)› |
‹*from-fm* (*Con p q*) = *And* (*from-fm p*) (*from-fm q*)› |
‹*from-fm* (*Dis p q*) = *Or* (*from-fm p*) (*from-fm q*)› |
‹*from-fm* (*Imp p q*) = *Impl* (*from-fm p*) (*from-fm q*)› |
‹*from-fm* (*Neg p*) = *FOL-Fitting.Neg* (*from-fm p*)› |
‹*from-fm* (*Uni p*) = *Forall* (*from-fm p*)› |
‹*from-fm* (*Exi p*) = *Exists* (*from-fm p*)›

primrec *to-tm* **and** *to-tm-list* **where**
‹*to-tm* (*FOL-Fitting.Var n*) = *Var n*› |
‹*to-tm* (*App a ts*) = *Fun a* (*to-tm-list ts*)› |
‹*to-tm-list* [] = []› |
‹*to-tm-list* (*t* # *ts*) = *to-tm t* # *to-tm-list ts*›

primrec *to-fm* **where**
‹*to-fm* ⊥ = *Neg* (*Imp* (*Pre 0* []) (*Pre 0* []))› |
‹*to-fm* ⊤ = *Imp* (*Pre 0* []) (*Pre 0* [])› |
‹*to-fm* (*Pred b ts*) = *Pre b* (*to-tm-list ts*)› |
‹*to-fm* (*And p q*) = *Con* (*to-fm p*) (*to-fm q*)› |
‹*to-fm* (*Or p q*) = *Dis* (*to-fm p*) (*to-fm q*)› |
‹*to-fm* (*Impl p q*) = *Imp* (*to-fm p*) (*to-fm q*)› |
‹*to-fm* (*FOL-Fitting.Neg p*) = *Neg* (*to-fm p*)› |
‹*to-fm* (*Forall p*) = *Uni* (*to-fm p*)› |
‹*to-fm* (*Exists p*) = *Exi* (*to-fm p*)›

theorem *to-from-tm* [*simp*]: ‹*to-tm* (*from-tm t*) = *t*› ‹*to-tm-list* (*from-tm-list ts*) = *ts*›
by (*induct t* **and** *ts rule*: *from-tm.induct from-tm-list.induct*) *simp-all*

theorem *to-from-fm* [*simp*]: ‹*to-fm* (*from-fm p*) = *p*›
by (*induct p*) *simp-all*

lemma *Truth* [*simp*]: ‹⊩ *Imp* (*Pre 0* []) (*Pre 0* []) # *z*›
using *AlphaImp Basic Ext ext.simps member.simps*(*2*) **by** *metis*

lemma *paramst* [*simp*]:
‹*FOL-Fitting.new-term c t* = *new-term c* (*to-tm t*)›
‹*FOL-Fitting.new-list c l* = *new-list c* (*to-tm-list l*)›
by (*induct t* **and** *l rule*: *FOL-Fitting.paramst.induct FOL-Fitting.paramsts.induct*) *simp-all*

lemma *params* [*iff*]: ‹*FOL-Fitting.new c p* ⟷ *new c* (*to-fm p*)›
by (*induct p*) *simp-all*

lemma *list-params* [*iff*]: ‹*FOL-Fitting.news c z* ⟷ *news c* (*map to-fm z*)›
by (*induct z*) *simp-all*

lemma *liftt* [*simp*]:
‹*to-tm* (*FOL-Fitting.liftt t*) = *inc-term* (*to-tm t*)›

‹*to-tm-list* (*FOL-Fitting.liftts l*) = *inc-list* (*to-tm-list l*)›
by (*induct t* **and** *l rule*: *FOL-Fitting.liftt.induct FOL-Fitting.liftts.induct*) *simp-all*

lemma *substt* [*simp*]:
‹*to-tm* (*FOL-Fitting.substt t s v*) = *sub-term v* (*to-tm s*) (*to-tm t*)›
‹*to-tm-list* (*FOL-Fitting.substts l s v*) = *sub-list v* (*to-tm s*) (*to-tm-list l*)›
by (*induct t* **and** *l rule*: *FOL-Fitting.substt.induct FOL-Fitting.substts.induct*) *simp-all*

lemma *subst* [*simp*]: ‹*to-fm* (*FOL-Fitting.subst A t s*) = *sub s* (*to-tm t*) (*to-fm A*)›
by (*induct A arbitrary*: *t s*) *simp-all*

lemma *sim*: ‹($\vdash$ *x*) $\Longrightarrow$ ($\Vdash$ (*map to-fm x*))›
by (*induct rule*: *SC.induct*) (*force intro*: *sequent-calculus.intros*)+

lemma *evalt* [*simp*]:
‹*semantics-term e f t* = *evalt e f* (*from-tm t*)›
‹*semantics-list e f ts* = *evalts e f* (*from-tm-list ts*)›
by (*induct t* **and** *ts rule*: *from-tm.induct from-tm-list.induct*) *simp-all*

lemma *shift* [*simp*]: ‹*shift e 0 x* = *e*⟨*0*:*x*⟩›
unfolding *shift-def FOL-Fitting.shift-def* **by** *simp*

lemma *semantics* [*iff*]: ‹*semantics e f g p* $\longleftrightarrow$ *eval e f g* (*from-fm p*)›
by (*induct p arbitrary*: *e*) *simp-all*

abbreviation *valid* (‹$\gg$ -› *0*) **where**
‹($\gg$ *p*) $\equiv$ $\forall$(*e* :: - $\Rightarrow$ *nat hterm*) *f g*. *semantics e f g p*›

theorem *complete-sound*: ‹$\gg$ *p* $\Longrightarrow$ $\Vdash$ [*p*]› ‹$\Vdash$ [*q*] $\Longrightarrow$ *semantics e f g q*›
by (*metis to-from-fm sim semantics list.map SC-completeness*) (*use sound* **in** *force*)

corollary ‹($\gg$ *p*) $\longleftrightarrow$ ($\Vdash$ [*p*])›
using *complete-sound* **by** *fast*

1.10 Reference

Asta Halkjær From (2019): Sequent Calculus https://www.isa-afp.org/entries/FOL_Seq_Calc1.html

end

Chapter 2

The prover

2.1 Proof search procedure

theory *Prover*
  imports *SeCaV*
    HOL−Library.Stream
    Abstract-Completeness.Abstract-Completeness
    Abstract-Soundness.Finite-Proof-Soundness
    HOL−Library.Countable
    HOL−Library.Code-Lazy
begin

This theory defines the actual proof search procedure.

2.1.1 Datatypes

A sequent is a list of formulas

type-synonym *sequent* = ‹*fm list*›

We introduce a number of rules to prove sequents. These rules mirror the proof system of SeCaV, but are higher-level in the sense that they apply to all formulas in the sequent at once. This obviates the need for the structural Ext rule. There is also no Basic rule, since this is implicit in the prover.

datatype *rule*
  = *AlphaDis* | *AlphaImp* | *AlphaCon*
  | *BetaCon* | *BetaImp* | *BetaDis*
  | *DeltaUni* | *DeltaExi*
  | *NegNeg*
  | *GammaExi* | *GammaUni*

2.1.2 Auxiliary functions

Before defining what the rules do, we need to define a number of auxiliary functions needed for the semantics of the rules.

listFunTm is a list of function and constant names in a term

```
primrec listFunTm :: ‹tm ⇒ nat list› and listFunTms :: ‹tm list ⇒ nat list›where
  ‹listFunTm (Fun n ts) = n # listFunTms ts›
| ‹listFunTm (Var n) = []›
| ‹listFunTms [] = []›
| ‹listFunTms (t # ts) = listFunTm t @ listFunTms ts›
```

generateNew uses the *listFunTms* function to obtain a fresh function index

```
definition generateNew :: ‹tm list ⇒ nat› where
  ‹generateNew ts ≡ 1 + foldr max (listFunTms ts) 0›
```

subtermTm returns a list of all terms occurring within a term

```
primrec subtermTm :: ‹tm ⇒ tm list› where
  ‹subtermTm (Fun n ts) = Fun n ts # remdups (concat (map subtermTm ts))›
| ‹subtermTm (Var n) = [Var n]›
```

subtermFm returns a list of all terms occurring within a formula

```
primrec subtermFm :: ‹fm ⇒ tm list› where
  ‹subtermFm (Pre - ts) = concat (map subtermTm ts)›
| ‹subtermFm (Imp p q) = subtermFm p @ subtermFm q›
| ‹subtermFm (Dis p q) = subtermFm p @ subtermFm q›
| ‹subtermFm (Con p q) = subtermFm p @ subtermFm q›
| ‹subtermFm (Exi p) = subtermFm p›
| ‹subtermFm (Uni p) = subtermFm p›
| ‹subtermFm (Neg p) = subtermFm p›
```

subtermFms returns a list of all terms occurring within a list of formulas

```
abbreviation ‹subtermFms z ≡ concat (map subtermFm z)›
```

subterms returns a list of all terms occurring within a sequent. This is used to determine which terms to instantiate Gamma-formulas with. We must always be able to instantiate Gamma-formulas, so if there are no terms in the sequent, the function simply returns a list containing the first function.

```
definition subterms :: ‹sequent ⇒ tm list› where
  ‹subterms z ≡ case remdups (subtermFms z) of
                [] ⇒ [Fun 0 []]
              | ts ⇒ ts›
```

We need to be able to detect if a sequent is an axiom to know whether a branch of the proof is done. The disjunct *Neg* (*Neg p*) ∈ *set z* is not necessary for the prover, but makes the proof of the lemma *branchDone-contradiction* easier.

```
fun branchDone :: ‹sequent ⇒ bool› where
  ‹branchDone [] = False›
| ‹branchDone (Neg p # z) = (p ∈ set z ∨ Neg (Neg p) ∈ set z ∨ branchDone z)›
| ‹branchDone (p # z) = (Neg p ∈ set z ∨ branchDone z)›
```

2.1.3 Effects of rules

This defines the resulting formulas when applying a rule to a single formula. This definition mirrors the semantics of SeCaV. If the rule and the formula do not match, the resulting formula is simply the original formula. Parameter A should be the list of terms on the branch.

```
definition parts :: ‹tm list ⇒ rule ⇒ fm ⇒ fm list list› where
  ‹parts A r f = (case (r, f) of
      (NegNeg, Neg (Neg p)) ⇒ [[p]]
    | (AlphaImp, Imp p q) ⇒ [[Neg p, q]]
    | (AlphaDis, Dis p q) ⇒ [[p, q]]
    | (AlphaCon, Neg (Con p q)) ⇒ [[Neg p, Neg q]]
    | (BetaImp, Neg (Imp p q)) ⇒ [[p], [Neg q]]
    | (BetaDis, Neg (Dis p q)) ⇒ [[Neg p], [Neg q]]
    | (BetaCon, Con p q) ⇒ [[p], [q]]
    | (DeltaExi, Neg (Exi p)) ⇒ [[Neg (sub 0 (Fun (generateNew A) []) p)]]
    | (DeltaUni, Uni p) ⇒ [[sub 0 (Fun (generateNew A) []) p]]
    | (GammaExi, Exi p) ⇒ [Exi p # map (λt. sub 0 t p) A]
    | (GammaUni, Neg (Uni p)) ⇒ [Neg (Uni p) # map (λt. Neg (sub 0 t p)) A]
    | - ⇒ [[f]])›
```

This function defines the Cartesian product of two lists. This is needed to create the list of branches created when applying a beta rule.

```
primrec list-prod :: ‹'a list list ⇒ 'a list list ⇒ 'a list list› where
  ‹list-prod - [] = []›
| ‹list-prod hs (t # ts) = map (λh. h @ t) hs @ list-prod hs ts›
```

This function computes the children of a node in the proof tree. For Alpha rules, Delta rules and Gamma rules, there will be only one sequent, which is the result of applying the rule to every formula in the current sequent. For Beta rules, the proof tree will branch into two branches once for each formula in the sequent that matches the rule, which results in 2^n branches (created using *list-prod*). The list of terms in the sequent needs to be updated after applying the rule to each formula since Delta rules and Gamma rules may introduce new terms. Note that any formulas that don't match the rule are left unchanged in the new sequent.

```
primrec children :: ‹tm list ⇒ rule ⇒ sequent ⇒ sequent list› where
  ‹children - - [] = [[]]›
| ‹children A r (p # z) =
  (let hs = parts A r p; A' = remdups (A @ subtermFms (concat hs))
   in list-prod hs (children A' r z))›
```

The proof state is the combination of a list of terms and a sequent.

```
type-synonym state = ‹tm list × sequent›
```

This function defines the effect of applying a rule to a proof state. If the sequent is an axiom, the effect is to end the branch of the proof tree, so an empty set of child branches is returned. Otherwise, we compute the children generated by applying the rule to the current proof state, then add any new subterms to the proof states of the children.

primrec *effect* :: ‹*rule* ⇒ *state* ⇒ *state fset*› **where**
 ‹*effect r* (*A*, *z*) =
 (*if branchDone z then* {||} *else*
  fimage ($\lambda z'$. (*remdups* (*A* @ *subterms z* @ *subterms* z'), z'))
  (*fset-of-list* (*children* (*remdups* (*A* @ *subtermFms z*)) *r z*)))›

2.1.4 The rule stream

We need to define an infinite stream of rules that the prover should try to apply. Since rules simply do nothing if they don't fit the formulas in the sequent, the rule stream is just all rules in the order: Alpha, Delta, Beta, Gamma, which guarantees completeness.

definition ‹*rulesList* ≡ [
 NegNeg, *AlphaImp*, *AlphaDis*, *AlphaCon*,
 DeltaExi, *DeltaUni*,
 BetaImp, *BetaDis*, *BetaCon*,
 GammaExi, *GammaUni*
]›

By cycling the list of all rules we obtain an infinite stream with every rule occurring infinitely often.

definition *rules* **where**
 ‹*rules* = *cycle rulesList*›

2.1.5 Abstract completeness

We write effect as a relation to use it with the abstract completeness framework.

definition *eff* **where**
 ‹*eff* ≡ λ*r s ss*. *effect r s* = *ss*›

To use the framework, we need to prove enabledness. This is trivial because all of our rules are always enabled and simply do nothing if they don't match the formulas.

lemma *all-rules-enabled*: ‹∀ *st*. ∀ *r* ∈ *i.R* (*cycle rulesList*). ∃ *sl*. *eff r st sl*›
 unfolding *eff-def* **by** *blast*

The first step of the framework is to prove that our prover fits the framework.

interpretation *RuleSystem eff rules UNIV*
 unfolding *rules-def RuleSystem-def*
 using *all-rules-enabled stream.set-sel*(*1*)
 by *blast*

Next, we need to prove that our rules are persistent. This is also trivial, since all of our rules are always enabled.

lemma *all-rules-persistent*: ‹$\forall\, r.\ r \in R \longrightarrow$ *per* r›
  by (*metis all-rules-enabled enabled-def per-def rules-def*)

We can then prove that our prover fully fits the framework.

interpretation *PersistentRuleSystem eff rules UNIV*
 unfolding *PersistentRuleSystem-def RuleSystem-def PersistentRuleSystem-axioms-def*
  using *all-rules-persistent enabled-R*
  by *blast*

We can then use the framework to define the prover. The mkTree function applies the rules to build the proof tree using the effect relation, but the prover is not actually executable yet.

definition ‹*secavProver* $\equiv$ *mkTree rules*›

abbreviation ‹*rootSequent* $t \equiv$ *snd* (*fst* (*root* t))›

end

2.2 Export

theory *Export*
  imports *Prover*
begin

In this theory, we make the prover executable using the code interpretation of the abstract completeness framework and the Isabelle to Haskell code generator.

To actually execute the prover, we need to lazily evaluate the stream of rules to apply. Otherwise, we will never actually get to a result.

code-lazy-type *stream*

We would also like to make the evaluation of streams a bit more efficient.

declare *Stream.smember-code* [*code del*]
lemma [*code*]: *Stream.smember* x (y ## s) = ($x = y \vee$ *Stream.smember* x s)
  unfolding *Stream.smember-def* **by** *auto*

To export code to Haskell, we need to specify that functions on the option type should be exported into the equivalent functions on the Maybe monad.

code-printing
  constant *the* ⇀ (*Haskell*) *MaybeExt.fromJust*
| **constant** *Option.is-none* ⇀ (*Haskell*) *MaybeExt.isNothing*

To use the Maybe monad, we need to import it, so we add a shim to do so in every module.

code-printing code-module *MaybeExt* ⇀ (*Haskell*)
‹*module MaybeExt*(*fromJust*, *isNothing*) *where*
import Data.Maybe(*fromJust*, *isNothing*);›

The default export setup will create a cycle of module imports, so we roll most of the theories into one module when exporting to Haskell to prevent this.

code-identifier
code-module *Stream* ⇀ (*Haskell*) *Prover*
| **code-module** *Prover* ⇀ (*Haskell*) *Prover*
| **code-module** *Export* ⇀ (*Haskell*) *Prover*
| **code-module** *Option* ⇀ (*Haskell*) *Prover*
| **code-module** *MaybeExt* ⇀ (*Haskell*) *Prover*
| **code-module** *Abstract-Completeness* ⇀ (*Haskell*) *Prover*

Finally, we define an executable version of the prover using the code interpretation from the framework, and a version where the list of terms is initially empty.

definition ‹*secavTreeCode* ≡ *i.mkTree* (λ*r s*. *Some* (*effect r s*)) *rules*›
definition ‹*secavProverCode* ≡ λ*z* . *secavTreeCode* ([], *z*)›

We then export this version of the prover into Haskell.

export-code open *secavProverCode* **in** *Haskell*

end

2.3 Lemmas about the prover

theory *ProverLemmas* **imports** *Prover* **begin**

This theory contains a number of lemmas about the prover. We will need these when proving soundness and completeness.

2.3.1 SeCaV lemmas

We need a few lemmas about the SeCaV system.

Incrementing variable indices does not change the function names in term or a list of terms.

lemma *paramst-liftt* [*simp*]:
‹*paramst* (*liftt t*) = *paramst t*› ‹*paramsts* (*liftts ts*) = *paramsts ts*›
by (*induct t* **and** *ts rule*: *liftt.induct liftts.induct*) *auto*

Subterms do not contain any functions except those in the original term

lemma *paramst-sub-term*:
‹*paramst* (*sub-term m s t*) ⊆ *paramst s* ∪ *paramst t*›

‹*paramsts* (*sub-list m s l*) ⊆ *paramst s* ∪ *paramsts l*›
by (*induct t* **and** *l rule*: *sub-term.induct sub-list.induct*) *auto*

Substituting a variable for a term does not introduce function names not in that term

lemma *params-sub*: ‹*params* (*sub m t p*) ⊆ *paramst t* ∪ *params p*›
proof (*induct p arbitrary*: *m t*)
case (*Pre x1 x2*)
then show *?case*
using *paramst-sub-term*(*2*) **by** *simp*
qed *fastforce*+

abbreviation ‹*paramss z* ≡ ⋃ *p* ∈ *set z*. *params p*›

If a function name is fresh, it is not in the list of function names in the sequent

lemma *news-paramss*: ‹*news i z* ⟷ *i* ∉ *paramss z*›
by (*induct z*) *auto*

If a list of terms is a subset of another, the set of function names in it is too

lemma *paramsts-subset*: ‹*set A* ⊆ *set B* ⟹ *paramsts A* ⊆ *paramsts B*›
by (*induct A*) *auto*

Substituting a variable by a term does not change the depth of a formula (only the term size changes)

lemma *size-sub* [*simp*]: ‹*size* (*sub i t p*) = *size p*›
by (*induct p arbitrary*: *i t*) *auto*

2.3.2 Fairness

While fairness of the rule stream should be pretty trivial (since we are simply repeating a static list of rules forever), the proof is a bit involved.

This function tells us what rule comes next in the stream.

primrec *next-rule* :: ‹*rule* ⇒ *rule*› **where**
‹*next-rule NegNeg* = *AlphaImp*›
| ‹*next-rule AlphaImp* = *AlphaDis*›
| ‹*next-rule AlphaDis* = *AlphaCon*›
| ‹*next-rule AlphaCon* = *DeltaExi*›
| ‹*next-rule DeltaExi* = *DeltaUni*›
| ‹*next-rule DeltaUni* = *BetaImp*›
| ‹*next-rule BetaImp* = *BetaDis*›
| ‹*next-rule BetaDis* = *BetaCon*›
| ‹*next-rule BetaCon* = *GammaExi*›
| ‹*next-rule GammaExi* = *GammaUni*›
| ‹*next-rule GammaUni* = *NegNeg*›

This function tells us the index of a rule in the list of rules to repeat.

primrec *rule-index* :: ‹*rule* ⇒ *nat*› **where**
‹*rule-index NegNeg* = *0*›
| ‹*rule-index AlphaImp* = *1*›
| ‹*rule-index AlphaDis* = *2*›
| ‹*rule-index AlphaCon* = *3*›
| ‹*rule-index DeltaExi* = *4*›
| ‹*rule-index DeltaUni* = *5*›
| ‹*rule-index BetaImp* = *6*›
| ‹*rule-index BetaDis* = *7*›
| ‹*rule-index BetaCon* = *8*›
| ‹*rule-index GammaExi* = *9*›
| ‹*rule-index GammaUni* = *10*›

The list of rules does not have any duplicates. This is important because we can then look up rules by their index.

lemma *distinct-rulesList*: ‹*distinct rulesList*›
unfolding *rulesList-def* **by** *simp*

If you cycle a list, it repeats every *length* elements.

lemma *cycle-nth*: ‹*xs* ≠ [] ⟹ *cycle xs* !! *n* = *xs* ! (*n mod length xs*)›
by (*metis cycle.sel*(*1*) *hd-rotate-conv-nth rotate-conv-mod sdrop-cycle sdrop-simps*(*1*))

The rule index function can actually be used to look up rules in the list.

lemma *nth-rule-index*: ‹*rulesList* ! (*rule-index r*) = *r*›
unfolding *rulesList-def* **by** (*cases r*) *simp-all*

lemma *rule-index-bnd*: ‹*rule-index r* < *length rulesList*›
unfolding *rulesList-def* **by** (*cases r*) *simp-all*

lemma *unique-rule-index*:
assumes ‹*n* < *length rulesList*› ‹*rulesList* ! *n* = *r*›
shows ‹*n* = *rule-index r*›
using *assms nth-rule-index distinct-rulesList rule-index-bnd nth-eq-iff-index-eq*
by *metis*

The rule indices repeat in the stream each cycle.

lemma *rule-index-mod*:
assumes ‹*rules* !! *n* = *r*›
shows ‹*n mod length rulesList* = *rule-index r*›
proof −
have ‹*n mod length rulesList* < *length rulesList*›
by (*simp add*: *rulesList-def*)
moreover have ‹*rulesList* ! (*n mod length rulesList*) = *r*›
using *assms cycle-nth* **unfolding** *rules-def rulesList-def* **by** (*metis list.distinct*(*1*))
ultimately show *?thesis*
by (*rule unique-rule-index*)
qed

We need some lemmas about the modulo function to show that the rules repeat at the right rate.

lemma *mod-hit*:
  fixes *k* :: *nat*
  assumes ‹*0* < *k*›
  shows ‹∀ *i* < *k*. ∃ *n* > *m*. *n mod k* = *i*›
proof *safe*
  fix *i*
  let *?n* = ‹(*1* + *m*) ∗ *k* + *i*›
  assume ‹*i* < *k*›
  then have ‹*?n mod k* = *i*›
    by (*metis mod-less mod-mult-self3*)
  moreover have ‹*?n* > *m*›
    using *assms*
    by (*metis One-nat-def Suc-eq-plus1-left Suc-leI add.commute add-lessD1 less-add-one mult.right-neutral nat-mult-less-cancel1 order-le-less trans-less-add1 zero-less-one*)
  ultimately show ‹∃ *n* > *m*. *n mod k* = *i*›
    by *fast*
qed

lemma *mod-suff*:
  assumes ‹∀ (*n* :: *nat*) > *m*. *P* (*n mod k*)› ‹*0* < *k*›
  shows ‹∀ *i* < *k*. *P i*›
  using *assms mod-hit* **by** *blast*

It is always possible to find an index after some point that results in any given rule.

lemma *rules-repeat*: ‹∃ *n* > *m*. *rules* !! *n* = *r*›
proof (*rule ccontr*)
  assume ‹¬ (∃ *n* > *m*. *rules* !! *n* = *r*)›
  then have ‹¬ (∃ *n* > *m*. *n mod length rulesList* = *rule-index r*)›
    using *rule-index-mod nth-rule-index* **by** *metis*
  then have ‹∀ *n* > *m*. *n mod length rulesList* ≠ *rule-index r*›
    by *blast*
  moreover have ‹*length rulesList* > *0*›
    unfolding *rulesList-def* **by** *simp*
  ultimately have ‹∀ *k* < *length rulesList*. *k* ≠ *rule-index r*›
    using *mod-suff*[**where** *P*=‹λ*a*. *a* ≠ *rule-index r*›] **by** *blast*
  then show *False*
    using *rule-index-bnd* **by** *blast*
qed

It is possible to find such an index no matter where in the stream we start.

lemma *rules-repeat-sdrop*: ‹∃ *n*. (*sdrop k rules*) !! *n* = *r*›
  using *rules-repeat* **by** (*metis less-imp-add-positive sdrop-snth*)

Using the lemma above, we prove that the stream of rules is fair by coinduction.

lemma *fair-rules*: ‹*fair rules*›
proof −
  { fix *r* **assume** ‹*r* ∈ *R*›
    then obtain *m* **where** *r*: ‹*r* = *rules* !! *m*› **unfolding** *sset-range* **by** *blast*
    { fix *n* :: *nat* **and** *rs* **let** *?rules* = ‹λ*n*. *sdrop n rules*›
      assume ‹*n* > *0*›
      then have ‹*alw* (*ev* (*holds* ((=) *r*))) (*rs* @− *?rules n*)›
      proof (*coinduction arbitrary*: *n rs*)
        case *alw*
        show *?case*
        proof (*rule exI*[*of* - ‹*rs* @− *?rules n*›], *safe*)
          show ‹∃ *n′ rs′*. *stl* (*rs* @− *?rules n*) = *rs′* @− *?rules n′* ∧ *n′* > *0*›
          proof (*cases rs*)
            case *Nil* **then show** *?thesis* **unfolding** *alw*
              by (*metis sdrop-simps*(*2*) *shift.simps*(*1*) *zero-less-Suc*)
          qed (*auto simp*: *alw intro*: *exI*[*of* - *n*])
        next
          have ‹*ev* (*holds* ((=) *r*)) (*sdrop n rules*)›
            unfolding *ev-holds-sset* **using** *rules-repeat-sdrop* **by** (*metis snth-sset*)
          then show ‹*ev* (*holds* ((=) *r*)) (*rs* @− *sdrop n rules*)›
            unfolding *ev-holds-sset* **by** *simp*
        qed
      qed
    }
  }
  then show ‹*fair rules*› **unfolding** *fair-def*
   by (*metis* (*full-types*) *alw-iff-sdrop ev-holds-sset neq0-conv order-refl sdrop.simps*(*1*) *stake-sdrop*)
qed

2.3.3 Substitution

We need some lemmas about substitution of variables for terms for the Delta and Gamma rules.

If a term is a subterm of another, so are all of its subterms.

lemma *subtermTm-le*: ‹*t* ∈ *set* (*subtermTm s*) ⟹ *set* (*subtermTm t*) ⊆ *set* (*subtermTm s*)›
  by (*induct s*) *auto*

Trying to substitute a variable that is not in the term does nothing (contrapositively).

lemma *sub-term-const-transfer*:
  ‹*sub-term m* (*Fun a* []) *t* ≠ *sub-term m s t* ⟹
    Fun a [] ∈ *set* (*subtermTm* (*sub-term m* (*Fun a* []) *t*))›
  ‹*sub-list m* (*Fun a* []) *ts* ≠ *sub-list m s ts* ⟹
    Fun a [] ∈ (⋃*t* ∈ *set* (*sub-list m* (*Fun a* []) *ts*). *set* (*subtermTm t*))›
proof (*induct t* **and** *ts rule*: *sub-term.induct sub-list.induct*)
  case (*Var x*)

then show *?case*
  by (*metis list.set-intros*(*1*) *sub-term.simps*(*1*) *subtermTm.simps*(*1*))
qed *auto*

If substituting different terms makes a difference, then the substitution has an effect.

lemma *sub-const-transfer*:
  assumes ‹*sub m* (*Fun a* []) *p* ≠ *sub m t p*›
  shows ‹*Fun a* [] ∈ *set* (*subtermFm* (*sub m* (*Fun a* []) *p*))›
  using *assms*
proof (*induct p arbitrary*: *m t*)
  case (*Pre i l*)
  then show *?case*
    using *sub-term-const-transfer*(*2*) **by** *simp*
qed *auto*

If the list of subterms is empty for all formulas in a sequent, constant 0 is used instead.

lemma *set-subterms*:
  fixes *z*
  defines ‹*ts* ≡ ⋃ *p* ∈ *set z*. *set* (*subtermFm p*)›
  shows ‹*set* (*subterms z*) = (*if ts* = {} *then* {*Fun 0* []} *else ts*)›
proof −
  have ∗: ‹*set* (*remdups* (*concat* (*map subtermFm z*))) = (⋃ *p* ∈ *set z*. *set* (*subtermFm p*))›
    by (*induct z*) *auto*
  then show *?thesis*
  proof (*cases* ‹*ts* = {}›)
    case *True*
    then show *?thesis*
      unfolding *subterms-def ts-def* **using** ∗
      by (*metis list.simps*(*15*) *list.simps*(*4*) *set-empty*)
  next
    case *False*
    then show *?thesis*
      unfolding *subterms-def ts-def* **using** ∗
      by (*metis empty-set list.exhaust list.simps*(*5*))
  qed
qed

The parameters and the subterm functions respect each other.

lemma *paramst-subtermTm*:
  ‹∀ *i* ∈ *paramst t*. ∃ *l*. *Fun i l* ∈ *set* (*subtermTm t*)›
  ‹∀ *i* ∈ *paramsts ts*. ∃ *l*. *Fun i l* ∈ (⋃ *t* ∈ *set ts*. *set* (*subtermTm t*))›
  by (*induct t* **and** *ts rule*: *paramst.induct paramsts.induct*) *fastforce*+

lemma *params-subtermFm*: ‹∀ *i* ∈ *params p*. ∃ *l*. *Fun i l* ∈ *set* (*subtermFm p*)›
proof (*induct p*)

case (*Pre x1 x2*)
then show *?case*
using *paramst-subtermTm* **by** *simp*
qed *auto*

lemma *subtermFm-subset-params*: ‹*set* (*subtermFm p*) $\subseteq$ *set A* $\Longrightarrow$ *params p* $\subseteq$ *paramsts A*›
using *params-subtermFm* **by** *force*

2.3.4 Custom cases

Some proofs are more efficient with some custom case lemmas.

lemma *Neg-exhaust*
[*case-names Pre Imp Dis Con Exi Uni NegPre NegImp NegDis NegCon NegExi NegUni NegNeg*]:
assumes
‹$\bigwedge$*i ts*. *x* = *Pre i ts* $\Longrightarrow$ *P*›
‹$\bigwedge$*p q*. *x* = *Imp p q* $\Longrightarrow$ *P*›
‹$\bigwedge$*p q*. *x* = *Dis p q* $\Longrightarrow$ *P*›
‹$\bigwedge$*p q*. *x* = *Con p q* $\Longrightarrow$ *P*›
‹$\bigwedge$*p*. *x* = *Exi p* $\Longrightarrow$ *P*›
‹$\bigwedge$*p*. *x* = *Uni p* $\Longrightarrow$ *P*›
‹$\bigwedge$*i ts*. *x* = *Neg* (*Pre i ts*) $\Longrightarrow$ *P*›
‹$\bigwedge$*p q*. *x* = *Neg* (*Imp p q*) $\Longrightarrow$ *P*›
‹$\bigwedge$*p q*. *x* = *Neg* (*Dis p q*) $\Longrightarrow$ *P*›
‹$\bigwedge$*p q*. *x* = *Neg* (*Con p q*) $\Longrightarrow$ *P*›
‹$\bigwedge$*p*. *x* = *Neg* (*Exi p*) $\Longrightarrow$ *P*›
‹$\bigwedge$*p*. *x* = *Neg* (*Uni p*) $\Longrightarrow$ *P*›
‹$\bigwedge$*p*. *x* = *Neg* (*Neg p*) $\Longrightarrow$ *P*›
shows *P*
using *assms*
proof (*induct x*)
case (*Neg p*)
then show *?case*
by (*cases p*) *simp-all*
qed *simp-all*

lemma *parts-exhaust*
[*case-names AlphaDis AlphaImp AlphaCon BetaDis BetaImp BetaCon DeltaUni DeltaExi NegNeg GammaExi GammaUni Other*]:
assumes
‹$\bigwedge$*p q*. *r* = *AlphaDis* $\Longrightarrow$ *x* = *Dis p q* $\Longrightarrow$ *P*›
‹$\bigwedge$*p q*. *r* = *AlphaImp* $\Longrightarrow$ *x* = *Imp p q* $\Longrightarrow$ *P*›
‹$\bigwedge$*p q*. *r* = *AlphaCon* $\Longrightarrow$ *x* = *Neg* (*Con p q*) $\Longrightarrow$ *P*›
‹$\bigwedge$*p q*. *r* = *BetaDis* $\Longrightarrow$ *x* = *Neg* (*Dis p q*) $\Longrightarrow$ *P*›
‹$\bigwedge$*p q*. *r* = *BetaImp* $\Longrightarrow$ *x* = *Neg* (*Imp p q*) $\Longrightarrow$ *P*›
‹$\bigwedge$*p q*. *r* = *BetaCon* $\Longrightarrow$ *x* = *Con p q* $\Longrightarrow$ *P*›
‹$\bigwedge$*p*. *r* = *DeltaUni* $\Longrightarrow$ *x* = *Uni p* $\Longrightarrow$ *P*›
‹$\bigwedge$*p*. *r* = *DeltaExi* $\Longrightarrow$ *x* = *Neg* (*Exi p*) $\Longrightarrow$ *P*›

```
    ‹⋀p. r = NegNeg ⟹ x = Neg (Neg p) ⟹ P›
    ‹⋀p. r = GammaExi ⟹ x = Exi p ⟹ P›
    ‹⋀p. r = GammaUni ⟹ x = Neg (Uni p) ⟹ P›
    ‹∀ A. parts A r x = [[x]] ⟹ P›
  shows P
  using assms
proof (cases r)
  case BetaCon
  then show ?thesis
    using assms
  proof (cases x rule: Neg-exhaust)
    case (Con p q)
    then show ?thesis
      using BetaCon assms by blast
  qed (simp-all add: parts-def)
next
  case BetaImp
  then show ?thesis
    using assms
  proof (cases x rule: Neg-exhaust)
    case (NegImp p q)
    then show ?thesis
      using BetaImp assms by blast
  qed (simp-all add: parts-def)
next
  case DeltaUni
  then show ?thesis
    using assms
  proof (cases x rule: Neg-exhaust)
    case (Uni p)
    then show ?thesis
      using DeltaUni assms by fast
  qed (simp-all add: parts-def)
next
  case DeltaExi
  then show ?thesis
    using assms
  proof (cases x rule: Neg-exhaust)
    case (NegExi p)
    then show ?thesis
      using DeltaExi assms by fast
  qed (simp-all add: parts-def)
next
  case n: NegNeg
  then show ?thesis
    using assms
  proof (cases x rule: Neg-exhaust)
    case (NegNeg p)
    then show ?thesis
```

using *n assms* **by** *fast*
qed (*simp-all add*: *parts-def*)
next
case *GammaExi*
then show *?thesis*
using *assms*
proof (*cases x rule*: *Neg-exhaust*)
case (*Exi p*)
then show *?thesis*
using *GammaExi assms* **by** *fast*
qed (*simp-all add*: *parts-def*)
next
case *GammaUni*
then show *?thesis*
using *assms*
proof (*cases x rule*: *Neg-exhaust*)
case (*NegUni p*)
then show *?thesis*
using *GammaUni assms* **by** *fast*
qed (*simp-all add*: *parts-def*)
qed (*cases x rule*: *Neg-exhaust*, *simp-all add*: *parts-def*)+

2.3.5 Unaffected formulas

We need some lemmas to show that formulas to which rules do not apply are not lost.

This function returns True if the rule applies to the formula, and False otherwise.

definition *affects* :: ‹*rule* ⇒ *fm* ⇒ *bool*› **where**
‹*affects r p* ≡ *case* (*r*, *p*) *of*
(*AlphaDis*, *Dis* - -) ⇒ *True*
| (*AlphaImp*, *Imp* - -) ⇒ *True*
| (*AlphaCon*, *Neg* (*Con* - -)) ⇒ *True*
| (*BetaCon*, *Con* - -) ⇒ *True*
| (*BetaImp*, *Neg* (*Imp* - -)) ⇒ *True*
| (*BetaDis*, *Neg* (*Dis* - -)) ⇒ *True*
| (*DeltaUni*, *Uni* -) ⇒ *True*
| (*DeltaExi*, *Neg* (*Exi* -)) ⇒ *True*
| (*NegNeg*, *Neg* (*Neg* -)) ⇒ *True*
| (*GammaExi*, *Exi* -) ⇒ *False*
| (*GammaUni*, *Neg* (*Uni* -)) ⇒ *False*
| (-, -) ⇒ *False*›

If a rule does not affect a formula, that formula will be in the sequent obtained after applying the rule.

lemma *parts-preserves-unaffected*:
assumes ‹¬ *affects r p*› ‹$z' \in$ *set* (*parts A r p*)›

shows ‹$p \in$ *set* z'›
using *assms* **unfolding** *affects-def*
by (*cases r p rule*: *parts-exhaust*) (*simp-all add*: *parts-def*)

The *list-prod* function computes the Cartesian product.

lemma *list-prod-is-cartesian*:
‹*set* (*list-prod hs ts*) $=$ {h @ t |h t. $h \in$ *set hs* $\wedge$ $t \in$ *set ts*}›
by (*induct ts*) *auto*

The *children* function produces the Cartesian product of the branches from the first formula and the branches from the rest of the sequent.

lemma *set-children-Cons*:
‹*set* (*children A r* (p # z)) $=$
{*hs* @ *ts* |*hs ts*. *hs* $\in$ *set* (*parts A r p*) $\wedge$
ts $\in$ *set* (*children* (*remdups* (*A* @ *subtermFms* (*concat* (*parts A r p*)))) *r z*)}›
using *list-prod-is-cartesian* **by** (*metis children.simps*(*2*))

The *children* function does not change unaffected formulas.

lemma *children-preserves-unaffected*:
assumes ‹$p \in$ *set* z› ‹$\neg$ *affects r p*› ‹$z' \in$ *set* (*children A r z*)›
shows ‹$p \in$ *set* z'›
using *assms parts-preserves-unaffected set-children-Cons*
by (*induct z arbitrary*: *A* z') *auto*

The *effect* function does not change unaffected formulas.

lemma *effect-preserves-unaffected*:
assumes ‹$p \in$ *set* z› **and** ‹$\neg$ *affects r p*› **and** ‹(*B*, z') |$\in$| *effect r* (*A*, *z*)›
shows ‹$p \in$ *set* z'›
using *assms children-preserves-unaffected*
unfolding *effect-def*
by (*smt* (*verit*, *best*) *Pair-inject femptyE fimageE fset-of-list-elem old.prod.case*)

2.3.6 Affected formulas

We need some lemmas to show that formulas to which rules do apply are decomposed into their constituent parts correctly.

If a formula occurs in a sequent on a child branch generated by *children*, it was part of the current sequent.

lemma *parts-in-children*:
assumes ‹$p \in$ *set* z› ‹$z' \in$ *set* (*children A r z*)›
shows ‹$\exists$ *B xs*. *set A* $\subseteq$ *set B* $\wedge$ *xs* $\in$ *set* (*parts B r p*) $\wedge$ *set xs* $\subseteq$ *set* z'›
using *assms*
proof (*induct z arbitrary*: *A* z')
case (*Cons a* -)
then show *?case*
proof (*cases* ‹$a = p$›)

case *True*
then show *?thesis*
using *Cons*(*3*) *set-children-Cons* **by** *fastforce*
next
case *False*
then show *?thesis*
using *Cons set-children-Cons*
by (*smt* (*verit*, *del-insts*) *le-sup-iff mem-Collect-eq set-ConsD set-append set-remdups subset-trans sup-ge2*)
qed
qed *simp*

If *effect* contains something, then the input sequent is not an axiom.

lemma *ne-effect-not-branchDone*: ‹$(B, z') \mid\in\mid$ *effect r* $(A, z) \Longrightarrow \neg$ *branchDone z*›
by (*cases* ‹*branchDone z*›) *simp-all*

The *effect* function decomposes formulas in the sequent using the *parts* function. (Unless the sequent is an axiom, in which case no child branches are generated.)

lemma *parts-in-effect*:
assumes ‹$p \in$ *set z*› **and** ‹$(B, z') \mid\in\mid$ *effect r* (A, z)›
shows ‹$\exists$ *C xs. set A* $\subseteq$ *set C* $\wedge$ *xs* $\in$ *set* (*parts C r p*) $\wedge$ *set xs* $\subseteq$ *set* z'›
using *assms parts-in-children ne-effect-not-branchDone*
by (*smt* (*verit*, *ccfv-threshold*) *Pair-inject effect.simps fimageE fset-of-list-elem le-sup-iff*
set-append set-remdups)

Specifically, this applied to the double negation elimination rule and the GammaUni rule.

corollary ‹*Neg* (*Neg p*) $\in$ *set z* $\Longrightarrow (B, z') \mid\in\mid$ *effect NegNeg* $(A, z) \Longrightarrow p \in$ *set* z'›
using *parts-in-effect* **unfolding** *parts-def* **by** *fastforce*

corollary ‹*Neg* (*Uni p*) $\in$ *set z* $\Longrightarrow (B, z') \mid\in\mid$ *effect GammaUni* $(A, z) \Longrightarrow$
set (*map* (λt. *Neg* (*sub 0 t p*)) *A*) $\subseteq$ *set* z'›
using *parts-in-effect* **unfolding** *parts-def* **by** *fastforce*

If the sequent is not an axiom, and the rule and sequent match, all of the child branches generated by *children* will be included in the proof tree.

lemma *eff-children*:
assumes ‹$\neg$ *branchDone z*› ‹*eff r* (A, z) *ss*›
shows ‹$\forall z' \in$ *set* (*children* (*remdups* (*A @ subtermFms z*)) *r z*). $\exists B$. $(B, z') \mid\in\mid$ *ss*›
using *assms* **unfolding** *eff-def* **using** *fset-of-list-elem* **by** *fastforce*

2.3.7 Generating new function names

We need to show that the *generateNew* function actually generates new function names. This requires a few lemmas about the interplay between *max* and *foldr*.

lemma *foldr-max*:
  fixes *xs* :: ‹*nat list*›
  shows ‹*foldr max xs 0* = (*if xs* = [] *then 0 else Max* (*set xs*))›
  by (*induct xs*) *simp-all*

lemma *Suc-max-new*:
  fixes *xs* :: ‹*nat list*›
  shows ‹*Suc* (*foldr max xs 0*) ∉ *set xs*›
proof (*cases xs*)
  case (*Cons x xs*)
  then have ‹*foldr max* (*x* # *xs*) *0* = *Max* (*set* (*x* # *xs*))›
    using *foldr-max* **by** *simp*
  then show *?thesis*
    using *Cons* **by** (*metis List.finite-set Max.insert add-0 empty-iff list.set*(*2*) *max-0-1*(*2*)
      n-not-Suc-n nat-add-max-left plus-1-eq-Suc remdups.simps(*2*) *set-remdups*)
qed *simp*

lemma *listFunTm-paramst*: ‹*set* (*listFunTm t*) = *paramst t*› ‹*set* (*listFunTms ts*) = *paramsts ts*›
  by (*induct t* **and** *ts rule*: *paramst.induct paramsts.induct*) *auto*

2.3.8 Finding axioms

The *branchDone* function correctly determines whether a sequent is an axiom.

lemma *branchDone-contradiction*: ‹*branchDone z* ⟷ (∃ *p*. *p* ∈ *set z* ∧ *Neg p* ∈ *set z*)›
  by (*induct z rule*: *branchDone.induct*) *auto*

2.3.9 Subterms

We need a few lemmas about the behaviour of our subterm functions.

Any term is a subterm of itself.

lemma *subtermTm-refl* [*simp*]: ‹*t* ∈ *set* (*subtermTm t*)›
  by (*induct t*) *simp-all*

The arguments of a predicate are subterms of it.

lemma *subterm-Pre-refl*: ‹*set ts* ⊆ *set* (*subtermFm* (*Pre n ts*))›
  by (*induct ts*) *auto*

The arguments of function are subterms of it.

```
lemma subterm-Fun-refl: ‹set ts ⊆ set (subtermTm (Fun n ts))›
  by (induct ts) auto
```

This function computes the predicates in a formula. We will use this function to help prove the final lemma in this section.

```
primrec preds :: ‹fm ⇒ fm set› where
  ‹preds (Pre n ts) = {Pre n ts}›
| ‹preds (Imp p q) = preds p ∪ preds q›
| ‹preds (Dis p q) = preds p ∪ preds q›
| ‹preds (Con p q) = preds p ∪ preds q›
| ‹preds (Exi p) = preds p›
| ‹preds (Uni p) = preds p›
| ‹preds (Neg p) = preds p›
```

If a term is a subterm of a formula, it is a subterm of some predicate in the formula.

```
lemma subtermFm-preds: ‹t ∈ set (subtermFm p) ⟷ (∃ pre ∈ preds p. t ∈ set
(subtermFm pre))›
  by (induct p) auto

lemma preds-shape: ‹pre ∈ preds p ⟹ ∃ n ts. pre = Pre n ts›
  by (induct p) auto
```

If a function is a subterm of a formula, so are the arguments of that function.

```
lemma fun-arguments-subterm:
  assumes ‹Fun n ts ∈ set (subtermFm p)›
  shows ‹set ts ⊆ set (subtermFm p)›
proof –
  obtain pre where pre: ‹pre ∈ preds p› ‹Fun n ts ∈ set (subtermFm pre)›
    using assms subtermFm-preds by blast
  then obtain n' ts' where ‹pre = Pre n' ts'›
    using preds-shape by blast
  then have ‹set ts ⊆ set (subtermFm pre)›
    using subtermTm-le pre by force
  then have ‹set ts ⊆ set (subtermFm p)›
    using pre subtermFm-preds by blast
  then show ?thesis
    by blast
qed

end
```

2.4 Hintikka sets for SeCaV

```
theory Hintikka
  imports Prover
begin
```

In this theory, we define the concept of a Hintikka set for SeCaV formulas. The definition mirrors the SeCaV proof system such that Hintikka sets are downwards closed with respect to the proof system.

This defines the set of all terms in a set of formulas (containing *Fun 0* [] if it would otherwise be empty).

definition
‹$terms\ H \equiv if\ (\bigcup p \in H.\ set\ (subtermFm\ p)) = \{\}\ then\ \{Fun\ 0\ []\}$
$else\ (\bigcup p \in H.\ set\ (subtermFm\ p))$›

locale *Hintikka* =
fixes *H* :: ‹*fm set*›
assumes
Basic: ‹$Pre\ n\ ts \in H \Longrightarrow Neg\ (Pre\ n\ ts) \notin H$› **and**
AlphaDis: ‹$Dis\ p\ q \in H \Longrightarrow p \in H \land q \in H$› **and**
AlphaImp: ‹$Imp\ p\ q \in H \Longrightarrow Neg\ p \in H \land q \in H$› **and**
AlphaCon: ‹$Neg\ (Con\ p\ q) \in H \Longrightarrow Neg\ p \in H \land Neg\ q \in H$› **and**
BetaCon: ‹$Con\ p\ q \in H \Longrightarrow p \in H \lor q \in H$› **and**
BetaImp: ‹$Neg\ (Imp\ p\ q) \in H \Longrightarrow p \in H \lor Neg\ q \in H$› **and**
BetaDis: ‹$Neg\ (Dis\ p\ q) \in H \Longrightarrow Neg\ p \in H \lor Neg\ q \in H$› **and**
GammaExi: ‹$Exi\ p \in H \Longrightarrow \forall t \in terms\ H.\ sub\ 0\ t\ p \in H$› **and**
GammaUni: ‹$Neg\ (Uni\ p) \in H \Longrightarrow \forall t \in terms\ H.\ Neg\ (sub\ 0\ t\ p) \in H$› **and**
DeltaUni: ‹$Uni\ p \in H \Longrightarrow \exists t \in terms\ H.\ sub\ 0\ t\ p \in H$› **and**
DeltaExi: ‹$Neg\ (Exi\ p) \in H \Longrightarrow \exists t \in terms\ H.\ Neg\ (sub\ 0\ t\ p) \in H$› **and**
Neg: ‹$Neg\ (Neg\ p) \in H \Longrightarrow p \in H$›

end

2.5 Escape path formulas are Hintikka

theory *EPathHintikka* **imports** *Hintikka ProverLemmas* **begin**

In this theory, we show that the formulas in the sequents on a saturated escape path in a proof tree form a Hintikka set. This is a crucial part of our completeness proof.

2.5.1 Definitions

In this section we define a few concepts that make the following proofs easier to read.

pseq is the sequent in a node.

definition *pseq* :: ‹$state \times rule \Rightarrow sequent$› **where**
‹$pseq\ z = snd\ (fst\ z)$›

ptms is the list of terms in a node.

definition *ptms* :: ‹$state \times rule \Rightarrow tm\ list$› **where**
‹$ptms\ z = fst\ (fst\ z)$›

2.5.2 Facts about streams

Escape paths are infinite, so if you drop the first n nodes, you are still on the path.

lemma *epath-sdrop*: ‹*epath steps* $\Longrightarrow$ *epath* (*sdrop n steps*)›
  by (*induct n*) (*auto elim*: *epath.cases*)

Dropping the first n elements of a stream can only reduce the set of elements in the stream.

lemma *sset-sdrop*: ‹*sset* (*sdrop n s*) $\subseteq$ *sset s*›
proof (*induct n arbitrary*: *s*)
  case (*Suc n*)
  then show *?case*
    by (*metis in-mono sdrop-simps*(*2*) *stl-sset subsetI*)
qed *simp*

2.5.3 Transformation of states on an escape path

We need to prove some lemmas about how the states of an escape path are connected.

Since escape paths are well-formed, the eff relation holds between the nodes on the path.

lemma *epath-eff*:
  assumes ‹*epath steps*› ‹*eff* (*snd* (*shd steps*)) (*fst* (*shd steps*)) *ss*›
  shows ‹*fst* (*shd* (*stl steps*)) |∈| *ss*›
  using *assms* **by** (*metis* (*mono-tags*, *lifting*) *epath.simps eff-def*)

The list of terms in a state contains the terms of the current sequent and the terms from the previous state.

lemma *effect-tms*:
  assumes ‹(B, z') |∈| *effect r* (A, z)›
  shows ‹B = *remdups* (A @ *subterms z* @ *subterms* z')›
  using *assms* **by** (*smt* (*verit*, *best*) *effect.simps fempty-iff fimageE prod.simps*(*1*))

The two previous lemmas can be combined into a single lemma.

lemma *epath-effect*:
  assumes ‹*epath steps*› ‹*shd steps* = ((A, z), r)›
  shows ‹$\exists$ B z' r'. (B, z') |∈| *effect r* (A, z) $\wedge$ *shd* (*stl steps*) = ((B, z'), r') $\wedge$
    (B = *remdups* (A @ *subterms z* @ *subterms* z'))›
  using *assms epath-eff effect-tms*
  by (*metis* (*mono-tags*, *lifting*) *eff-def fst-conv prod.collapse snd-conv*)

The list of terms in the next state on an escape path contains the terms in the current state plus the terms from the next state.

lemma *epath-stl-ptms*:
  assumes ‹*epath steps*›

shows ‹*ptms* (*shd* (*stl steps*)) = *remdups* (*ptms* (*shd steps*) @
subterms (*pseq* (*shd steps*)) @ *subterms* (*pseq* (*shd* (*stl steps*))))›
using *assms epath-effect*
by (*metis* (*mono-tags*) *eff-def effect-tms epath-eff pseq-def ptms-def surjective-pairing*)

The list of terms never decreases on an escape path.

lemma *epath-sdrop-ptms*:
assumes ‹*epath steps*›
shows ‹*set* (*ptms* (*shd steps*)) ⊆ *set* (*ptms* (*shd* (*sdrop n steps*)))›
using *assms*
proof (*induct n*)
case (*Suc n*)
then have ‹*epath* (*sdrop n steps*)›
using *epath-sdrop* **by** *blast*
then show *?case*
using *Suc epath-stl-ptms* **by** *fastforce*
qed *simp*

2.5.4 Preservation of formulas on escape paths

If a property will eventually hold on a path, there is some index from which it begins to hold, and before which it does not hold.

lemma *ev-prefix-sdrop*:
assumes ‹*ev* (*holds P*) *xs*›
shows ‹∃ *n*. *list-all* (*not P*) (*stake n xs*) ∧ *holds P* (*sdrop n xs*)›
using *assms*
proof (*induct xs*)
case (*base xs*)
then show *?case*
by (*metis list.pred-inject*(*1*) *sdrop.simps*(*1*) *stake.simps*(*1*))
next
case (*step xs*)
then show *?case*
by (*metis holds.elims*(*1*) *list.pred-inject*(*2*) *list-all-simps*(*2*) *sdrop.simps*(*1*−*2*) *stake.simps*(*1*−*2*))
qed

More specifically, the path will consists of a prefix and a suffix for which the property does not hold and does hold, respectively.

lemma *ev-prefix*:
assumes ‹*ev* (*holds P*) *xs*›
shows ‹∃ *pre suf*. *list-all* (*not P*) *pre* ∧ *holds P suf* ∧ *xs* = *pre* @− *suf*›
using *assms ev-prefix-sdrop* **by** (*metis stake-sdrop*)

All rules are always enabled, so they are also always enabled at specific steps.

lemma *always-enabledAtStep*: ‹*enabledAtStep r xs*›
by (*simp add*: *RuleSystem-Defs.enabled-def eff-def*)

If a formula is in the sequent in the first state of an escape path and none of the rule applications in some prefix of the path affect that formula, the formula will still be in the sequent after that prefix.

lemma *epath-preserves-unaffected*:
  assumes ‹$p \in$ *set* (*pseq* (*shd steps*))› **and** ‹*epath steps*› **and** ‹*steps* = *pre* @− *suf*› **and**
    ‹*list-all* (*not* (λ*step*. *affects* (*snd step*) *p*)) *pre*›
  shows ‹$p \in$ *set* (*pseq* (*shd suf*))›
  using *assms*
proof (*induct pre arbitrary*: *steps*)
  case *Nil*
  then show *?case*
    by *simp*
next
  case (*Cons q pre*)
  from *this*(*3*) **show** *?case*
  proof *cases*
    case (*epath sl*)
    from *this*(*2*−*4*) **show** *?thesis*
     using *Cons*(*1*−*2*, *4*−*5*) *effect-preserves-unaffected* **unfolding** *eff-def pseq-def list-all-def*
      by (*metis* (*no-types*, *lifting*) *list.sel*(*1*) *list.set-intros*(*1*−*2*) *prod.exhaust-sel*
        shift.simps(*2*) *shift-simps*(*1*) *stream.sel*(*2*))
  qed
qed

2.5.5 Formulas on an escape path form a Hintikka set

This definition captures the set of formulas on an entire path

definition ‹*tree-fms steps* ≡ $\bigcup$ *ss* ∈ *sset steps*. *set* (*pseq ss*)›

The sequent at the head of a path is in the set of formulas on that path

lemma *pseq-in-tree-fms*: ‹⟦$x \in$ *sset steps*; $p \in$ *set* (*pseq x*)⟧ ⟹ $p \in$ *tree-fms steps*›
  using *pseq-def tree-fms-def* **by** *blast*

If a formula is in the set of formulas on a path, there is some index on the path where that formula can be found in the sequent.

lemma *tree-fms-in-pseq*: ‹$p \in$ *tree-fms steps* ⟹ ∃ *n*. $p \in$ *set* (*pseq* (*steps* !! *n*))›
  unfolding *pseq-def tree-fms-def* **using** *sset-range*[*of steps*] **by** *simp*

If a path is saturated, so is any suffix of that path (since saturation is defined in terms of the always operator).

lemma *Saturated-sdrop*: ‹*Saturated steps* ⟹ *Saturated* (*sdrop n steps*)›
  by (*simp add*: *RuleSystem-Defs.Saturated-def alw-iff-sdrop saturated-def*)

This is an abbreviation that determines whether a given rule is applied in a given state.

abbreviation ‹*is-rule r step* ≡ *snd step* = *r*›

If a path is saturated, it is always possible to find a state in which a given rule is applied.

lemma *Saturated-ev-rule*:
  assumes ‹*Saturated steps*›
  shows ‹*ev* (*holds* (*is-rule r*)) (*sdrop n steps*)›
proof −
  have ‹*Saturated* (*sdrop n steps*)›
    using ‹*Saturated steps*› *Saturated-sdrop* **by** *fast*
  moreover have ‹*r* ∈ *Prover.R*›
    by (*metis rules-repeat snth-sset*)
  ultimately have ‹*saturated r* (*sdrop n steps*)›
    unfolding *Saturated-def* **by** *fast*
  then show *?thesis*
    unfolding *saturated-def* **using** *always-enabledAtStep holds.elims*(*3*) **by** *blast*
qed

On an escape path, the sequent is never an axiom (since that would end the branch, and escape paths are infinitely long).

lemma *epath-never-branchDone*:
  assumes ‹*epath steps*›
  shows ‹*alw* (*holds* (*not* (*branchDone o pseq*))) *steps*›
proof (*rule ccontr*)
  assume ‹¬ *?thesis*›
  then have ‹*ev* (*holds* (*branchDone o pseq*)) *steps*›
    by (*simp add*: *alw-iff-sdrop ev-iff-sdrop*)
  then obtain *n* **where** *n*: ‹*holds* (*branchDone o pseq*) (*sdrop n steps*)›
    using *sdrop-wait* **by** *blast*
  let *?suf* = ‹*sdrop n steps*›
  have ‹∀ *r A*. *effect r* (*A*, *pseq* (*shd ?suf*)) = {||}›
    unfolding *effect-def* **using** *n* **by** *simp*
  moreover have ‹*epath ?suf*›
    using ‹*epath steps*› *epath-sdrop* **by** *blast*
  then have ‹∀ *r A*. ∃ *z' r'*. *z'* |∈| *effect r* (*A*, *pseq* (*shd ?suf*)) ∧ *shd* (*stl ?suf*) = (*z'*, *r'*)›
    using *epath-effect* **by** (*metis calculation prod.exhaust-sel pseq-def*)
  ultimately show *False*
    by *blast*
qed

Finally we arrive at the main result of this theory: The set of formulas on a saturated escape path form a Hintikka set.

The proof basically says that, given a formula, we can find some index into the path where a rule is applied to decompose that formula into the parts needed for the Hintikka set. The lemmas above are used to guarantee that the formula does not disappear (and that the branch does not end) before the rule is applied, and that the correct formulas are generated by the effect function when the rule is finally applied. For Beta rules, only one of the constituent formulas need to be on the path, since the path runs along only one of the two branches. For Gamma and Delta rules, the construction of the list of terms in each state guarantees that the formulas are instantiated with terms in the Hintikka set.

lemma *escape-path-Hintikka*:
  assumes ‹*epath steps*› **and** ‹*Saturated steps*›
  shows ‹*Hintikka* (*tree-fms steps*)›
    (**is** ‹*Hintikka ?H*›)
proof
  fix *n ts*
  assume *pre*: ‹*Pre n ts* ∈ *?H*›
  then obtain *m* **where** *m*: ‹*Pre n ts* ∈ *set* (*pseq* (*shd* (*sdrop m steps*)))›
    using *tree-fms-in-pseq* **by** *auto*

  show ‹*Neg* (*Pre n ts*) ∉ *?H*›
  proof
    assume ‹*Neg* (*Pre n ts*) ∈ *?H*›
    then obtain *k* **where** *k*: ‹*Neg* (*Pre n ts*) ∈ *set* (*pseq* (*shd* (*sdrop k steps*)))›
      using *tree-fms-in-pseq* **by** *auto*

    let *?pre* = ‹*stake* (*m* + *k*) *steps*›
    let *?suf* = ‹*sdrop* (*m* + *k*) *steps*›

    have
      1: ‹¬ *affects r* (*Pre n ts*)› **and**
      2: ‹¬ *affects r* (*Neg* (*Pre n ts*))› **for** *r*
      unfolding *affects-def* **by** (*cases r*, *simp-all*)+

    have ‹*list-all* (*not* (λ*step*. *affects* (*snd step*) (*Pre n ts*))) *?pre*›
      unfolding *list-all-def* **using** *1* **by** (*induct ?pre*) *simp-all*
    then have *p*: ‹*Pre n ts* ∈ *set* (*pseq* (*shd ?suf*))›
      using ‹*epath steps*› *epath-preserves-unaffected m epath-sdrop*
      by (*metis* (*no-types*, *lifting*) *list.pred-mono-strong list-all-append*
          sdrop-add stake-add stake-sdrop)

    have ‹*list-all* (*not* (λ*step*. *affects* (*snd step*) (*Neg* (*Pre n ts*)))) *?pre*›
      unfolding *list-all-def* **using** *2* **by** (*induct ?pre*) *simp-all*
    then have *np*: ‹*Neg* (*Pre n ts*) ∈ *set* (*pseq* (*shd ?suf*))›
      using ‹*epath steps*› *epath-preserves-unaffected k epath-sdrop*
     by (*smt* (*verit*, *best*) *add.commute list.pred-mono-strong list-all-append sdrop-add*
          stake-add stake-sdrop)

    have ‹*holds* (*branchDone o pseq*) *?suf*›
      using *p np branchDone-contradiction* **by** *auto*
    moreover have ‹¬ *holds* (*branchDone o pseq*) *?suf*›
      using ‹*epath steps*› *epath-never-branchDone* **by** (*simp add*: *alw-iff-sdrop*)
    ultimately show *False*

by *blast*
qed
next
fix *p q*
assume ‹*Dis p q* ∈ *?H*› (**is** ‹*?f* ∈ *?H*›)
then obtain *n* **where** *n*: ‹*?f* ∈ *set* (*pseq* (*shd* (*sdrop n steps*)))›
using *tree-fms-in-pseq* **by** *auto*
let *?steps* = ‹*sdrop n steps*›
let *?r* = *AlphaDis*
have ‹*ev* (*holds* (*is-rule ?r*)) *?steps*›
using ‹*Saturated steps*› *Saturated-ev-rule* **by** *blast*
then obtain *pre suf* **where**
pre: ‹*list-all* (*not* (*is-rule ?r*)) *pre*› **and**
suf: ‹*holds* (*is-rule ?r*) *suf*› **and**
ori: ‹*?steps* = *pre* @− *suf*›
using *ev-prefix* **by** *blast*

have ‹*affects r ?f* ⟷ *r* = *?r*› **for** *r*
unfolding *affects-def* **by** (*cases r*) *simp-all*
then have ‹*list-all* (*not* (λ*step*. *affects* (*snd step*) *?f*)) *pre*›
using *pre* **by** *simp*
moreover have ‹*epath* (*pre* @− *suf*)›
using ‹*epath steps*› *epath-sdrop ori* **by** *metis*
ultimately have ‹*?f* ∈ *set* (*pseq* (*shd suf*))›
using *epath-preserves-unaffected n ori* **by** *metis*

moreover have ‹*epath suf*›
using ‹*epath* (*pre* @− *suf*)› *epath-sdrop* **by** (*metis alwD alw-iff-sdrop alw-shift*)
then obtain *B z′ r′* **where**
z′: ‹(*B*, *z′*) |∈| *effect ?r* (*ptms* (*shd suf*), *pseq* (*shd suf*))› ‹*shd* (*stl suf*) = ((*B*, *z′*), *r′*)›
using *suf epath-effect* **unfolding** *pseq-def ptms-def*
by (*metis* (*mono-tags*, *lifting*) *holds.elims*(*2*) *prod.collapse*)
ultimately have ‹*p* ∈ *set z′*› ‹*q* ∈ *set z′*›
using *parts-in-effect* **unfolding** *parts-def* **by** *fastforce+*

then show ‹*p* ∈ *?H* ∧ *q* ∈ *?H*›
using *z′*(*2*) *ori pseq-in-tree-fms*
by (*metis* (*no-types*, *opaque-lifting*) *Un-iff fst-conv pseq-def shd-sset snd-conv sset-sdrop*
sset-shift stl-sset subset-eq)
next
fix *p q*
assume ‹*Imp p q* ∈ *tree-fms steps*› (**is** ‹*?f* ∈ *?H*›)
then obtain *n* **where** *n*: ‹*?f* ∈ *set* (*pseq* (*shd* (*sdrop n steps*)))›
using *tree-fms-in-pseq* **by** *auto*
let *?steps* = ‹*sdrop n steps*›
let *?r* = *AlphaImp*
have ‹*ev* (*holds* (*is-rule ?r*)) *?steps*›

using ‹*Saturated steps*› *Saturated-ev-rule* **by** *blast*
then obtain *pre suf* **where**
pre: ‹*list-all* (*not* (*is-rule ?r*)) *pre*› **and**
suf: ‹*holds* (*is-rule ?r*) *suf*› **and**
ori: ‹*?steps* = *pre* @− *suf*›
using *ev-prefix* **by** *blast*

have ‹*affects r ?f* ⟷ *r* = *?r*› **for** *r*
unfolding *affects-def* **by** (*cases r*) *simp-all*
then have ‹*list-all* (*not* (λ*step*. *affects* (*snd step*) *?f*)) *pre*›
using *pre* **by** *simp*
moreover have ‹*epath* (*pre* @− *suf*)›
using ‹*epath steps*› *epath-sdrop ori* **by** *metis*
ultimately have ‹*?f* ∈ *set* (*pseq* (*shd suf*))›
using *epath-preserves-unaffected n ori* **by** *metis*

moreover have ‹*epath suf*›
using ‹*epath* (*pre* @− *suf*)› *epath-sdrop* **by** (*metis alwD alw-iff-sdrop alw-shift*)
then obtain *B z′ r′* **where**
z′: ‹(*B*, *z′*) |∈| *effect ?r* (*ptms* (*shd suf*), *pseq* (*shd suf*))› ‹*shd* (*stl suf*) = ((*B*, *z′*), *r′*)›
using *suf epath-effect* **unfolding** *pseq-def ptms-def*
by (*metis* (*mono-tags*, *lifting*) *holds.elims*(*2*) *prod.collapse*)
ultimately have ‹*Neg p* ∈ *set z′*› ‹*q* ∈ *set z′*›
using *parts-in-effect* **unfolding** *parts-def* **by** *fastforce*+

then show ‹*Neg p* ∈ *?H* ∧ *q* ∈ *?H*›
using *z′*(*2*) *ori pseq-in-tree-fms*
by (*metis* (*no-types*, *opaque-lifting*) *Un-iff fst-conv pseq-def shd-sset snd-conv sset-sdrop*
sset-shift stl-sset subset-eq)
next
fix *p q*
assume ‹*Neg* (*Con p q*) ∈ *?H*› (**is** ‹*?f* ∈ *?H*›)
then obtain *n* **where** *n*: ‹*?f* ∈ *set* (*pseq* (*shd* (*sdrop n steps*)))›
using *tree-fms-in-pseq* **by** *auto*
let *?steps* = ‹*sdrop n steps*›
let *?r* = *AlphaCon*
have ‹*ev* (*holds* (*is-rule ?r*)) *?steps*›
using ‹*Saturated steps*› *Saturated-ev-rule* **by** *blast*
then obtain *pre suf* **where**
pre: ‹*list-all* (*not* (*is-rule ?r*)) *pre*› **and**
suf: ‹*holds* (*is-rule ?r*) *suf*› **and**
ori: ‹*?steps* = *pre* @− *suf*›
using *ev-prefix* **by** *blast*

have ‹*affects r ?f* ⟷ *r* = *?r*› **for** *r*
unfolding *affects-def* **by** (*cases r*) *simp-all*
then have ‹*list-all* (*not* (λ*step*. *affects* (*snd step*) *?f*)) *pre*›

using *pre* **by** *simp*
moreover have ‹*epath* (*pre* @− *suf*)›
using ‹*epath steps*› *epath-sdrop ori* **by** *metis*
ultimately have ‹*?f* ∈ *set* (*pseq* (*shd suf*))›
using *epath-preserves-unaffected n ori* **by** *metis*

moreover have ‹*epath suf*›
using ‹*epath* (*pre* @− *suf*)› *epath-sdrop* **by** (*metis alwD alw-iff-sdrop alw-shift*)
then obtain *B z′ r′* **where**
z′: ‹(*B*, *z′*) |∈| *effect ?r* (*ptms* (*shd suf*), *pseq* (*shd suf*))› ‹*shd* (*stl suf*) = ((*B*, *z′*), *r′*)›
using *suf epath-effect* **unfolding** *pseq-def ptms-def*
by (*metis* (*mono-tags*, *lifting*) *holds.elims*(*2*) *prod.collapse*)
ultimately have ‹*Neg p* ∈ *set z′*› ‹*Neg q* ∈ *set z′*›
using *parts-in-effect* **unfolding** *parts-def* **by** *fastforce+*

then show ‹*Neg p* ∈ *?H* ∧ *Neg q* ∈ *?H*›
using *z′*(*2*) *ori pseq-in-tree-fms*
by (*metis* (*no-types*, *opaque-lifting*) *Un-iff fst-conv pseq-def shd-sset snd-conv sset-sdrop*
sset-shift stl-sset subset-eq)
next
fix *p q*
assume ‹*Con p q* ∈ *?H*› (**is** ‹*?f* ∈ *?H*›)
then obtain *n* **where** *n*: ‹*?f* ∈ *set* (*pseq* (*shd* (*sdrop n steps*)))›
using *tree-fms-in-pseq* **by** *auto*
let *?steps* = ‹*sdrop n steps*›
let *?r* = *BetaCon*
have ‹*ev* (*holds* (*is-rule ?r*)) *?steps*›
using ‹*Saturated steps*› *Saturated-ev-rule* **by** *blast*
then obtain *pre suf* **where**
pre: ‹*list-all* (*not* (*is-rule ?r*)) *pre*› **and**
suf: ‹*holds* (*is-rule ?r*) *suf*› **and**
ori: ‹*?steps* = *pre* @− *suf*›
using *ev-prefix* **by** *blast*

have ‹*affects r ?f* ⟷ *r* = *?r*› **for** *r*
unfolding *affects-def* **by** (*cases r*) *simp-all*
then have ‹*list-all* (*not* (λ*step*. *affects* (*snd step*) *?f*)) *pre*›
using *pre* **by** *simp*
moreover have ‹*epath* (*pre* @− *suf*)›
using ‹*epath steps*› *epath-sdrop ori* **by** *metis*
ultimately have ‹*?f* ∈ *set* (*pseq* (*shd suf*))›
using *epath-preserves-unaffected n ori* **by** *metis*

moreover have ‹*epath suf*›
using ‹*epath* (*pre* @− *suf*)› *epath-sdrop* **by** (*metis alwD alw-iff-sdrop alw-shift*)
then obtain *B z′ r′* **where**
z′: ‹(*B*, *z′*) |∈| *effect ?r* (*ptms* (*shd suf*), *pseq* (*shd suf*))› ‹*shd* (*stl suf*) = ((*B*,

z'), *r'*)›
  using *suf epath-effect* **unfolding** *pseq-def ptms-def*
  by (*metis* (*mono-tags*, *lifting*) *holds.elims*(*2*) *prod.collapse*)
 ultimately consider ‹*p* ∈ *set z'*› | ‹*q* ∈ *set z'*›
  using *parts-in-effect* **unfolding** *parts-def* **by** *fastforce*

 then show ‹*p* ∈ *?H* ∨ *q* ∈ *?H*›
  using *z'*(*2*) *ori pseq-in-tree-fms*
  by (*metis* (*no-types*, *opaque-lifting*) *Un-iff fst-conv pseq-def shd-sset snd-conv sset-sdrop*
   sset-shift stl-sset subset-eq)
next
 fix *p q*
 assume ‹*Neg* (*Imp p q*) ∈ *?H*› (**is** ‹*?f* ∈ *?H*›)
 then obtain *n* **where** *n*: ‹*?f* ∈ *set* (*pseq* (*shd* (*sdrop n steps*)))›
  using *tree-fms-in-pseq* **by** *auto*
 let *?steps* = ‹*sdrop n steps*›
 let *?r* = *BetaImp*
 have ‹*ev* (*holds* (*is-rule ?r*)) *?steps*›
  using ‹*Saturated steps*› *Saturated-ev-rule* **by** *blast*
 then obtain *pre suf* **where**
  pre: ‹*list-all* (*not* (*is-rule ?r*)) *pre*› **and**
  suf: ‹*holds* (*is-rule ?r*) *suf*› **and**
  ori: ‹*?steps* = *pre* @− *suf*›
  using *ev-prefix* **by** *blast*

 have ‹*affects r ?f* ⟷ *r* = *?r*› **for** *r*
  unfolding *affects-def* **by** (*cases r*) *simp-all*
 then have ‹*list-all* (*not* (λ*step*. *affects* (*snd step*) *?f*)) *pre*›
  using *pre* **by** *simp*
 moreover have ‹*epath* (*pre* @− *suf*)›
  using ‹*epath steps*› *epath-sdrop ori* **by** *metis*
 ultimately have ‹*?f* ∈ *set* (*pseq* (*shd suf*))›
  using *epath-preserves-unaffected n ori* **by** *metis*

 moreover have ‹*epath suf*›
  using ‹*epath* (*pre* @− *suf*)› *epath-sdrop* **by** (*metis alwD alw-iff-sdrop alw-shift*)
 then obtain *B z' r'* **where**
  z': ‹(*B*, *z'*) |∈| *effect ?r* (*ptms* (*shd suf*), *pseq* (*shd suf*))› ‹*shd* (*stl suf*) = ((*B*, *z'*), *r'*)›
  using *suf epath-effect* **unfolding** *pseq-def ptms-def*
  by (*metis* (*mono-tags*, *lifting*) *holds.elims*(*2*) *prod.collapse*)
 ultimately consider ‹*p* ∈ *set z'*› | ‹*Neg q* ∈ *set z'*›
  using *parts-in-effect* **unfolding** *parts-def* **by** *fastforce*

 then show ‹*p* ∈ *?H* ∨ *Neg q* ∈ *?H*›
  using *z'*(*2*) *ori pseq-in-tree-fms*
  by (*metis* (*no-types*, *opaque-lifting*) *Un-iff fst-conv pseq-def shd-sset snd-conv sset-sdrop*

```
      sset-shift stl-sset subset-eq)
next
  fix p q
  assume ‹Neg (Dis p q) ∈ ?H› (is ‹?f ∈ ?H›)
  then obtain n where n: ‹?f ∈ set (pseq (shd (sdrop n steps)))›
    using tree-fms-in-pseq by auto
  let ?steps = ‹sdrop n steps›
  let ?r = BetaDis
  have ‹ev (holds (is-rule ?r)) ?steps›
    using ‹Saturated steps› Saturated-ev-rule by blast
  then obtain pre suf where
    pre: ‹list-all (not (is-rule ?r)) pre› and
    suf: ‹holds (is-rule ?r) suf› and
    ori: ‹?steps = pre @- suf›
    using ev-prefix by blast

  have ‹affects r ?f ⟷ r = ?r› for r
    unfolding affects-def by (cases r) simp-all
  then have ‹list-all (not (λstep. affects (snd step) ?f)) pre›
    using pre by simp
  moreover have ‹epath (pre @- suf)›
    using ‹epath steps› epath-sdrop ori by metis
  ultimately have ‹?f ∈ set (pseq (shd suf))›
    using epath-preserves-unaffected n ori by metis

  moreover have ‹epath suf›
   using ‹epath (pre @- suf)› epath-sdrop by (metis alwD alw-iff-sdrop alw-shift)
  then obtain B z' r' where
    z': ‹(B, z') |∈| effect ?r (ptms (shd suf), pseq (shd suf))› ‹shd (stl suf) = ((B,
z'), r')›
    using suf epath-effect unfolding pseq-def ptms-def
    by (metis (mono-tags, lifting) holds.elims(2) prod.collapse)
  ultimately consider ‹Neg p ∈ set z'› | ‹Neg q ∈ set z'›
    using parts-in-effect unfolding parts-def by fastforce

  then show ‹Neg p ∈ ?H ∨ Neg q ∈ ?H›
    using z'(2) ori pseq-in-tree-fms
     by (metis (no-types, opaque-lifting) Un-iff fst-conv pseq-def shd-sset snd-conv
sset-sdrop
      sset-shift stl-sset subset-eq)
next
  fix p
  assume ‹Exi p ∈ ?H› (is ‹?f ∈ ?H›)
  then obtain n where n: ‹?f ∈ set (pseq (shd (sdrop n steps)))›
    using tree-fms-in-pseq by auto

  let ?r = GammaExi

  show ‹∀ t ∈ terms ?H. sub 0 t p ∈ ?H›
```

```
  proof
    fix t
    assume t: ‹t ∈ terms ?H›
    show ‹sub 0 t p ∈ ?H›
    proof –
      have ‹∃ m. t ∈ set (subterms (pseq (shd (sdrop m steps))))›
      proof (cases ‹(⋃ f ∈ ?H. set (subtermFm f)) = {}›)
        case True
        moreover have ‹∀ p ∈ set (pseq (shd steps)). p ∈ ?H›
          unfolding tree-fms-def by (metis pseq-in-tree-fms shd-sset tree-fms-def)
        ultimately have ‹∀ p ∈ set (pseq (shd steps)). subtermFm p = []›
          by simp
        then have ‹concat (map subtermFm (pseq (shd steps))) = []›
          by (induct ‹pseq (shd steps)›) simp-all
        then have ‹subterms (pseq (shd steps)) = [Fun 0 []]›
          unfolding subterms-def by (metis list.simps(4) remdups-eq-nil-iff)
        then show ?thesis
          using True t unfolding terms-def
          by (metis empty-iff insert-iff list.set-intros(1) sdrop.simps(1))
      next
        case False
        then obtain pt where pt: ‹t ∈ set (subtermFm pt)› ‹pt ∈ ?H›
          using t unfolding terms-def by (metis (no-types, lifting) UN-E)
        then obtain m where m: ‹pt ∈ set (pseq (shd (sdrop m steps)))›
          using tree-fms-in-pseq by auto
        then show ?thesis
          using pt(1) set-subterms by fastforce
      qed
      then obtain m where ‹t ∈ set (subterms (pseq (shd (sdrop m steps))))›
        by blast
      then have ‹t ∈ set (ptms (shd (stl (sdrop m steps))))›
        using epath-stl-ptms epath-sdrop ‹epath steps›
        by (metis (no-types, opaque-lifting) Un-iff set-append set-remdups)
      moreover have ‹epath (stl (sdrop m steps))›
        using epath-sdrop ‹epath steps› by (meson epath.cases)
      ultimately have ‹∀ k ≥ m. t ∈ set (ptms (shd (stl (sdrop k steps))))›
          using epath-sdrop-ptms by (metis (no-types, lifting) le-Suc-ex sdrop-add
sdrop-stl subsetD)
      then have above: ‹∀ k > m. t ∈ set (ptms (shd (sdrop k steps)))›
      by (metis Nat.lessE less-irrefl-nat less-trans-Suc linorder-not-less sdrop-simps(2))

      let ?pre = ‹stake (n + m + 1) steps›
      let ?suf = ‹sdrop (n + m + 1) steps›

      have ∗: ‹¬ affects r ?f› for r
        unfolding affects-def by (cases r, simp-all)+

      have ‹ev (holds (is-rule ?r)) ?suf›
        using ‹Saturated steps› Saturated-ev-rule by blast
```

```
      then obtain pre suf k where
        pre: ‹list-all (not (is-rule ?r)) pre› and
        suf: ‹holds (is-rule ?r) suf› and
        ori: ‹pre = stake k ?suf› ‹suf = sdrop k ?suf›
        using ev-prefix-sdrop by blast

      have k: ‹∃ k > m. suf = sdrop k steps›
           using ori by (meson le-add2 less-add-one order-le-less-trans sdrop-add
trans-less-add1)

      have ‹list-all (not (λstep. affects (snd step) ?f)) ?pre›
        unfolding list-all-def using ∗ by (induct ?pre) simp-all
      then have ‹?f ∈ set (pseq (shd ?suf))›
        using ‹epath steps› epath-preserves-unaffected n epath-sdrop
        by (metis (no-types, lifting) list.pred-mono-strong list-all-append
            sdrop-add stake-add stake-sdrop)
      then have ‹?f ∈ set (pseq (shd suf))›
        using ‹epath steps› epath-preserves-unaffected n epath-sdrop ∗ ori
        by (metis (no-types, lifting) list.pred-mono-strong pre stake-sdrop)
      moreover have ‹epath suf›
        using ‹epath steps› epath-sdrop ori by blast
      then obtain B z′ r′ where
        z′: ‹(B, z′) |∈| effect ?r (ptms (shd suf), pseq (shd suf))›
        ‹shd (stl suf) = ((B, z′), r′)›
        using suf epath-effect unfolding pseq-def ptms-def
        by (metis (mono-tags, lifting) holds.elims(2) prod.collapse)

      moreover have ‹t ∈ set (ptms (shd suf))›
        using above k by (meson le-add2 less-add-one order-le-less-trans)
      ultimately have ‹sub 0 t p ∈ set z′›
         using parts-in-effect[where A=‹ptms (shd suf)›] unfolding parts-def by
fastforce
      then show ?thesis
        using k pseq-in-tree-fms z′(2)
          by (metis Pair-inject in-mono prod.collapse pseq-def shd-sset sset-sdrop
stl-sset)
    qed
  qed
next
  fix p
  assume ‹Neg (Uni p) ∈ ?H› (is ‹?f ∈ ?H›)
  then obtain n where n: ‹?f ∈ set (pseq (shd (sdrop n steps)))›
    using tree-fms-in-pseq by auto

  let ?r = GammaUni

  show ‹∀ t ∈ terms ?H. Neg (sub 0 t p) ∈ ?H›
  proof
    fix t
```

```
    assume t: ‹t ∈ terms ?H›
    show ‹Neg (sub 0 t p) ∈ ?H›
    proof −
      have ‹∃ m. t ∈ set (subterms (pseq (shd (sdrop m steps))))›
      proof (cases ‹(⋃ f ∈ ?H. set (subtermFm f)) = {}›)
        case True
        moreover have ‹∀ p ∈ set (pseq (shd steps)). p ∈ ?H›
          unfolding tree-fms-def by (metis pseq-in-tree-fms shd-sset tree-fms-def)
        ultimately have ‹∀ p ∈ set (pseq (shd steps)). subtermFm p = []›
          by simp
        then have ‹concat (map subtermFm (pseq (shd steps))) = []›
          by (induct ‹pseq (shd steps)›) simp-all
        then have ‹subterms (pseq (shd steps)) = [Fun 0 []]›
          unfolding subterms-def by (metis list.simps(4) remdups-eq-nil-iff)
        then show ?thesis
          using True t unfolding terms-def
          by (metis empty-iff insert-iff list.set-intros(1) sdrop.simps(1))
      next
        case False
        then obtain pt where pt: ‹t ∈ set (subtermFm pt)› ‹pt ∈ ?H›
          using t unfolding terms-def by (metis (no-types, lifting) UN-E)
        then obtain m where m: ‹pt ∈ set (pseq (shd (sdrop m steps)))›
          using tree-fms-in-pseq by auto
        then show ?thesis
          using pt(1) set-subterms by fastforce
      qed
      then obtain m where ‹t ∈ set (subterms (pseq (shd (sdrop m steps))))›
        by blast
      then have ‹t ∈ set (ptms (shd (stl (sdrop m steps))))›
        using epath-stl-ptms epath-sdrop ‹epath steps›
        by (metis (no-types, lifting) Un-iff set-append set-remdups)
      moreover have ‹epath (stl (sdrop m steps))›
        using epath-sdrop ‹epath steps› by (meson epath.cases)
      ultimately have ‹∀ k ≥ m. t ∈ set (ptms (shd (stl (sdrop k steps))))›
          using epath-sdrop-ptms by (metis (no-types, lifting) le-Suc-ex sdrop-add
sdrop-stl subsetD)
      then have above: ‹∀ k > m. t ∈ set (ptms (shd (sdrop k steps)))›
      by (metis Nat.lessE less-irrefl-nat less-trans-Suc linorder-not-less sdrop-simps(2))

      let ?pre = ‹stake (n + m + 1) steps›
      let ?suf = ‹sdrop (n + m + 1) steps›

      have *: ‹¬ affects r ?f› for r
        unfolding affects-def by (cases r, simp-all)+

      have ‹ev (holds (is-rule ?r)) ?suf›
        using ‹Saturated steps› Saturated-ev-rule by blast
      then obtain pre suf k where
        pre: ‹list-all (not (is-rule ?r)) pre› and
```

```
      suf: ‹holds (is-rule ?r) suf› and
      ori: ‹pre = stake k ?suf› ‹suf = sdrop k ?suf›
      using ev-prefix-sdrop by blast

    have k: ‹∃ k > m. suf = sdrop k steps›
        using ori by (meson le-add2 less-add-one order-le-less-trans sdrop-add trans-less-add1)

    have ‹list-all (not (λstep. affects (snd step) ?f)) ?pre›
      unfolding list-all-def using ∗ by (induct ?pre) simp-all
    then have ‹?f ∈ set (pseq (shd ?suf))›
      using ‹epath steps› epath-preserves-unaffected n epath-sdrop
      by (metis (no-types, lifting) list.pred-mono-strong list-all-append
          sdrop-add stake-add stake-sdrop)
    then have ‹?f ∈ set (pseq (shd suf))›
      using ‹epath steps› epath-preserves-unaffected n epath-sdrop ∗ ori
      by (metis (no-types, lifting) list.pred-mono-strong pre stake-sdrop)
    moreover have ‹epath suf›
      using ‹epath steps› epath-sdrop ori by blast
    then obtain B z′ r′ where
      z′: ‹(B, z′) |∈| effect ?r (ptms (shd suf), pseq (shd suf))›
      ‹shd (stl suf) = ((B, z′), r′)›
      using suf epath-effect unfolding pseq-def ptms-def
      by (metis (mono-tags, lifting) holds.elims(2) prod.collapse)

    moreover have ‹t ∈ set (ptms (shd suf))›
      using above k by (meson le-add2 less-add-one order-le-less-trans)
    ultimately have ‹Neg (sub 0 t p) ∈ set z′›
       using parts-in-effect[where A=‹ptms (shd suf)›] unfolding parts-def by fastforce
    then show ?thesis
      using k pseq-in-tree-fms z′(2)
         by (metis Pair-inject in-mono prod.collapse pseq-def shd-sset sset-sdrop stl-sset)
   qed
 qed
next
 fix p
 assume ‹Uni p ∈ tree-fms steps› (is ‹?f ∈ ?H›)
 then obtain n where n: ‹?f ∈ set (pseq (shd (sdrop n steps)))›
   using tree-fms-in-pseq by auto
 let ?steps = ‹sdrop n steps›
 let ?r = DeltaUni
 have ‹ev (holds (is-rule ?r)) ?steps›
   using ‹Saturated steps› Saturated-ev-rule by blast
 then obtain pre suf where
   pre: ‹list-all (not (is-rule ?r)) pre› and
   suf: ‹holds (is-rule ?r) suf› and
   ori: ‹?steps = pre @− suf›
```

using *ev-prefix* **by** *blast*

have ‹*affects r ?f* ⟷ *r* = *?r*› **for** *r*
unfolding *affects-def* **by** (*cases r*) *simp-all*
then have ‹*list-all* (*not* (λ*step. affects* (*snd step*) *?f*)) *pre*›
using *pre* **by** *simp*
moreover have ‹*epath* (*pre* @− *suf*)›
using ‹*epath steps*› *epath-sdrop ori* **by** *metis*
ultimately have ‹*?f* ∈ *set* (*pseq* (*shd suf*))›
using *epath-preserves-unaffected n ori* **by** *metis*

moreover have ‹*epath suf*›
using ‹*epath* (*pre* @− *suf*)› *epath-sdrop* **by** (*metis alwD alw-iff-sdrop alw-shift*)
then obtain *B z′ r′* **where**
z′: ‹(*B*, *z′*) |∈| *effect ?r* (*ptms* (*shd suf*), *pseq* (*shd suf*))› ‹*shd* (*stl suf*) = ((*B*, *z′*), *r′*)›
using *suf epath-effect* **unfolding** *pseq-def ptms-def*
by (*metis* (*mono-tags*, *lifting*) *holds.elims*(*2*) *prod.collapse*)
ultimately obtain *C* **where**
C: ‹*set* (*ptms* (*shd suf*)) ⊆ *set C*› ‹*sub 0* (*Fun* (*generateNew C*) []) *p* ∈ *set z′*›
using *parts-in-effect*[**where** *B*=*B* **and** *z′*=‹*z′*› **and** *z*=‹*pseq* (*shd suf*)› **and** *r*=‹*?r*› **and** *p*=‹*Uni p*›]
unfolding *parts-def* **by** *auto*
then have ∗: ‹*sub 0* (*Fun* (*generateNew C*) []) *p* ∈ *?H*›
using *z′*(*2*) *ori pseq-in-tree-fms*
by (*metis* (*no-types*, *lifting*) *Pair-inject Un-iff in-mono prod.collapse pseq-def shd-sset*
sset-sdrop sset-shift stl-sset)
let *?t* = ‹*Fun* (*generateNew C*) []›
show ‹∃ *t* ∈ *terms ?H*. *sub 0 t p* ∈ *?H*›
proof (*cases* ‹*?t* ∈ *set* (*subtermFm* (*sub 0 ?t p*))›)
case *True*
then have ‹*?t* ∈ *terms ?H*›
unfolding *terms-def* **using** ∗ **by** (*metis UN-I empty-iff*)
then show *?thesis*
using ∗ **by** *blast*
next
case *False*
then have ‹*sub 0 t p* = *sub 0 ?t p*› **for** *t*
using *sub-const-transfer* **by** *metis*
moreover have ‹*terms ?H* ≠ {}›
unfolding *terms-def* **by** *simp*
then have ‹∃ *t*. *t* ∈ *terms ?H*›
by *blast*
ultimately show *?thesis*
using ∗ **by** *metis*
qed
next
fix *p*

assume ‹*Neg* (*Exi p*) ∈ *tree-fms steps*› (**is** ‹*?f* ∈ *?H*›)
then obtain *n* **where** *n*: ‹*?f* ∈ *set* (*pseq* (*shd* (*sdrop n steps*)))›
  using *tree-fms-in-pseq* **by** *auto*
let *?steps* = ‹*sdrop n steps*›
let *?r* = *DeltaExi*
have ‹*ev* (*holds* (*is-rule ?r*)) *?steps*›
  using ‹*Saturated steps*› *Saturated-ev-rule* **by** *blast*
then obtain *pre suf* **where**
  pre: ‹*list-all* (*not* (*is-rule ?r*)) *pre*› **and**
  suf: ‹*holds* (*is-rule ?r*) *suf*› **and**
  ori: ‹*?steps* = *pre* @− *suf*›
  using *ev-prefix* **by** *blast*

have ‹*affects r ?f* ⟷ *r* = *?r*› **for** *r*
  unfolding *affects-def* **by** (*cases r*) *simp-all*
then have ‹*list-all* (*not* (*λstep. affects* (*snd step*) *?f*)) *pre*›
  using *pre* **by** *simp*
moreover have ‹*epath* (*pre* @− *suf*)›
  using ‹*epath steps*› *epath-sdrop ori* **by** *metis*
ultimately have ‹*?f* ∈ *set* (*pseq* (*shd suf*))›
  using *epath-preserves-unaffected n ori* **by** *metis*

moreover have ‹*epath suf*›
  using ‹*epath* (*pre* @− *suf*)› *epath-sdrop* **by** (*metis alwD alw-iff-sdrop alw-shift*)
then obtain *B z′ r′* **where**
  z′: ‹(*B*, *z′*) |∈| *effect ?r* (*ptms* (*shd suf*), *pseq* (*shd suf*))› ‹*shd* (*stl suf*) = ((*B*, *z′*), *r′*)›
  using *suf epath-effect* **unfolding** *pseq-def ptms-def*
  by (*metis* (*mono-tags*, *lifting*) *holds.elims*(*2*) *prod.collapse*)
ultimately obtain *C* **where**
  C: ‹*set* (*ptms* (*shd suf*)) ⊆ *set C*› ‹*Neg* (*sub 0* (*Fun* (*generateNew C*) []) *p*) ∈ *set z′*›
  using *parts-in-effect*[**where** *B*=*B* **and** *z′*=*z′* **and** *z*=‹*pseq* (*shd suf*)› **and** *r*=‹*?r*› **and** *p*=‹*Neg* (*Exi p*)›]
  unfolding *parts-def* **by** *auto*
then have ∗: ‹*Neg* (*sub 0* (*Fun* (*generateNew C*) []) *p*) ∈ *?H*›
  using *z′*(*2*) *ori pseq-in-tree-fms*
  by (*metis* (*no-types*, *lifting*) *Pair-inject Un-iff in-mono prod.collapse pseq-def shd-sset*
    sset-sdrop sset-shift stl-sset)
let *?t* = ‹*Fun* (*generateNew C*) []›
show ‹∃ *t* ∈ *terms ?H*. *Neg* (*sub 0 t p*) ∈ *?H*›
proof (*cases* ‹*?t* ∈ *set* (*subtermFm* (*Neg* (*sub 0 ?t p*)))›)
  case *True*
  then have ‹*?t* ∈ *terms ?H*›
    unfolding *terms-def* **using** ∗ **by** (*metis UN-I empty-iff*)
  then show *?thesis*
    using ∗ **by** *blast*
next

```
    case False
    then have ‹Neg (sub 0 t p) = Neg (sub 0 ?t p)› for t
      using sub-const-transfer by (metis subtermFm.simps(7))
    moreover have ‹terms ?H ≠ {}›
      unfolding terms-def by simp
    then have ‹∃ t. t ∈ terms ?H›
      by blast
    ultimately show ?thesis
      using * by metis
  qed
next
  fix p
  assume ‹Neg (Neg p) ∈ tree-fms steps› (is ‹?f ∈ ?H›)
  then obtain n where n: ‹?f ∈ set (pseq (shd (sdrop n steps)))›
    using tree-fms-in-pseq by auto
  let ?steps = ‹sdrop n steps›
  let ?r = NegNeg
  have ‹ev (holds (is-rule ?r)) ?steps›
    using ‹Saturated steps› Saturated-ev-rule by blast
  then obtain pre suf where
    pre: ‹list-all (not (is-rule ?r)) pre› and
    suf: ‹holds (is-rule ?r) suf› and
    ori: ‹?steps = pre @− suf›
    using ev-prefix by blast

  have ‹affects r ?f ⟷ r = ?r› for r
    unfolding affects-def by (cases r) simp-all
  then have ‹list-all (not (λstep. affects (snd step) ?f)) pre›
    using pre by simp
  moreover have ‹epath (pre @− suf)›
    using ‹epath steps› epath-sdrop ori by metis
  ultimately have ‹?f ∈ set (pseq (shd suf))›
    using epath-preserves-unaffected n ori by metis

  moreover have ‹epath suf›
   using ‹epath (pre @− suf)› epath-sdrop by (metis alwD alw-iff-sdrop alw-shift)
  then obtain B z' r' where
    z': ‹(B, z') |∈| effect ?r (ptms (shd suf), pseq (shd suf))› ‹shd (stl suf) = ((B,
z'), r')›
    using suf epath-effect unfolding pseq-def ptms-def
    by (metis (mono-tags, lifting) holds.elims(2) prod.collapse)
  ultimately have ‹p ∈ set z'›
    using parts-in-effect unfolding parts-def by fastforce

  then show ‹p ∈ ?H›
    using z'(2) ori pseq-in-tree-fms
     by (metis (no-types, lifting) Pair-inject Un-iff in-mono prod.collapse pseq-def
shd-sset
        sset-sdrop sset-shift stl-sset)
```

qed

end

2.6 Bounded semantics

theory *Usemantics* **imports** *SeCaV* **begin**

In this theory, we define an alternative semantics for SeCaV formulas where the quantifiers are bounded to terms in a specific set. This is needed to construct a countermodel from a Hintikka set.

This function defines the semantics, which are bounded by the set u.

primrec *usemantics* **where**
‹$usemantics\ u\ e\ f\ g\ (Pre\ i\ l) = g\ i\ (semantics\text{-}list\ e\ f\ l)$›
| ‹$usemantics\ u\ e\ f\ g\ (Imp\ p\ q) = (usemantics\ u\ e\ f\ g\ p \longrightarrow usemantics\ u\ e\ f\ g\ q)$›
| ‹$usemantics\ u\ e\ f\ g\ (Dis\ p\ q) = (usemantics\ u\ e\ f\ g\ p \lor usemantics\ u\ e\ f\ g\ q)$›
| ‹$usemantics\ u\ e\ f\ g\ (Con\ p\ q) = (usemantics\ u\ e\ f\ g\ p \land usemantics\ u\ e\ f\ g\ q)$›
| ‹$usemantics\ u\ e\ f\ g\ (Exi\ p) = (\exists\, x \in u.\ usemantics\ u\ (SeCaV.shift\ e\ 0\ x)\ f\ g\ p)$›
| ‹$usemantics\ u\ e\ f\ g\ (Uni\ p) = (\forall\, x \in u.\ usemantics\ u\ (SeCaV.shift\ e\ 0\ x)\ f\ g\ p)$›
| ‹$usemantics\ u\ e\ f\ g\ (Neg\ p) = (\neg\ usemantics\ u\ e\ f\ g\ p)$›

An environment is well-formed if the variables are actually in the quantifier set u.

definition *is-env* :: ‹$'a\ set \Rightarrow (nat \Rightarrow 'a) \Rightarrow bool$› **where**
‹$is\text{-}env\ u\ e \equiv \forall\, n.\ e\ n \in u$›

A function interpretation is well-formed if it is closed in the quantifier set u.

definition *is-fdenot* :: ‹$'a\ set \Rightarrow (nat \Rightarrow 'a\ list \Rightarrow 'a) \Rightarrow bool$› **where**
‹$is\text{-}fdenot\ u\ f \equiv \forall\, i\ l.\ list\text{-}all\ (\lambda x.\ x \in u)\ l \longrightarrow f\ i\ l \in u$›

If we choose to quantify over the universal set, we obtain the usual semantics

lemma *usemantics-UNIV*: ‹$usemantics\ UNIV\ e\ f\ g\ p \longleftrightarrow semantics\ e\ f\ g\ p$›
by (*induct p arbitrary*: *e*) *auto*

If a function name n is not in a formula, it does not matter whether it is in the function interpretation or not.

lemma *uupd-lemma* [*iff*]: ‹$n \notin params\ p \Longrightarrow usemantics\ u\ e\ (f(n := x))\ g\ p \longleftrightarrow usemantics\ u\ e\ f\ g\ p$›
by (*induct p arbitrary*: *e*) *simp-all*

The semantics of substituting variable i by term t in formula a are well-defined

lemma *usubst-lemma* [*iff*]:
‹$usemantics\ u\ e\ f\ g\ (subst\ a\ t\ i) \longleftrightarrow usemantics\ u\ (SeCaV.shift\ e\ i\ (semantics\text{-}term\ e\ f\ t))\ f\ g\ a$›
by (*induct a arbitrary*: *e i t*) *simp-all*

Soundness of SeCaV with regards to the bounded semantics

We would like to prove that the SeCaV proof system is sound under the bounded semantics.

If the environment and the function interpretation are well-formed, the semantics of terms are in the quantifier set *u*.

lemma *usemantics-term* [*simp*]:
  assumes ‹*is-env u e*› ‹*is-fdenot u f*›
  shows ‹*semantics-term e f t* ∈ *u*› ‹*list-all* (λ*x*. *x* ∈ *u*) (*semantics-list e f ts*)›
  using *assms* **by** (*induct t* **and** *ts rule*: *semantics-term.induct semantics-list.induct*)
    (*simp-all add*: *is-env-def is-fdenot-def*)

If a function interpretation is well-formed, replacing the value by one in the quantifier set results in a well-formed function interpretation.

lemma *is-fdenot-shift* [*simp*]: ‹*is-fdenot u f* ⟹ *x* ∈ *u* ⟹ *is-fdenot u* (*f*(*i* := λ-. *x*))›
  unfolding *is-fdenot-def SeCaV.shift-def* **by** *simp*

If a sequent is provable in the SeCaV proof system and the environment and function interpretation are well-formed, the sequent is valid under the bounded semantics.

theorem *sound-usemantics*:
  assumes ‹⊩ *z*› **and** ‹*is-env u e*› **and** ‹*is-fdenot u f*›
  shows ‹∃ *p* ∈ *set z*. *usemantics u e f g p*›
  using *assms*
proof (*induct arbitrary*: *f rule*: *sequent-calculus.induct*)
  case (*10 i p z*)
  then show *?case*
  proof (*cases* ‹∀ *x* ∈ *u*. *usemantics u e* (*f*(*i* := λ-. *x*)) *g* (*sub 0* (*Fun i* []) *p*)›)
    case *False*
    moreover have ‹∀ *x* ∈ *u*. ∃ *p* ∈ *set* (*sub 0* (*Fun i* []) *p* # *z*). *usemantics u e* (*f*(*i* := λ-. *x*)) *g p*›
      using *10 is-fdenot-shift* **by** *metis*
    ultimately have ‹∃ *x* ∈ *u*. ∃ *p* ∈ *set z*. *usemantics u e* (*f*(*i* := λ-. *x*)) *g p*›
      by *fastforce*
    moreover have ‹*list-all* (λ*p*. *i* ∉ *params p*) *z*›
      using *10* **by** *simp*
    ultimately show *?thesis*
      using *10 Ball-set insert-iff list.set*(*2*) *uupd-lemma* **by** *metis*
  qed *simp*
next
  case (*11 i p z*)
  then show *?case*
  proof (*cases* ‹∀ *x* ∈ *u*. *usemantics u e* (*f*(*i* := λ-. *x*)) *g* (*Neg* (*sub 0* (*Fun i* []) *p*))›)
    case *False*
    moreover have

‹$\forall x \in u.\ \exists p \in$ *set* (*Neg* (*sub 0* (*Fun i* [])) *p*) # *z*). *usemantics u e* ($f(i := \lambda$-. x)) *g p*›
using *11 is-fdenot-shift* **by** *metis*
ultimately have ‹$\exists x \in u.\ \exists p \in$ *set z*. *usemantics u e* ($f(i := \lambda$-. x)) *g p*›
by *fastforce*
moreover have ‹*list-all* ($\lambda p.\ i \notin$ *params p*) *z*›
using *11* **by** *simp*
ultimately show *?thesis*
using *11 Ball-set insert-iff list.set*(*2*) *uupd-lemma* **by** *metis*
qed *simp*
qed *fastforce*+

end

2.7 Countermodels from Hintikka sets

theory *Countermodel*
imports *Hintikka Usemantics ProverLemmas*
begin

In this theory, we will construct a countermodel in the bounded semantics from a Hintikka set. This will allow us to prove completeness of the prover.

A predicate is satisfied in the model based on a set of formulas S when its negation is in S.

abbreviation (*input*)
‹*G S n ts* $\equiv$ *Neg* (*Pre n ts*) $\in$ *S*›

Alternate interpretation for environments: if a variable is not present, we interpret it as some existing term.

abbreviation
‹*E S n* $\equiv$ *if Var n* $\in$ *terms S then Var n else SOME t. t* $\in$ *terms S*›

Alternate interpretation for functions: if a function application is not present, we interpret it as some existing term.

abbreviation
‹*F S i l* $\equiv$ *if Fun i l* $\in$ *terms S then Fun i l else SOME t. t* $\in$ *terms S*›

The terms function never returns the empty set (because it will add *Fun 0* [] if that is the case).

lemma *terms-ne* [*simp*]: ‹*terms S* $\neq$ {}›
unfolding *terms-def* **by** *simp*

If a term is in the set of terms, it is either the default term or a subterm of some formula in the set.

lemma *terms-cases*: ‹$t \in$ *terms S* $\Longrightarrow$ t = *Fun 0* [] $\vee$ ($\exists p \in S.\ t \in$ *set* (*subtermFm p*))›

unfolding *terms-def* **by** (*simp split*: *if-splits*)

The set of terms is downwards closed under the subterm function.

```
lemma terms-downwards-closed: ‹t ∈ terms S ⟹ set (subtermTm t) ⊆ terms S›
proof (induct t)
  case (Fun n ts)
  moreover have ‹∀ t ∈ set ts. t ∈ set ts›
    by simp
  moreover have ‹∀ t ∈ set ts. t ∈ terms S›
  proof
    fix t
    assume *: ‹t ∈ set ts›
    then show ‹t ∈ terms S›
    proof (cases ‹terms S = {Fun 0 []}›)
      case True
      then show ?thesis
        using Fun * by simp
    next
      case False
      moreover obtain p where p: ‹p ∈ S› ‹Fun n ts ∈ set (subtermFm p)›
        using Fun(2) terms-cases * by fastforce
      then have ‹set ts ⊆ set (subtermFm p)›
        using fun-arguments-subterm by blast
      ultimately show ‹t ∈ terms S›
        unfolding terms-def using * p(1) by (metis UN-iff in-mono)
    qed
  qed
  ultimately have ‹∀ t ∈ set ts. set (subtermTm t) ⊆ terms S›
    using Fun by meson
  moreover note ‹Fun n ts ∈ terms S›
  ultimately show ?case
    by auto
next
  case (Var x)
  then show ?case
    by simp
qed
```

If terms are actually in a set of formulas, interpreting the environment over these formulas allows for a Herbrand interpretation.

```
lemma usemantics-E:
    ‹t ∈ terms S ⟹ semantics-term (E S) (F S) t = t›
    ‹list-all (λt. t ∈ terms S) ts ⟹ semantics-list (E S) (F S) ts = ts›
proof (induct t and ts arbitrary: ts rule: semantics-term.induct semantics-list.induct)
  case (Fun i ts′)
  moreover have ‹∀ t′ ∈ set ts′. t′ ∈ set (subtermTm (Fun i ts′))›
    using subterm-Fun-refl by blast
  ultimately have ‹list-all (λt. t ∈ terms S) ts′›
```

using *terms-downwards-closed* **unfolding** *list-all-def* **by** (*metis* (*no-types*, *lifting*) *subsetD*)
then show *?case*
using *Fun* **by** *simp*
qed *simp-all*

Our alternate interpretation of environments is well-formed for the terms function.

lemma *is-env-E*:
‹*is-env* (*terms S*) (*E S*)›
unfolding *is-env-def*
proof
fix *n*
show ‹*E S n* ∈ *terms S*›
by (*cases* ‹*Var n* ∈ *terms S*›) (*simp-all add*: *some-in-eq*)
qed

Our alternate function interpretation is well-formed for the terms function.

lemma *is-fdenot-F*:
‹*is-fdenot* (*terms S*) (*F S*)›
unfolding *is-fdenot-def*
proof (*intro allI impI*)
fix *i l*
assume ‹*list-all* (λ*x*. *x* ∈ *terms S*) *l*›
then show ‹*F S i l* ∈ *terms S*›
by (*cases* ‹∀ *n*. *Var n* ∈ *terms S*›) (*simp-all add*: *some-in-eq*)
qed

abbreviation
‹*M S* ≡ *usemantics* (*terms S*) (*E S*) (*F S*) (*G S*)›

If S is a Hintikka set, then we can construct a countermodel for any formula using our bounded semantics and a Herbrand interpretation.

theorem *Hintikka-counter-model*:
assumes ‹*Hintikka S*›
shows ‹(*p* ∈ *S* ⟶ ¬ *M S p*) ∧ (*Neg p* ∈ *S* ⟶ *M S p*)›
proof (*induct p rule*: *wf-induct* [**where** *r*=‹*measure size*›])
case *1*
then show *?case* **..**
next
fix *x*
assume *wf*: ‹∀ *q*. (*q*, *x*) ∈ *measure size* ⟶
(*q* ∈ *S* ⟶ ¬ *M S q*) ∧ (*Neg q* ∈ *S* ⟶ *M S q*)›
show ‹(*x* ∈ *S* ⟶ ¬ *M S x*) ∧ (*Neg x* ∈ *S* ⟶ *M S x*)›
proof (*cases x*)
case (*Pre n ts*)
show *?thesis*
proof (*intro conjI impI*)

```
      assume ‹x ∈ S›
      then have ‹Neg (Pre n ts) ∉ S›
        using assms Pre Hintikka.Basic by blast
      moreover have ‹list-all (λt. t ∈ terms S) ts›
       using ‹x ∈ S› Pre subterm-Pre-refl unfolding terms-def list-all-def by force
      ultimately show ‹¬ M S x›
        using Pre usemantics-E
        by (metis (no-types, lifting) usemantics.simps(1))
    next
      assume ‹Neg x ∈ S›
      then have ‹G S n ts›
        using assms Pre Hintikka.Basic by blast
      moreover have ‹list-all (λt. t ∈ terms S) ts›
       using ‹Neg x ∈ S› Pre subterm-Pre-refl unfolding terms-def list-all-def by
force
      ultimately show ‹M S x›
        using Pre usemantics-E
        by (metis (no-types, lifting) usemantics.simps(1))
    qed
  next
    case (Imp p q)
    show ?thesis
    proof (intro conjI impI)
      assume ‹x ∈ S›
      then have ‹Neg p ∈ S› ‹q ∈ S›
        using Imp assms Hintikka.AlphaImp by blast+
      then show ‹¬ M S x›
        using wf Imp by fastforce
    next
      assume ‹Neg x ∈ S›
      then have ‹p ∈ S ∨ Neg q ∈ S›
        using Imp assms Hintikka.BetaImp by blast
      then show ‹M S x›
        using wf Imp by fastforce
    qed
  next
    case (Dis p q)
    show ?thesis
    proof (intro conjI impI)
      assume ‹x ∈ S›
      then have ‹p ∈ S› ‹q ∈ S›
        using Dis assms Hintikka.AlphaDis by blast+
      then show ‹¬ M S x›
        using wf Dis by fastforce
    next
      assume ‹Neg x ∈ S›
      then have ‹Neg p ∈ S ∨ Neg q ∈ S›
        using Dis assms Hintikka.BetaDis by blast
      then show ‹M S x›
```

```
      using wf Dis by fastforce
    qed
  next
    case (Con p q)
    show ?thesis
    proof (intro conjI impI)
      assume ‹x ∈ S›
      then have ‹p ∈ S ∨ q ∈ S›
        using Con assms Hintikka.BetaCon by blast
      then show ‹¬ M S x›
        using wf Con by fastforce
    next
      assume ‹Neg x ∈ S›
      then have ‹Neg p ∈ S› ‹Neg q ∈ S›
        using Con assms Hintikka.AlphaCon by blast+
      then show ‹M S x›
        using wf Con by fastforce
    qed
  next
    case (Exi p)
    show ?thesis
    proof (intro conjI impI)
      assume ‹x ∈ S›
      then have ‹∀ t ∈ terms S. sub 0 t p ∈ S›
        using Exi assms Hintikka.GammaExi by blast
      then have ‹∀ t ∈ terms S. ¬ M S (sub 0 t p)›
        using wf Exi size-sub
          by (metis (no-types, lifting) add.right-neutral add-Suc-right fm.size(12)
in-measure lessI)
      moreover have ‹∀ t ∈ terms S. semantics-term (E S) (F S) t = t›
        using usemantics-E(1) terms-downwards-closed unfolding list-all-def by
blast
      ultimately have ‹∀ t ∈ terms S. ¬ usemantics (terms S) (SeCaV.shift (E
S) 0 t) (F S) (G S) p›
        by simp
      then show ‹¬ M S x›
        using Exi by simp
    next
      assume ‹Neg x ∈ S›
      then obtain t where ‹t ∈ terms S› ‹Neg (sub 0 t p) ∈ S›
        using Exi assms Hintikka.DeltaExi by metis
      then have ‹M S (sub 0 t p)›
        using wf Exi size-sub
          by (metis (no-types, lifting) add.right-neutral add-Suc-right fm.size(12)
in-measure lessI)
      moreover have ‹semantics-term (E S) (F S) t = t›
        using ‹t ∈ terms S› usemantics-E(1) terms-downwards-closed unfolding
list-all-def by blast
      ultimately show ‹M S x›
```

using *Exi* ‹$t \in$ *terms* S› **by** *auto*
qed
next
case (*Uni p*)
show *?thesis*
proof (*intro conjI impI*)
assume ‹$x \in S$›
then obtain t **where** ‹$t \in$ *terms* S› ‹*sub* *0* t $p \in S$›
using *Uni assms Hintikka.DeltaUni* **by** *metis*
then have ‹$\neg$ M S (*sub* *0* t p)›
using *wf Uni size-sub*
by (*metis* (*no-types*, *lifting*) *add.right-neutral add-Suc-right fm.size*(*13*) *in-measure lessI*)
moreover have ‹*semantics-term* (E S) (F S) $t = t$›
using ‹$t \in$ *terms* S› *usemantics-E*(*1*) *terms-downwards-closed* **unfolding** *list-all-def* **by** *blast*
ultimately show ‹$\neg$ M S x›
using *Uni* ‹$t \in$ *terms* S› **by** *auto*
next
assume ‹*Neg* $x \in S$›
then have ‹$\forall\, t \in$ *terms* S. *Neg* (*sub* *0* t p) $\in S$›
using *Uni assms Hintikka.GammaUni* **by** *blast*
then have ‹$\forall\, t \in$ *terms* S. M S (*sub* *0* t p)›
using *wf Uni size-sub*
by (*metis* (*no-types*, *lifting*) *Nat.add-0-right add-Suc-right fm.size*(*13*) *in-measure lessI*)
moreover have ‹$\forall\, t \in$ *terms* S. *semantics-term* (E S) (F S) $t = t$›
using *usemantics-E*(*1*) *terms-downwards-closed* **unfolding** *list-all-def* **by** *blast*
ultimately have ‹$\forall\, t \in$ *terms* S. $\neg$ *usemantics* (*terms* S) (*SeCaV.shift* (E S) *0* t) (F S) (G S) (*Neg* p)›
by *simp*
then show ‹M S x›
using *Uni* **by** *simp*
qed
next
case (*Neg p*)
show *?thesis*
proof (*intro conjI impI*)
assume ‹$x \in S$›
then show ‹$\neg$ M S x›
using *wf Neg* **by** *fastforce*
next
assume ‹*Neg* $x \in S$›
then have ‹$p \in S$›
using *Neg assms Hintikka.Neg* **by** *blast*
then show ‹M S x›
using *wf Neg* **by** *fastforce*
qed

```
  qed
qed

end
```

2.8 Soundness

```
theory Soundness
  imports ProverLemmas
begin
```

In this theory, we prove that the prover is sound with regards to the SeCaV proof system using the abstract soundness framework.

If some suffix of the sequents in all of the children of a state are provable, so is some suffix of the sequent in the current state, with the prefix in each sequent being the same. (As a side condition, the lists of terms need to be compatible.)

```
lemma SeCaV-children-pre:
  assumes ‹∀ z' ∈ set (children A r z). (⊩ pre @ z')›
    and ‹paramss (pre @ z) ⊆ paramsts A›
  shows ‹⊩ pre @ z›
  using assms
proof (induct z arbitrary: pre A)
  case Nil
  then show ?case
    by simp
next
  case (Cons p z)
  then have ih: ‹∀ z' ∈ set (children A r z). (⊩ pre @ z') ⟹ (⊩ pre @ z)›
    if ‹paramss (pre @ z) ⊆ paramsts A› for pre A
    using that by simp

  let ?A = ‹remdups (A @ subtermFms (concat (parts A r p)))›

  have A: ‹paramss (pre @ p # z) ⊆ paramsts ?A›
    using paramsts-subset Cons.prems(2) by fastforce

  have ‹∀ z' ∈ set (list-prod (parts A r p) (children ?A r z)). (⊩ pre @ z')›
    using Cons.prems by (metis children.simps(2))
  then have ‹∀ z' ∈ {hs @ ts |hs ts. hs ∈ set (parts A r p) ∧ ts ∈ set (children ?A
r z)}.
      (⊩ pre @ z')›
    using list-prod-is-cartesian by blast
  then have *:
    ‹∀ hs ∈ set (parts A r p). ∀ ts ∈ set (children ?A r z). (⊩ pre @ hs @ ts)›
    by blast
  then show ?case
```

```
  proof (cases r p rule: parts-exhaust)
    case (AlphaDis p q)
    then have ‹∀ z' ∈ set (children ?A r z). (⊩ pre @ p # q # z')›
      using ∗ unfolding parts-def by simp
    then have ‹∀ z' ∈ set (children ?A r z). (⊩ (pre @ [p, q]) @ z')›
      by simp
    then have ‹⊩ pre @ p # q # z›
      using AlphaDis ih[where pre=‹pre @ [p, q]› and A=‹?A›] A by simp
    then have ‹⊩ p # q # pre @ z›
      using Ext by simp
    then have ‹⊩ Dis p q # pre @ z›
      using SeCaV.AlphaDis by blast
    then show ?thesis
      using AlphaDis Ext by simp
  next
    case (AlphaImp p q)
    then have ‹∀ z' ∈ set (children ?A r z). (⊩ pre @ Neg p # q # z')›
      using ∗ unfolding parts-def by simp
    then have ‹∀ z' ∈ set (children ?A r z). (⊩ (pre @ [Neg p, q]) @ z')›
      by simp
    then have ‹⊩ pre @ Neg p # q # z›
      using AlphaImp ih[where pre=‹pre @ [Neg p, q]› and A=‹?A›] A by simp
    then have ‹⊩ Neg p # q # pre @ z›
      using Ext by simp
    then have ‹⊩ Imp p q # pre @ z›
      using SeCaV.AlphaImp by blast
    then show ?thesis
      using AlphaImp Ext by simp
  next
    case (AlphaCon p q)
    then have ‹∀ z' ∈ set (children ?A r z). (⊩ pre @ Neg p # Neg q # z')›
      using ∗ unfolding parts-def by simp
    then have ‹∀ z' ∈ set (children ?A r z). (⊩ (pre @ [Neg p, Neg q]) @ z')›
      by simp
    then have ‹⊩ pre @ Neg p # Neg q # z›
       using AlphaCon ih[where pre=‹pre @ [Neg p, Neg q]› and A=‹?A›] A by
simp
    then have ‹⊩ Neg p # Neg q # pre @ z›
      using Ext by simp
    then have ‹⊩ Neg (Con p q) # pre @ z›
      using SeCaV.AlphaCon by blast
    then show ?thesis
      using AlphaCon Ext by simp
  next
    case (BetaCon p q)
    then have
      ‹∀ z' ∈ set (children ?A r z). (⊩ pre @ p # z')›
      ‹∀ z' ∈ set (children ?A r z). (⊩ pre @ q # z')›
      using ∗ unfolding parts-def by simp-all
```

```
  then have
    ‹∀ z' ∈ set (children ?A r z). (⊩ (pre @ [p]) @ z')›
    ‹∀ z' ∈ set (children ?A r z). (⊩ (pre @ [q]) @ z')›
    by simp-all
  then have ‹⊩ pre @ p # z› ‹⊩ pre @ q # z›
    using BetaCon ih[where pre=‹pre @ [-]› and A=‹?A›] A by simp-all
  then have ‹⊩ p # pre @ z› ‹⊩ q # pre @ z›
    using Ext by simp-all
  then have ‹⊩ Con p q # pre @ z›
    using SeCaV.BetaCon by blast
  then show ?thesis
    using BetaCon Ext by simp
next
  case (BetaImp p q)
  then have
    ‹∀ z' ∈ set (children ?A r z). (⊩ pre @ p # z')›
    ‹∀ z' ∈ set (children ?A r z). (⊩ pre @ Neg q # z')›
    using * unfolding parts-def by simp-all
  then have
    ‹∀ z' ∈ set (children ?A r z). (⊩ (pre @ [p]) @ z')›
    ‹∀ z' ∈ set (children ?A r z). (⊩ (pre @ [Neg q]) @ z')›
    by simp-all
  then have ‹⊩ pre @ p # z› ‹⊩ pre @ Neg q # z›
    using BetaImp ih ih[where pre=‹pre @ [-]› and A=‹?A›] A by simp-all
  then have ‹⊩ p # pre @ z› ‹⊩ Neg q # pre @ z›
    using Ext by simp-all
  then have ‹⊩ Neg (Imp p q) # pre @ z›
    using SeCaV.BetaImp by blast
  then show ?thesis
    using BetaImp Ext by simp
next
  case (BetaDis p q)
  then have
    ‹∀ z' ∈ set (children ?A r z). (⊩ pre @ Neg p # z')›
    ‹∀ z' ∈ set (children ?A r z). (⊩ pre @ Neg q # z')›
    using * unfolding parts-def by simp-all
  then have
    ‹∀ z' ∈ set (children ?A r z). (⊩ (pre @ [Neg p]) @ z')›
    ‹∀ z' ∈ set (children ?A r z). (⊩ (pre @ [Neg q]) @ z')›
    by simp-all
  then have ‹⊩ pre @ Neg p # z› ‹⊩ pre @ Neg q # z›
    using BetaDis ih[where pre=‹pre @ [-]› and A=‹?A›] A by simp-all
  then have ‹⊩ Neg p # pre @ z› ‹⊩ Neg q # pre @ z›
    using Ext by simp-all
  then have ‹⊩ Neg (Dis p q) # pre @ z›
    using SeCaV.BetaDis by blast
  then show ?thesis
    using BetaDis Ext by simp
next
```

case (*DeltaUni p*)
let *?i* = ‹*generateNew A*›
have ‹∀ *z′* ∈ *set* (*children ?A r z*). (⊩ *pre* @ *sub 0* (*Fun ?i* []) *p* # *z′*)›
using *DeltaUni* ∗ **unfolding** *parts-def* **by** *simp*
then have ‹∀ *z′* ∈ *set* (*children ?A r z*). (⊩ (*pre* @ [*sub 0* (*Fun ?i* []) *p*]) @ *z′*)›
by *simp*
moreover have ‹*set* (*subtermFm* (*sub 0* (*Fun ?i* []) *p*)) ⊆ *set ?A*›
using *DeltaUni* **unfolding** *parts-def* **by** *simp*
then have ‹*params* (*sub 0* (*Fun ?i* []) *p*) ⊆ *paramsts ?A*›
using *subtermFm-subset-params* **by** *blast*
ultimately have ‹⊩ *pre* @ *sub 0* (*Fun ?i* []) *p* # *z*›
using *DeltaUni ih*[**where** *pre*=‹*pre* @ [-]› **and** *A*=‹*?A*›] *A* **by** *simp*
then have ‹⊩ *sub 0* (*Fun ?i* []) *p* # *pre* @ *z*›
using *Ext* **by** *simp*
moreover have ‹*?i* ∉ *paramsts A*›
by (*induct A*) (*metis Suc-max-new generateNew-def listFunTm-paramst*(*2*) *plus-1-eq-Suc*)+
then have ‹*news ?i* (*p* # *pre* @ *z*)›
using *DeltaUni Cons.prems*(*2*) *news-paramss* **by** *auto*
ultimately have ‹⊩ *Uni p* # *pre* @ *z*›
using *SeCaV.DeltaUni* **by** *blast*
then show *?thesis*
using *DeltaUni Ext* **by** *simp*
next
case (*DeltaExi p*)
let *?i* = ‹*generateNew A*›
have ‹∀ *z′* ∈ *set* (*children ?A r z*). (⊩ *pre* @ *Neg* (*sub 0* (*Fun ?i* []) *p*) # *z′*)›
using *DeltaExi* ∗ **unfolding** *parts-def* **by** *simp*
then have ‹∀ *z′* ∈ *set* (*children ?A r z*). (⊩ (*pre* @ [*Neg* (*sub 0* (*Fun ?i* []) *p*)]) @ *z′*)›
by *simp*
moreover have ‹*set* (*subtermFm* (*sub 0* (*Fun ?i* []) *p*)) ⊆ *set ?A*›
using *DeltaExi* **unfolding** *parts-def* **by** *simp*
then have ‹*params* (*sub 0* (*Fun ?i* []) *p*) ⊆ *paramsts ?A*›
using *subtermFm-subset-params* **by** *blast*
ultimately have ‹⊩ *pre* @ *Neg* (*sub 0* (*Fun ?i* []) *p*) # *z*›
using *DeltaExi ih*[**where** *pre*=‹*pre* @ [-]› **and** *A*=‹*?A*›] *A* **by** *simp*
then have ‹⊩ *Neg* (*sub 0* (*Fun ?i* []) *p*) # *pre* @ *z*›
using *Ext* **by** *simp*
moreover have ‹*?i* ∉ *paramsts A*›
by (*induct A*) (*metis Suc-max-new generateNew-def listFunTm-paramst*(*2*) *plus-1-eq-Suc*)+
then have ‹*news ?i* (*p* # *pre* @ *z*)›
using *DeltaExi Cons.prems*(*2*) *news-paramss* **by** *auto*
ultimately have ‹⊩ *Neg* (*Exi p*) # *pre* @ *z*›
using *SeCaV.DeltaExi* **by** *blast*
then show *?thesis*
using *DeltaExi Ext* **by** *simp*
next

```
    case (NegNeg p)
    then have ‹∀ z' ∈ set (children ?A r z). (⊩ pre @ p # z')›
      using * unfolding parts-def by simp
    then have ‹∀ z' ∈ set (children ?A r z). (⊩ (pre @ [p]) @ z')›
      by simp
    then have ‹⊩ pre @ p # z›
      using NegNeg ih[where pre=‹pre @ [-]› and A=‹?A›] A by simp
    then have ‹⊩ p # pre @ z›
      using Ext by simp
    then have ‹⊩ Neg (Neg p) # pre @ z›
      using SeCaV.Neg by blast
    then show ?thesis
      using NegNeg Ext by simp
  next
    case (GammaExi p)
    then have ‹∀ z' ∈ set (children ?A r z). (⊩ pre @ Exi p # map (λt. sub 0 t p)
A @ z')›
      using * unfolding parts-def by simp
    then have ‹∀ z' ∈ set (children ?A r z). (⊩ ((pre @ Exi p # map (λt. sub 0 t
p) A) @ z'))›
      by simp
    moreover have ‹∀ t ∈ set A. params (sub 0 t p) ⊆ paramsts A ∪ params p›
      using params-sub by fastforce
    then have ‹∀ t ∈ set A. params (sub 0 t p) ⊆ paramsts ?A›
       using GammaExi A by fastforce
    then have ‹paramss (map (λt. sub 0 t p) A) ⊆ paramsts ?A›
      by auto
    ultimately have ‹⊩ pre @ Exi p # map (λt. sub 0 t p) A @ z›
      using GammaExi ih[where pre=‹pre @ Exi p # map - A› and A=‹?A›] A
by simp
    moreover have ‹ext (map (λt. sub 0 t p) A @ Exi p # pre @ z)
        (pre @ Exi p # map (λt. sub 0 t p) A @ z)›
      by auto
    ultimately have ‹⊩ map (λt. sub 0 t p) A @ Exi p # pre @ z›
      using Ext by blast
    then have ‹⊩ Exi p # pre @ z›
    proof (induct A)
      case Nil
      then show ?case
        by simp
    next
      case (Cons a A)
      then have ‹⊩ Exi p # map (λt. sub 0 t p) A @ Exi p # pre @ z›
        using SeCaV.GammaExi by simp
      then have ‹⊩ map (λt. sub 0 t p) A @ Exi p # pre @ z›
        using Ext by simp
      then show ?case
        using Cons.hyps by blast
    qed
```

```
    then show ?thesis
      using GammaExi Ext by simp
  next
    case (GammaUni p)
    then have ‹∀ z' ∈ set (children ?A r z). (⊩ pre @ Neg (Uni p) # map (λt. Neg
(sub 0 t p)) A @ z')›
      using ∗ unfolding parts-def by simp
    then have ‹∀ z' ∈ set (children ?A r z). (⊩ ((pre @ Neg (Uni p) # map (λt.
Neg (sub 0 t p)) A) @ z'))›
      by simp
    moreover have ‹∀ t ∈ set A. params (sub 0 t p) ⊆ paramsts A ∪ params p›
      using params-sub by fastforce
    then have ‹∀ t ∈ set A. params (sub 0 t p) ⊆ paramsts ?A›
      using GammaUni A by fastforce
    then have ‹paramss (map (λt. sub 0 t p) A) ⊆ paramsts ?A›
      by auto
    ultimately have ‹⊩ pre @ Neg (Uni p) # map (λt. Neg (sub 0 t p)) A @ z›
      using GammaUni ih[where pre=‹pre @ Neg (Uni p) # map - A› and
A=‹?A›] A by simp
    moreover have ‹ext (map (λt. Neg (sub 0 t p)) A @ Neg (Uni p) # pre @ z)
        (pre @ Neg (Uni p) # map (λt. Neg (sub 0 t p)) A @ z)›
      by auto
    ultimately have ‹⊩ map (λt. Neg (sub 0 t p)) A @ Neg (Uni p) # pre @ z›
      using Ext by blast
    then have ‹⊩ Neg (Uni p) # pre @ z›
    proof (induct A)
      case Nil
      then show ?case
        by simp
    next
      case (Cons a A)
      then have ‹⊩ Neg (Uni p) # map (λt. Neg (sub 0 t p)) A @ Neg (Uni p) #
pre @ z›
        using SeCaV.GammaUni by simp
      then have ‹⊩ map (λt. Neg (sub 0 t p)) A @ Neg (Uni p) # pre @ z›
        using Ext by simp
      then show ?case
        using Cons.hyps by blast
    qed
    then show ?thesis
      using GammaUni Ext by simp
  next
    case Other
    then have ‹∀ z' ∈ set (children ?A r z). (⊩ (pre @ [p]) @ z')›
      using ∗ by simp
    then show ?thesis
      using ih[where pre=‹pre @ [p]› and A=‹?A›] A by simp
  qed
qed
```

As a special case, the prefix can be empty.

```
corollary SeCaV-children:
  assumes ‹∀ z' ∈ set (children A r z). (⊩ z')› and ‹paramss z ⊆ paramsts A›
  shows ‹⊩ z›
  using SeCaV-children-pre assms by (metis append-Nil)
```

Using this lemma, we can instantiate the abstract soundness framework.

```
interpretation Soundness eff rules UNIV ‹λ- (A, z). (⊩ z)›
  unfolding Soundness-def
proof safe
  fix r A z ss S
  assume r-enabled: ‹eff r (A, z) ss›

  assume ‹∀ s'. s' |∈| ss ⟶ (∀ S ∈ UNIV. case s' of (A, z) ⇒ ⊩ z)›
  then have next-sound: ‹∀ B z. (B, z) |∈| ss ⟶ (⊩ z)›
    by simp

  show ‹⊩ z›
  proof (cases ‹branchDone z›)
    case True
    then obtain p where ‹p ∈ set z› ‹Neg p ∈ set z›
      using branchDone-contradiction by blast
    then show ?thesis
      using Ext Basic by fastforce
  next
    case False
    let ?A = ‹remdups (A @ subtermFms z)›
    have ‹∀ z' ∈ set (children ?A r z). (⊩ z')›
      using False r-enabled eff-children next-sound by blast
    moreover have ‹set (subtermFms z) ⊆ set ?A›
      by simp
    then have ‹paramss z ⊆ paramsts ?A›
      using subtermFm-subset-params by fastforce
    ultimately show ‹⊩ z›
      using SeCaV-children by blast
  qed
qed
```

Using the result from the abstract soundness framework, we can finally state our soundness result: for a finite, well-formed proof tree, the sequent at the root of the tree is provable in the SeCaV proof system.

```
theorem prover-soundness-SeCaV:
  assumes ‹tfinite t› and ‹wf t›
  shows ‹⊩ rootSequent t›
  using assms soundness by fastforce

end
```

2.9 Completeness

```
theory Completeness
  imports Countermodel EPathHintikka
begin
```

In this theory, we prove that the prover is complete with regards to the SeCaV proof system using the abstract completeness framework.

We start out by specializing the abstract completeness theorem to our prover. It is necessary to reproduce the final theorem here so we can alter it to state that our prover produces a proof tree instead of simply stating that a proof tree exists.

```
theorem epath-prover-completeness:
  fixes A :: ‹tm list› and z :: ‹fm list›
  defines ‹t ≡ secavProver (A, z)›
  shows ‹(fst (root t) = (A, z) ∧ wf t ∧ tfinite t) ∨
    (∃ steps. fst (shd steps) = (A, z) ∧ epath steps ∧ Saturated steps)›
    (is ‹?A ∨ ?B›)
proof −
  { assume ‹¬ ?A›
    with assms have ‹¬ tfinite (mkTree rules (A, z))›
      unfolding secavProver-def using wf-mkTree fair-rules by simp
    then obtain steps where ‹ipath (mkTree rules (A, z)) steps› using Konig by
blast
    with assms have ‹fst (shd steps) = (A, z) ∧ epath steps ∧ Saturated steps›
     by (metis UNIV-I fair-rules ipath.cases ipath-mkTree-Saturated mkTree.simps(1)
prod.sel(1)
         wf-ipath-epath wf-mkTree)
    then have ?B by blast
  }
  then show ?thesis by blast
qed
```

This is an abbreviation for validity under our bounded semantics (for well-formed interpretations).

```
abbreviation
  ‹uvalid z ≡ ∀ u (e :: nat ⇒ tm) f g. is-env u e ⟶ is-fdenot u f ⟶
    (∃ p ∈ set z. usemantics u e f g p)›
```

The sequent in the first state of a saturated escape path is not valid. This follows from our results in the theories EPathHintikka and Countermodel.

```
lemma epath-countermodel:
  assumes ‹fst (shd steps) = (A, z)› and ‹epath steps› and ‹Saturated steps›
  shows ‹¬ uvalid z›
proof
  assume ‹uvalid z›
  moreover have ‹Hintikka (tree-fms steps)› (is ‹Hintikka ?S›)
```

```
    using assms escape-path-Hintikka assms by simp
  moreover have ‹∀ p ∈ set z. p ∈ tree-fms steps›
    using assms shd-sset by (metis Pair-inject prod.collapse pseq-def pseq-in-tree-fms)
  then have ‹∃ g. ∀ p ∈ set z. ¬ usemantics (terms ?S) (E ?S) (F ?S) g p›
    using calculation(2) Hintikka-counter-model assms by blast
  moreover have ‹is-env (terms ?S) (E ?S)› ‹is-fdenot (terms ?S) (F ?S)›
    using is-env-E is-fdenot-F by blast+
  ultimately show False
    by blast
qed
```

Combining the results above, we can prove completeness with regards to our bounded semantics: if a sequent is valid under our bounded semantics, the prover will produce a finite, well-formed proof tree with the sequent at its root.

```
theorem prover-completeness-usemantics:
  fixes A :: ‹tm list›
  assumes ‹uvalid z›
  defines ‹t ≡ secavProver (A, z)›
  shows ‹fst (root t) = (A, z) ∧ wf t ∧ tfinite t›
  using assms epath-prover-completeness epath-countermodel by blast
```

Since our bounded semantics are sound, we can derive our main completeness theorem as a corollary: if a sequent is provable in the SeCaV proof system, the prover will produce a finite, well-formed proof tree with the sequent at its root.

```
corollary prover-completeness-SeCaV:
  fixes A :: ‹tm list›
  assumes ‹⊩ z›
  defines ‹t ≡ secavProver (A, z)›
  shows ‹fst (root t) = (A, z) ∧ wf t ∧ tfinite t›
proof –
  have ‹uvalid z›
    using assms sound-usemantics by blast
  then show ?thesis
    using assms prover-completeness-usemantics by blast
qed

end
```

2.10 Results

theory *Results* **imports** *Soundness Completeness Sequent-Calculus-Verifier* **begin**

In this theory, we collect our soundness and completeness results and prove some extra results linking the SeCaV proof system, the usual semantics of SeCaV, and our bounded semantics.

2.10.1 Alternate semantics

The existence of a finite, well-formed proof tree with a formula at its root implies that the formula is valid under our bounded semantics.

corollary *prover-soundness-usemantics*:
assumes ‹*tfinite t*› ‹*wf t*› ‹*is-env u e*› ‹*is-fdenot u f*›
shows ‹$\exists\, p \in$ *set* (*rootSequent t*). *usemantics u e f g p*›
using *assms prover-soundness-SeCaV sound-usemantics* **by** *blast*

The prover returns a finite, well-formed proof tree if and only if the sequent to be proved is valid under our bounded semantics.

theorem *prover-usemantics*:
fixes A :: ‹*tm list*› **and** z :: ‹*fm list*›
defines ‹$t \equiv$ *secavProver* (A, z)›
shows ‹*tfinite t* $\wedge$ *wf t* $\longleftrightarrow$ *uvalid z*›
using *assms prover-soundness-usemantics prover-completeness-usemantics*
unfolding *secavProver-def* **by** *fastforce*

The prover returns a finite, well-formed proof tree for a single formula if and only if the formula is valid under our bounded semantics.

corollary
fixes p :: *fm*
defines ‹$t \equiv$ *secavProver* ([], [p])›
shows ‹*tfinite t* $\wedge$ *wf t* $\longleftrightarrow$ *uvalid* [p]›
using *assms prover-usemantics* **by** *simp*

2.10.2 SeCaV

The prover returns a finite, well-formed proof tree if and only if the sequent to be proven is provable in the SeCaV proof system.

theorem *prover-SeCaV*:
fixes A :: ‹*tm list*› **and** z :: ‹*fm list*›
defines ‹$t \equiv$ *secavProver* (A, z)›
shows ‹*tfinite t* $\wedge$ *wf t* $\longleftrightarrow$ ($\Vdash z$)›
using *assms prover-soundness-SeCaV prover-completeness-SeCaV*
unfolding *secavProver-def* **by** *fastforce*

The prover returns a finite, well-formed proof tree if and only if the single formula to be proven is provable in the SeCaV proof system.

corollary
fixes p :: *fm*
defines ‹$t \equiv$ *secavProver* ([], [p])›
shows ‹*tfinite t* $\wedge$ *wf t* $\longleftrightarrow$ ($\Vdash$ [p])›
using *assms prover-SeCaV* **by** *blast*

2.10.3 Semantics

If the prover returns a finite, well-formed proof tree, some formula in the sequent at the root of the tree is valid under the usual SeCaV semantics.

corollary *prover-soundness-semantics*:
  assumes ‹*tfinite t*› ‹*wf t*›
  shows ‹$\exists\, p \in$ *set* (*rootSequent t*). *semantics e f g p*›
  using *assms prover-soundness-SeCaV sound* **by** *blast*

If the prover returns a finite, well-formed proof tree, the single formula in the sequent at the root of the tree is valid under the usual SeCaV semantics.

corollary
  assumes ‹*tfinite t*› ‹*wf t*› ‹*snd* (*fst* (*root t*)) = [*p*]›
  shows ‹*semantics e f g p*›
  using *assms prover-soundness-SeCaV complete-sound*(*2*) **by** *metis*

If a formula is valid under the usual SeCaV semantics, the prover will return a finite, well-formed proof tree with the formula at its root when called on it.

corollary *prover-completeness-semantics*:
  fixes *A* :: ‹*tm list*›
  assumes ‹$\forall$ (*e* :: *nat* $\Rightarrow$ *nat hterm*) *f g*. *semantics e f g p*›
  defines ‹*t* $\equiv$ *secavProver* (*A*, [*p*])›
  shows ‹*fst* (*root t*) = (*A*, [*p*]) $\wedge$ *wf t* $\wedge$ *tfinite t*›
proof −
  have ‹$\Vdash$ [*p*]›
    using *assms complete-sound*(*1*) **by** *blast*
  then show *?thesis*
    using *assms prover-completeness-SeCaV* **by** *blast*
qed

The prover produces a finite, well-formed proof tree for a formula if and only if that formula is valid under the usual SeCaV semantics.

theorem *prover-semantics*:
  fixes *A* :: ‹*tm list*› **and** *p* :: *fm*
  defines ‹*t* $\equiv$ *secavProver* (*A*, [*p*])›
  shows ‹*tfinite t* $\wedge$ *wf t* $\longleftrightarrow$ ($\forall$ (*e* :: *nat* $\Rightarrow$ *nat hterm*) *f g*. *semantics e f g p*)›
  using *assms prover-soundness-semantics prover-completeness-semantics*
  unfolding *secavProver-def* **by** *fastforce*

Validity in the two semantics (in the proper universes) coincide.

theorem *semantics-usemantics*:
  ‹($\forall$ (*e* :: *nat* $\Rightarrow$ *nat hterm*) *f g*. *semantics e f g p*) $\longleftrightarrow$
   ($\forall$ (*u* :: *tm set*) *e f g*. *is-env u e* $\longrightarrow$ *is-fdenot u f* $\longrightarrow$ *usemantics u e f g p*)›
  using *prover-semantics prover-usemantics* **by** *simp*

end