

# A Sequent Calculus Prover for First-Order Logic with Functions

Asta Halkjær From      Frederik Krogsdal Jacobsen

February 2, 2022

## **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

<b>1</b>	<b>SeCaV</b>	<b>2</b>
1.1	Sequent Calculus Verifier (SeCaV)	2
1.2	Syntax: Terms / Formulas	2
1.3	Semantics: Terms / Formulas	2
1.4	Auxiliary Functions	3
1.5	Sequent Calculus	4
1.6	Shorthands	4
1.7	Appendix: Soundness	5
1.7.1	Increment Function	5
1.7.2	Parameters: Terms	6
1.7.3	Parameters: Formulas	6
1.7.4	Update Lemmas	7
1.7.5	Substitution	7
1.7.6	Auxiliary Lemmas	8
1.7.7	Soundness	8
1.8	Reference	9
1.9	Appendix: Completeness	9
1.10	Reference	11
<b>2</b>	<b>The prover</b>	<b>12</b>
2.1	Proof search procedure	12
2.1.1	Datatypes	12
2.1.2	Auxiliary functions	12
2.1.3	Effects of rules	14
2.1.4	The rule stream	15
2.1.5	Abstract completeness	15
2.2	Export	16
2.3	Lemmas about the prover	17
2.3.1	SeCaV lemmas	17
2.3.2	Fairness	18
2.3.3	Substitution	21
2.3.4	Custom cases	23
2.3.5	Unaffected formulas	25

2.3.6	Affected formulas . . . . .	26
2.3.7	Generating new function names . . . . .	28
2.3.8	Finding axioms . . . . .	28
2.3.9	Subterms . . . . .	28
2.4	Hintikka sets for SeCaV . . . . .	29
2.5	Escape path formulas are Hintikka . . . . .	30
2.5.1	Definitions . . . . .	30
2.5.2	Facts about streams . . . . .	31
2.5.3	Transformation of states on an escape path . . . . .	31
2.5.4	Preservation of formulas on escape paths . . . . .	32
2.5.5	Formulas on an escape path form a Hintikka set . . . . .	33
2.6	Bounded semantics . . . . .	48
2.7	Countermodels from Hintikka sets . . . . .	50
2.8	Soundness . . . . .	56
2.9	Completeness . . . . .	63
2.10	Results . . . . .	64
2.10.1	Alternate semantics . . . . .	65
2.10.2	SeCaV . . . . .	65
2.10.3	Semantics . . . . .	66

# 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**  $\langle \text{shift } e \ v \ x \equiv \lambda n. \text{ if } n < v \text{ then } e \ n \text{ else if } n = v \text{ then } x \text{ else } e \ (n - 1) \rangle$

**primrec** *semantics-term* and *semantics-list* where

$\langle \text{semantics-term } e \ f \ (\text{Var } n) = e \ n \rangle$  |  
 $\langle \text{semantics-term } e \ f \ (\text{Fun } i \ l) = f \ i \ (\text{semantics-list } e \ f \ l) \rangle$  |  
 $\langle \text{semantics-list } e \ f \ [] = [] \rangle$  |  
 $\langle \text{semantics-list } e \ f \ (t \ # \ l) = \text{semantics-term } e \ f \ t \ # \ \text{semantics-list } e \ f \ l \rangle$

**primrec** *semantics* where

$\langle \text{semantics } e \ f \ g \ (\text{Pre } i \ l) = g \ i \ (\text{semantics-list } e \ f \ l) \rangle$  |  
 $\langle \text{semantics } e \ f \ g \ (\text{Imp } p \ q) = (\text{semantics } e \ f \ g \ p \longrightarrow \text{semantics } e \ f \ g \ q) \rangle$  |  
 $\langle \text{semantics } e \ f \ g \ (\text{Dis } p \ q) = (\text{semantics } e \ f \ g \ p \vee \text{semantics } e \ f \ g \ q) \rangle$  |  
 $\langle \text{semantics } e \ f \ g \ (\text{Con } p \ q) = (\text{semantics } e \ f \ g \ p \wedge \text{semantics } e \ f \ g \ q) \rangle$  |  
 $\langle \text{semantics } e \ f \ g \ (\text{Exi } p) = (\exists x. \text{semantics } (\text{shift } e \ 0 \ x) \ f \ g \ p) \rangle$  |  
 $\langle \text{semantics } e \ f \ g \ (\text{Uni } p) = (\forall x. \text{semantics } (\text{shift } e \ 0 \ x) \ f \ g \ p) \rangle$  |  
 $\langle \text{semantics } e \ f \ g \ (\text{Neg } p) = (\neg \text{semantics } e \ f \ g \ p) \rangle$

— Test

**corollary**  $\langle \text{semantics } e \ f \ g \ (\text{Imp } (\text{Pre } 0 \ [])) \ (\text{Pre } 0 \ []) \rangle$

by *simp*

**lemma**  $\langle \neg \text{ semantics } e \text{ f g } (\text{Neg } (\text{Imp } (\text{Pre } 0 \ [])) (\text{Pre } 0 \ [])) \rangle$   
by *simp*

## 1.4 Auxiliary Functions

**primrec** *new-term* and *new-list* where

$\langle \text{new-term } c \text{ (Var } n) = \text{True} \rangle |$   
 $\langle \text{new-term } c \text{ (Fun } i \ l) = (\text{if } i = c \text{ then False else new-list } c \ l) \rangle |$   
 $\langle \text{new-list } c \ [] = \text{True} \rangle |$   
 $\langle \text{new-list } c \ (t \ # \ l) = (\text{if new-term } c \ t \text{ then new-list } c \ l \text{ else False}) \rangle$

**primrec** *new* where

$\langle \text{new } c \text{ (Pre } i \ l) = \text{new-list } c \ l \rangle |$   
 $\langle \text{new } c \text{ (Imp } p \ q) = (\text{if new } c \ p \text{ then new } c \ q \text{ else False}) \rangle |$   
 $\langle \text{new } c \text{ (Dis } p \ q) = (\text{if new } c \ p \text{ then new } c \ q \text{ else False}) \rangle |$   
 $\langle \text{new } c \text{ (Con } p \ q) = (\text{if new } c \ p \text{ then new } c \ q \text{ else False}) \rangle |$   
 $\langle \text{new } c \text{ (Exi } p) = \text{new } c \ p \rangle |$   
 $\langle \text{new } c \text{ (Uni } p) = \text{new } c \ p \rangle |$   
 $\langle \text{new } c \text{ (Neg } p) = \text{new } c \ p \rangle$

**primrec** *news* where

$\langle \text{news } c \ [] = \text{True} \rangle |$   
 $\langle \text{news } c \ (p \ # \ z) = (\text{if new } c \ p \text{ then news } c \ z \text{ else False}) \rangle$

**primrec** *inc-term* and *inc-list* where

$\langle \text{inc-term } (\text{Var } n) = \text{Var } (n + 1) \rangle |$   
 $\langle \text{inc-term } (\text{Fun } i \ l) = \text{Fun } i \ (\text{inc-list } l) \rangle |$   
 $\langle \text{inc-list } [] = [] \rangle |$   
 $\langle \text{inc-list } (t \ # \ l) = \text{inc-term } t \ # \ \text{inc-list } l \rangle$

**primrec** *sub-term* and *sub-list* where

$\langle \text{sub-term } v \ s \ (\text{Var } n) = (\text{if } n < v \text{ then Var } n \text{ else if } n = v \text{ then } s \text{ else Var } (n - 1)) \rangle |$   
 $\langle \text{sub-term } v \ s \ (\text{Fun } i \ l) = \text{Fun } i \ (\text{sub-list } v \ s \ l) \rangle |$   
 $\langle \text{sub-list } v \ s \ [] = [] \rangle |$   
 $\langle \text{sub-list } v \ s \ (t \ # \ l) = \text{sub-term } v \ s \ t \ # \ \text{sub-list } v \ s \ l \rangle$

**primrec** *sub* where

$\langle \text{sub } v \ s \ (\text{Pre } i \ l) = \text{Pre } i \ (\text{sub-list } v \ s \ l) \rangle |$   
 $\langle \text{sub } v \ s \ (\text{Imp } p \ q) = \text{Imp } (\text{sub } v \ s \ p) \ (\text{sub } v \ s \ q) \rangle |$   
 $\langle \text{sub } v \ s \ (\text{Dis } p \ q) = \text{Dis } (\text{sub } v \ s \ p) \ (\text{sub } v \ s \ q) \rangle |$   
 $\langle \text{sub } v \ s \ (\text{Con } p \ q) = \text{Con } (\text{sub } v \ s \ p) \ (\text{sub } v \ s \ q) \rangle |$   
 $\langle \text{sub } v \ s \ (\text{Exi } p) = \text{Exi } (\text{sub } (v + 1) \ (\text{inc-term } s) \ p) \rangle |$   
 $\langle \text{sub } v \ s \ (\text{Uni } p) = \text{Uni } (\text{sub } (v + 1) \ (\text{inc-term } s) \ p) \rangle |$   
 $\langle \text{sub } v \ s \ (\text{Neg } p) = \text{Neg } (\text{sub } v \ s \ p) \rangle$

**primrec** *member* where

$\langle \text{member } p [] = \text{False} \rangle \mid$   
 $\langle \text{member } p (q \# z) = (\text{if } p = q \text{ then True else member } p z) \rangle$

**primrec** *ext where*

$\langle \text{ext } y [] = \text{True} \rangle \mid$   
 $\langle \text{ext } y (p \# z) = (\text{if member } p y \text{ then ext } y z \text{ else False}) \rangle$

— Simplifications

**lemma** *member [iff]*:  $\langle \text{member } p z \longleftrightarrow p \in \text{set } z \rangle$   
**by** (*induct z*) *simp-all*

**lemma** *ext [iff]*:  $\langle \text{ext } y z \longleftrightarrow \text{set } z \subseteq \text{set } y \rangle$   
**by** (*induct z*) *simp-all*

## 1.5 Sequent Calculus

**inductive** *sequent-calculus* ( $\langle \vdash \rightarrow 0 \rangle$ ) **where**

$\langle \vdash p \# z \text{ if } \langle \text{member } (\text{Neg } p) z \rangle \mid$   
 $\langle \vdash \text{Dis } p q \# z \text{ if } \langle \vdash p \# q \# z \rangle \mid$   
 $\langle \vdash \text{Imp } p q \# z \text{ if } \langle \vdash \text{Neg } p \# q \# z \rangle \mid$   
 $\langle \vdash \text{Neg } (\text{Con } p q) \# z \text{ if } \langle \vdash \text{Neg } p \# \text{Neg } q \# z \rangle \mid$   
 $\langle \vdash \text{Con } p q \# z \text{ if } \langle \vdash p \# z \rangle \text{ and } \langle \vdash q \# z \rangle \mid$   
 $\langle \vdash \text{Neg } (\text{Imp } p q) \# z \text{ if } \langle \vdash p \# z \rangle \text{ and } \langle \vdash \text{Neg } q \# z \rangle \mid$   
 $\langle \vdash \text{Neg } (\text{Dis } p q) \# z \text{ if } \langle \vdash \text{Neg } p \# z \rangle \text{ and } \langle \vdash \text{Neg } q \# z \rangle \mid$   
 $\langle \vdash \text{Exi } p \# z \text{ if } \langle \vdash \text{sub } 0 t p \# z \rangle \mid$   
 $\langle \vdash \text{Neg } (\text{Uni } p) \# z \text{ if } \langle \vdash \text{Neg } (\text{sub } 0 t p) \# z \rangle \mid$   
 $\langle \vdash \text{Uni } p \# z \text{ if } \langle \vdash \text{sub } 0 (\text{Fun } i []) p \# z \rangle \text{ and } \langle \text{news } i (p \# z) \rangle \mid$   
 $\langle \vdash \text{Neg } (\text{Exi } p) \# z \text{ if } \langle \vdash \text{Neg } (\text{sub } 0 (\text{Fun } i []) p) \# z \rangle \text{ and } \langle \text{news } i (p \# z) \rangle \mid$   
 $\langle \vdash \text{Neg } (\text{Neg } p) \# z \text{ if } \langle \vdash p \# z \rangle \mid$   
 $\langle \vdash y \rangle \text{ if } \langle \vdash z \rangle \text{ and } \langle \text{ext } y z \rangle$

— Test

**corollary**  $\langle \vdash [\text{Imp } (\text{Pre } 0 []) (\text{Pre } 0 [])] \rangle$   
**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**  $\text{GammaExi} = \text{sequent-calculus.intros}(8)$

**lemmas**  $\text{GammaUni} = \text{sequent-calculus.intros}(9)$

**lemmas**  $\text{DeltaUni} = \text{sequent-calculus.intros}(10)$

**lemmas**  $\text{DeltaExi} = \text{sequent-calculus.intros}(11)$

**lemmas**  $\text{Neg} = \text{sequent-calculus.intros}(12)$

**lemmas**  $\text{Ext} = \text{sequent-calculus.intros}(13)$

— Test

**lemma**  $\langle \vdash$

[  
   $\text{Imp} (\text{Pre } 0 \ \square) (\text{Pre } 0 \ \square)$   
]

$\rangle$

**proof** —

**from**  $\text{AlphaImp}$  **have**  $?thesis$  **if**  $\langle \vdash$

[  
   $\text{Neg} (\text{Pre } 0 \ \square),$   
   $\text{Pre } 0 \ \square$   
]

$\rangle$

**using**  $that$  **by**  $simp$

**with**  $\text{Ext}$  **have**  $?thesis$  **if**  $\langle \vdash$

[  
   $\text{Pre } 0 \ \square,$   
   $\text{Neg} (\text{Pre } 0 \ \square)$   
]

$\rangle$

**using**  $that$  **by**  $simp$

**with**  $\text{Basic}$  **show**  $?thesis$

**by**  $simp$

**qed**

## 1.7 Appendix: Soundness

### 1.7.1 Increment Function

**primrec**  $\text{liftt} :: \langle tm \Rightarrow tm \rangle$  **and**  $\text{liftts} :: \langle tm \text{ list} \Rightarrow tm \text{ list} \rangle$  **where**

$\langle \text{liftt} (\text{Var } i) = \text{Var} (\text{Suc } i) \rangle$  |

$\langle \text{liftt} (\text{Fun } a \ ts) = \text{Fun } a \ (\text{liftts } ts) \rangle$  |

$\langle \text{liftts } \square = \square \rangle$  |

$\langle \text{liftts} (t \ # \ ts) = \text{liftt } t \ # \ \text{liftts } ts \rangle$

### 1.7.2 Parameters: Terms

**primrec**  $\text{paramst} :: \langle tm \Rightarrow nat\ set \rangle$  and  $\text{paramsts} :: \langle tm\ list \Rightarrow nat\ set \rangle$  **where**

$\langle \text{paramst}\ (Var\ n) = \{\} \rangle \mid$   
 $\langle \text{paramst}\ (Fun\ a\ ts) = \{a\} \cup \text{paramsts}\ ts \rangle \mid$   
 $\langle \text{paramsts}\ [] = \{\} \rangle \mid$   
 $\langle \text{paramsts}\ (t\ \# \ ts) = (\text{paramst}\ t \cup \text{paramsts}\ ts) \rangle$

**lemma**  $p0$  [simp]:  $\langle \text{paramsts}\ ts = \bigcup (\text{set}\ (\text{map}\ \text{paramst}\ ts)) \rangle$   
**by** (induct ts) simp-all

**primrec**  $\text{paramst}' :: \langle tm \Rightarrow nat\ set \rangle$  **where**

$\langle \text{paramst}'\ (Var\ n) = \{\} \rangle \mid$   
 $\langle \text{paramst}'\ (Fun\ a\ ts) = \{a\} \cup \bigcup (\text{set}\ (\text{map}\ \text{paramst}'\ ts)) \rangle$

**lemma**  $p1$  [simp]:  $\langle \text{paramst}'\ t = \text{paramst}\ t \rangle$   
**by** (induct t) simp-all

### 1.7.3 Parameters: Formulas

**primrec**  $\text{params} :: \langle fm \Rightarrow nat\ set \rangle$  **where**

$\langle \text{params}\ (Pre\ b\ ts) = \text{paramsts}\ ts \rangle \mid$   
 $\langle \text{params}\ (Imp\ p\ q) = \text{params}\ p \cup \text{params}\ q \rangle \mid$   
 $\langle \text{params}\ (Dis\ p\ q) = \text{params}\ p \cup \text{params}\ q \rangle \mid$   
 $\langle \text{params}\ (Con\ p\ q) = \text{params}\ p \cup \text{params}\ q \rangle \mid$   
 $\langle \text{params}\ (Exi\ p) = \text{params}\ p \rangle \mid$   
 $\langle \text{params}\ (Uni\ p) = \text{params}\ p \rangle \mid$   
 $\langle \text{params}\ (Neg\ p) = \text{params}\ p \rangle$

**primrec**  $\text{params}' :: \langle fm \Rightarrow nat\ set \rangle$  **where**

$\langle \text{params}'\ (Pre\ b\ ts) = \bigcup (\text{set}\ (\text{map}\ \text{paramst}'\ ts)) \rangle \mid$   
 $\langle \text{params}'\ (Imp\ p\ q) = \text{params}'\ p \cup \text{params}'\ q \rangle \mid$   
 $\langle \text{params}'\ (Dis\ p\ q) = \text{params}'\ p \cup \text{params}'\ q \rangle \mid$   
 $\langle \text{params}'\ (Con\ p\ q) = \text{params}'\ p \cup \text{params}'\ q \rangle \mid$   
 $\langle \text{params}'\ (Exi\ p) = \text{params}'\ p \rangle \mid$   
 $\langle \text{params}'\ (Uni\ p) = \text{params}'\ p \rangle \mid$   
 $\langle \text{params}'\ (Neg\ p) = \text{params}'\ p \rangle$

**lemma**  $p2$  [simp]:  $\langle \text{params}'\ p = \text{params}\ p \rangle$   
**by** (induct p) simp-all

**fun**  $\text{paramst}'' :: \langle tm \Rightarrow nat\ set \rangle$  **where**

$\langle \text{paramst}''\ (Var\ n) = \{\} \rangle \mid$   
 $\langle \text{paramst}''\ (Fun\ a\ ts) = \{a\} \cup (\bigcup t \in \text{set}\ ts.\ \text{paramst}''\ t) \rangle$

**lemma**  $p1'$  [simp]:  $\langle \text{paramst}''\ t = \text{paramst}\ t \rangle$   
**by** (induct t) simp-all

**fun**  $\text{params}'' :: \langle fm \Rightarrow nat\ set \rangle$  **where**

$\langle \text{params}''\ (Pre\ b\ ts) = (\bigcup t \in \text{set}\ ts.\ \text{paramst}''\ t) \rangle \mid$

$\langle \text{params}'' (\text{Imp } p \ q) = \text{params}'' p \cup \text{params}'' q \rangle \mid$   
 $\langle \text{params}'' (\text{Dis } p \ q) = \text{params}'' p \cup \text{params}'' q \rangle \mid$   
 $\langle \text{params}'' (\text{Con } p \ q) = \text{params}'' p \cup \text{params}'' q \rangle \mid$   
 $\langle \text{params}'' (\text{Exi } p) = \text{params}'' p \rangle \mid$   
 $\langle \text{params}'' (\text{Uni } p) = \text{params}'' p \rangle \mid$   
 $\langle \text{params}'' (\text{Neg } p) = \text{params}'' p \rangle$

**lemma**  $p2'$  [simp]:  $\langle \text{params}'' p = \text{params } p \rangle$   
**by** (induct p) simp-all

### 1.7.4 Update Lemmas

**lemma**  $\text{upd-lemma}'$  [simp]:  
 $\langle n \notin \text{paramst } t \implies \text{semantics-term } e (f(n := z)) \ t = \text{semantics-term } e \ f \ t \rangle$   
 $\langle n \notin \text{paramsts } ts \implies \text{semantics-list } e (f(n := z)) \ ts = \text{semantics-list } e \ f \ ts \rangle$   
**by** (induct t and ts rule: paramst.induct paramsts.induct) auto

**lemma**  $\text{upd-lemma}$  [iff]:  $\langle n \notin \text{params } p \implies \text{semantics } e (f(n := z)) \ g \ p \longleftrightarrow \text{semantics } e \ f \ g \ p \rangle$   
**by** (induct p arbitrary: e) simp-all

### 1.7.5 Substitution

**primrec**  $\text{subst} :: \langle \text{tm} \Rightarrow \text{tm} \Rightarrow \text{nat} \Rightarrow \text{tm} \rangle$  and  $\text{substts} :: \langle \text{tm list} \Rightarrow \text{tm} \Rightarrow \text{nat} \Rightarrow \text{tm list} \rangle$  **where**  
 $\langle \text{subst } (\text{Var } i) \ s \ k = (\text{if } k < i \text{ then } \text{Var } (i - 1) \text{ else if } i = k \text{ then } s \text{ else } \text{Var } i) \rangle \mid$   
 $\langle \text{subst } (\text{Fun } a \ ts) \ s \ k = \text{Fun } a \ (\text{substts } ts \ s \ k) \rangle \mid$   
 $\langle \text{substts } [] \ s \ k = [] \rangle \mid$   
 $\langle \text{substts } (t \ # \ ts) \ s \ k = \text{subst } t \ s \ k \ # \ \text{substts } ts \ s \ k \rangle$

**primrec**  $\text{subst} :: \langle \text{fm} \Rightarrow \text{tm} \Rightarrow \text{nat} \Rightarrow \text{fm} \rangle$  **where**  
 $\langle \text{subst } (\text{Pre } b \ ts) \ s \ k = \text{Pre } b \ (\text{substts } ts \ s \ k) \rangle \mid$   
 $\langle \text{subst } (\text{Imp } p \ q) \ s \ k = \text{Imp } (\text{subst } p \ s \ k) \ (\text{subst } q \ s \ k) \rangle \mid$   
 $\langle \text{subst } (\text{Dis } p \ q) \ s \ k = \text{Dis } (\text{subst } p \ s \ k) \ (\text{subst } q \ s \ k) \rangle \mid$   
 $\langle \text{subst } (\text{Con } p \ q) \ s \ k = \text{Con } (\text{subst } p \ s \ k) \ (\text{subst } q \ s \ k) \rangle \mid$   
 $\langle \text{subst } (\text{Exi } p) \ s \ k = \text{Exi } (\text{subst } p \ (\text{liftt } s) \ (\text{Suc } k)) \rangle \mid$   
 $\langle \text{subst } (\text{Uni } p) \ s \ k = \text{Uni } (\text{subst } p \ (\text{liftt } s) \ (\text{Suc } k)) \rangle \mid$   
 $\langle \text{subst } (\text{Neg } p) \ s \ k = \text{Neg } (\text{subst } p \ s \ k) \rangle$

**lemma**  $\text{shift-eq}$  [simp]:  $\langle i = j \implies (\text{shift } e \ i \ T) \ j = T \rangle$  **for**  $i :: \text{nat}$   
**unfolding**  $\text{shift-def}$  **by** simp

**lemma**  $\text{shift-gt}$  [simp]:  $\langle j < i \implies (\text{shift } e \ i \ T) \ j = e \ j \rangle$  **for**  $i :: \text{nat}$   
**unfolding**  $\text{shift-def}$  **by** simp

**lemma**  $\text{shift-lt}$  [simp]:  $\langle i < j \implies (\text{shift } e \ i \ T) \ j = e \ (j - 1) \rangle$  **for**  $i :: \text{nat}$   
**unfolding**  $\text{shift-def}$  **by** simp

**lemma**  $\text{shift-commute}$  [simp]:  $\langle \text{shift } (\text{shift } e \ i \ U) \ 0 \ T = \text{shift } (\text{shift } e \ 0 \ T) \ (\text{Suc } i) \ U \rangle$

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

**lemma** *subst-lemma'* [*simp*]:

$\langle \text{semantics-term } e f (\text{subst } t u i) = \text{semantics-term } (\text{shift } e i (\text{semantics-term } e f u)) f t \rangle$

$\langle \text{semantics-list } e f (\text{substts } ts u i) = \text{semantics-list } (\text{shift } e i (\text{semantics-term } e f u)) f ts \rangle$

**by** (*induct t and ts rule: subst.induct substts.induct*) *simp-all*

**lemma** *lift-lemma* [*simp*]:

$\langle \text{semantics-term } (\text{shift } e 0 x) f (\text{liftt } t) = \text{semantics-term } e f t \rangle$

$\langle \text{semantics-list } (\text{shift } e 0 x) f (\text{liftts } ts) = \text{semantics-list } e f ts \rangle$

**by** (*induct t and ts rule: liftt.induct liftts.induct*) *simp-all*

**lemma** *subst-lemma* [*iff*]:

$\langle \text{semantics } e f g (\text{subst } a t i) \longleftrightarrow \text{semantics } (\text{shift } e i (\text{semantics-term } e f t)) f g a \rangle$

**by** (*induct a arbitrary: e i t*) *simp-all*

### 1.7.6 Auxiliary Lemmas

**lemma** *s1* [*iff*]:  $\langle \text{new-term } c t \longleftrightarrow (c \notin \text{paramst } t) \langle \text{new-list } c l \longleftrightarrow (c \notin \text{paramsts } l) \rangle \rangle$

**by** (*induct t and l rule: new-term.induct new-list.induct*) *simp-all*

**lemma** *s2* [*iff*]:  $\langle \text{new } c p \longleftrightarrow (c \notin \text{params } p) \rangle$

**by** (*induct p*) *simp-all*

**lemma** *s3* [*iff*]:  $\langle \text{news } c z \longleftrightarrow \text{list-all } (\lambda p. c \notin \text{params } p) z \rangle$

**by** (*induct z*) *simp-all*

**lemma** *s4* [*simp*]:  $\langle \text{inc-term } t = \text{liftt } t \rangle \langle \text{inc-list } l = \text{liftts } l \rangle$

**by** (*induct t and l rule: inc-term.induct inc-list.induct*) *simp-all*

**lemma** *s5* [*simp*]:  $\langle \text{sub-term } v s t = \text{subst } t s v \rangle \langle \text{sub-list } v s l = \text{substts } l s v \rangle$

**by** (*induct t and l rule: inc-term.induct inc-list.induct*) *simp-all*

**lemma** *s6* [*simp*]:  $\langle \text{sub } v s p = \text{subst } p s v \rangle$

**by** (*induct p arbitrary: v s*) *simp-all*

### 1.7.7 Soundness

**theorem** *sound*:  $\langle \Vdash z \implies \exists p \in \text{set } z. \text{semantics } e f g p \rangle$

**proof** (*induct arbitrary: f rule: sequent-calculus.induct*)

**case** (*10 i p z*)

**then show** *?case*

**proof** (*cases*  $\langle \forall x. \text{semantics } e (f(i := \lambda-. x)) g (\text{sub } 0 (\text{Fun } i [] p)) \rangle$ )

**case** *False*

**moreover have**  $\langle \text{list-all } (\lambda p. i \notin \text{params } p) z \rangle$

**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 ⟨∀ x. semantics e (f(i := λ-. x)) g (Neg (sub 0 (Fun i [])) p)⟩)
    case False
    moreover have ⟨list-all (λp. i ∉ 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 ⇒ ∃ p. member p z ∧ semantics e f g p
  using sound by force

```

```

corollary † [p] ⇒ 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

```

$\langle \text{from-fm } (\text{Pre } b \ ts) = \text{Pred } b \ (\text{from-tm-list } ts) \rangle \mid$   
 $\langle \text{from-fm } (\text{Con } p \ q) = \text{And } (\text{from-fm } p) \ (\text{from-fm } q) \rangle \mid$   
 $\langle \text{from-fm } (\text{Dis } p \ q) = \text{Or } (\text{from-fm } p) \ (\text{from-fm } q) \rangle \mid$   
 $\langle \text{from-fm } (\text{Imp } p \ q) = \text{Impl } (\text{from-fm } p) \ (\text{from-fm } q) \rangle \mid$   
 $\langle \text{from-fm } (\text{Neg } p) = \text{FOL-Fitting.Neg } (\text{from-fm } p) \rangle \mid$   
 $\langle \text{from-fm } (\text{Uni } p) = \text{Forall } (\text{from-fm } p) \rangle \mid$   
 $\langle \text{from-fm } (\text{Exi } p) = \text{Exists } (\text{from-fm } p) \rangle$

**primrec** *to-tm* and *to-tm-list* where

$\langle \text{to-tm } (\text{FOL-Fitting.Var } n) = \text{Var } n \rangle \mid$   
 $\langle \text{to-tm } (\text{App } a \ ts) = \text{Fun } a \ (\text{to-tm-list } ts) \rangle \mid$   
 $\langle \text{to-tm-list } [] = [] \rangle \mid$   
 $\langle \text{to-tm-list } (t \ # \ ts) = \text{to-tm } t \ # \ \text{to-tm-list } ts \rangle$

**primrec** *to-fm* where

$\langle \text{to-fm } \perp = \text{Neg } (\text{Imp } (\text{Pre } 0 \ []) \ (\text{Pre } 0 \ [])) \rangle \mid$   
 $\langle \text{to-fm } \top = \text{Imp } (\text{Pre } 0 \ []) \ (\text{Pre } 0 \ []) \rangle \mid$   
 $\langle \text{to-fm } (\text{Pred } b \ ts) = \text{Pre } b \ (\text{to-tm-list } ts) \rangle \mid$   
 $\langle \text{to-fm } (\text{And } p \ q) = \text{Con } (\text{to-fm } p) \ (\text{to-fm } q) \rangle \mid$   
 $\langle \text{to-fm } (\text{Or } p \ q) = \text{Dis } (\text{to-fm } p) \ (\text{to-fm } q) \rangle \mid$   
 $\langle \text{to-fm } (\text{Impl } p \ q) = \text{Imp } (\text{to-fm } p) \ (\text{to-fm } q) \rangle \mid$   
 $\langle \text{to-fm } (\text{FOL-Fitting.Neg } p) = \text{Neg } (\text{to-fm } p) \rangle \mid$   
 $\langle \text{to-fm } (\text{Forall } p) = \text{Uni } (\text{to-fm } p) \rangle \mid$   
 $\langle \text{to-fm } (\text{Exists } p) = \text{Exi } (\text{to-fm } p) \rangle$

**theorem** *to-from-tm [simp]*:  $\langle \text{to-tm } (\text{from-tm } t) = t \ \langle \text{to-tm-list } (\text{from-tm-list } ts) = ts \rangle$

**by** (*induct t and ts rule: from-tm.induct from-tm-list.induct*) *simp-all*

**theorem** *to-from-fm [simp]*:  $\langle \text{to-fm } (\text{from-fm } p) = p \rangle$

**by** (*induct p*) *simp-all*

**lemma** *Truth [simp]*:  $\langle \vdash \text{Imp } (\text{Pre } 0 \ []) \ (\text{Pre } 0 \ []) \ \# \ z \rangle$

**using** *AlphaImp Basic Ext ext.simps member.simps(2)* **by** *metis*

**lemma** *paramst [simp]*:

$\langle \text{FOL-Fitting.new-term } c \ t = \text{new-term } c \ (\text{to-tm } t) \rangle$

$\langle \text{FOL-Fitting.new-list } c \ l = \text{new-list } c \ (\text{to-tm-list } l) \rangle$

**by** (*induct t and l rule: FOL-Fitting.paramst.induct FOL-Fitting.paramsts.induct*) *simp-all*

**lemma** *params [iff]*:  $\langle \text{FOL-Fitting.new } c \ p \longleftrightarrow \text{new } c \ (\text{to-fm } p) \rangle$

**by** (*induct p*) *simp-all*

**lemma** *list-params [iff]*:  $\langle \text{FOL-Fitting.news } c \ z \longleftrightarrow \text{news } c \ (\text{map to-fm } z) \rangle$

**by** (*induct z*) *simp-all*

**lemma** *liftt [simp]*:

$\langle \text{to-tm } (\text{FOL-Fitting.liftt } t) = \text{inc-term } (\text{to-tm } t) \rangle$

$\langle \text{to-tm-list } (FOL\text{-Fitting.liftts } l) = \text{inc-list } (\text{to-tm-list } l) \rangle$   
**by** (induct  $t$  and  $l$  rule:  $FOL\text{-Fitting.liftt.induct } FOL\text{-Fitting.liftts.induct}$ ) *simp-all*

**lemma** *subst* [*simp*]:

$\langle \text{to-tm } (FOL\text{-Fitting.substt } t s v) = \text{sub-term } v (\text{to-tm } s) (\text{to-tm } t) \rangle$   
 $\langle \text{to-tm-list } (FOL\text{-Fitting.substts } l s v) = \text{sub-list } v (\text{to-tm } s) (\text{to-tm-list } l) \rangle$   
**by** (induct  $t$  and  $l$  rule:  $FOL\text{-Fitting.substt.induct } FOL\text{-Fitting.substts.induct}$ )  
*simp-all*

**lemma** *subst* [*simp*]:  $\langle \text{to-fm } (FOL\text{-Fitting.subst } A t s) = \text{sub } s (\text{to-tm } t) (\text{to-fm } A) \rangle$   
**by** (induct  $A$  arbitrary:  $t s$ ) *simp-all*

**lemma** *sim*:  $\langle (\vdash x) \implies (\Vdash (\text{map to-fm } x)) \rangle$   
**by** (induct rule:  $SC.induct$ ) (force intro:  $\text{sequent-calculus.intros}$ ) $+$

**lemma** *evalt* [*simp*]:

$\langle \text{semantics-term } e f t = \text{evalt } e f (\text{from-tm } t) \rangle$   
 $\langle \text{semantics-list } e f ts = \text{evalts } e f (\text{from-tm-list } ts) \rangle$   
**by** (induct  $t$  and  $ts$  rule:  $\text{from-tm.induct } \text{from-tm-list.induct}$ ) *simp-all*

**lemma** *shift* [*simp*]:  $\langle \text{shift } e 0 x = e(0:x) \rangle$   
**unfolding** *shift-def*  $FOL\text{-Fitting.shift-def}$  **by** *simp*

**lemma** *semantics* [*iff*]:  $\langle \text{semantics } e f g p \iff \text{eval } e f g (\text{from-fm } p) \rangle$   
**by** (induct  $p$  arbitrary:  $e$ ) *simp-all*

**abbreviation** *valid* ( $\gg - 0$ ) **where**

$\langle (\gg p) \equiv \forall (e :: - \Rightarrow \text{nat hterm}) f g. \text{semantics } e f g p \rangle$

**theorem** *complete-sound*:  $\langle (\gg p) \implies \Vdash [p] \rangle \langle \Vdash [q] \implies \text{semantics } e f g q \rangle$

**by** (*metis to-from-fm sim semantics list.map SC-completeness*) (use *sound* **in** *force*)

**corollary**  $\langle (\gg p) \iff (\Vdash [p]) \rangle$

**using** *complete-sound* **by** *fast*

## 1.10 Reference

Asta Halkjær From (2019): Sequent Calculus [https://www.isa-afp.org/entries/FOL\\_Seq\\_Calc1.html](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* ::  $\langle tm \Rightarrow nat\ list \rangle$  **and** *listFunTms* ::  $\langle tm\ list \Rightarrow nat\ list \rangle$  **where**  
 $\langle listFunTm\ (Fun\ n\ ts) = n\ \# \ listFunTms\ ts \rangle$   
 $| \langle listFunTm\ (Var\ n) = [] \rangle$   
 $| \langle listFunTms\ [] = [] \rangle$   
 $| \langle listFunTms\ (t\ \# \ ts) = listFunTm\ t\ @ \ listFunTms\ ts \rangle$

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

**definition** *generateNew* ::  $\langle tm\ list \Rightarrow nat \rangle$  **where**  
 $\langle generateNew\ ts \equiv 1 + foldr\ max\ (listFunTms\ ts)\ 0 \rangle$

subtermTm returns a list of all terms occurring within a term

**primrec** *subtermTm* ::  $\langle tm \Rightarrow tm\ list \rangle$  **where**  
 $\langle subtermTm\ (Fun\ n\ ts) = Fun\ n\ ts\ \# \ remdups\ (concat\ (map\ subtermTm\ ts)) \rangle$   
 $| \langle subtermTm\ (Var\ n) = [Var\ n] \rangle$

subtermFm returns a list of all terms occurring within a formula

**primrec** *subtermFm* ::  $\langle fm \Rightarrow tm\ list \rangle$  **where**  
 $\langle subtermFm\ (Pre\ -\ ts) = concat\ (map\ subtermTm\ ts) \rangle$   
 $| \langle subtermFm\ (Imp\ p\ q) = subtermFm\ p\ @ \ subtermFm\ q \rangle$   
 $| \langle subtermFm\ (Dis\ p\ q) = subtermFm\ p\ @ \ subtermFm\ q \rangle$   
 $| \langle subtermFm\ (Con\ p\ q) = subtermFm\ p\ @ \ subtermFm\ q \rangle$   
 $| \langle subtermFm\ (Exi\ p) = subtermFm\ p \rangle$   
 $| \langle subtermFm\ (Uni\ p) = subtermFm\ p \rangle$   
 $| \langle subtermFm\ (Neg\ p) = subtermFm\ p \rangle$

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

**abbreviation**  $\langle subtermFms\ z \equiv concat\ (map\ subtermFm\ z) \rangle$

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* ::  $\langle sequent \Rightarrow tm\ list \rangle$  **where**  
 $\langle subterms\ z \equiv case\ remdups\ (subtermFms\ z)\ of$   
 $\quad [] \Rightarrow [Fun\ 0\ []]$   
 $| ts \Rightarrow ts \rangle$

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  $Neg\ (Neg\ p) \in set\ z$  is not necessary for the prover, but makes the proof of the lemma *branchDone-contradiction* easier.

**fun** *branchDone* ::  $\langle sequent \Rightarrow bool \rangle$  **where**  
 $\langle branchDone\ [] = False \rangle$   
 $| \langle branchDone\ (Neg\ p\ \# \ z) = (p \in set\ z \vee Neg\ (Neg\ p) \in set\ z \vee branchDone\ z) \rangle$   
 $| \langle branchDone\ (p\ \# \ z) = (Neg\ p \in set\ z \vee branchDone\ z) \rangle$

### 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* ::  $\langle tm\ list \Rightarrow rule \Rightarrow fm \Rightarrow fm\ list\ list \rangle$  **where**  
 $\langle parts\ A\ r\ f = (case\ (r,\ f)\ of$   
 $\quad (NegNeg,\ Neg\ (Neg\ p)) \Rightarrow [[p]]$   
 $\quad | (AlphaImp,\ Imp\ p\ q) \Rightarrow [[Neg\ p,\ q]]$   
 $\quad | (AlphaDis,\ Dis\ p\ q) \Rightarrow [[p,\ q]]$   
 $\quad | (AlphaCon,\ Neg\ (Con\ p\ q)) \Rightarrow [[Neg\ p,\ Neg\ q]]$   
 $\quad | (BetaImp,\ Neg\ (Imp\ p\ q)) \Rightarrow [[p], [Neg\ q]]$   
 $\quad | (BetaDis,\ Neg\ (Dis\ p\ q)) \Rightarrow [[Neg\ p], [Neg\ q]]$   
 $\quad | (BetaCon,\ Con\ p\ q) \Rightarrow [[p], [q]]$   
 $\quad | (DeltaExi,\ Neg\ (Exi\ p)) \Rightarrow [[Neg\ (sub\ 0\ (Fun\ (generateNew\ A)\ [])\ p)]]$   
 $\quad | (DeltaUni,\ Uni\ p) \Rightarrow [[sub\ 0\ (Fun\ (generateNew\ A)\ [])\ p]]$   
 $\quad | (GammaExi,\ Exi\ p) \Rightarrow [Exi\ p\ \# \ map\ (\lambda t.\ sub\ 0\ t\ p)\ A]$   
 $\quad | (GammaUni,\ Neg\ (Uni\ p)) \Rightarrow [Neg\ (Uni\ p)\ \# \ map\ (\lambda t.\ Neg\ (sub\ 0\ t\ p))\ A]$   
 $\quad | - \Rightarrow [[f]] \rangle$

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* ::  $\langle 'a\ list\ list \Rightarrow 'a\ list\ list \Rightarrow 'a\ list\ list \rangle$  **where**  
 $\langle list-prod\ -\ [] = [] \rangle$   
 $| \langle list-prod\ hs\ (t\ \# \ ts) = map\ (\lambda h.\ h\ @\ t)\ hs\ @\ list-prod\ hs\ ts \rangle$

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* ::  $\langle tm\ list \Rightarrow rule \Rightarrow sequent \Rightarrow sequent\ list \rangle$  **where**  
 $\langle children\ -\ -\ [] = [[]] \rangle$   
 $| \langle children\ A\ r\ (p\ \# \ z) =$   
 $\quad (let\ hs = parts\ A\ r\ p;\ A' = remdups\ (A\ @\ subtermFms\ (concat\ hs))$   
 $\quad in\ list-prod\ hs\ (children\ A'\ r\ z)) \rangle$

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

**type-synonym** *state* =  $\langle tm\ list \times sequent \rangle$

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* ::  $\langle \text{rule} \Rightarrow \text{state} \Rightarrow \text{state fset} \rangle$  **where**  
 $\langle \text{effect } r (A, z) =$   
 $(\text{if } \text{branchDone } z \text{ then } \{\}\ \text{else}$   
 $\quad \text{fimage } (\lambda z'. (\text{remdups } (A @ \text{subterms } z @ \text{subterms } z'), z'))$   
 $\quad (\text{fset-of-list } (\text{children } (\text{remdups } (A @ \text{subtermFms } z)) r z))) \rangle$

#### 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**  $\langle \text{rulesList} \equiv [$   
 $\quad \text{NegNeg}, \text{AlphaImp}, \text{AlphaDis}, \text{AlphaCon},$   
 $\quad \text{DeltaExi}, \text{DeltaUni},$   
 $\quad \text{BetaImp}, \text{BetaDis}, \text{BetaCon},$   
 $\quad \text{GammaExi}, \text{GammaUni}$   
 $\rangle$

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

**definition** *rules* **where**  
 $\langle \text{rules} = \text{cycle rulesList} \rangle$

#### 2.1.5 Abstract completeness

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

**definition** *eff* **where**  
 $\langle \text{eff} \equiv \lambda r s ss. \text{effect } r s = ss \rangle$

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*:  $\langle \forall st. \forall r \in i.R (\text{cycle rulesList}). \exists sl. \text{eff } r st sl \rangle$   
**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*:  $\langle \forall r. r \in R \longrightarrow \text{per } r \rangle$   
**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**  $\langle \text{secavProver} \equiv \text{mkTree rules} \rangle$

**abbreviation**  $\langle \text{rootSequent } t \equiv \text{snd } (\text{fst } (\text{root } t)) \rangle$

**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*]:  $\text{Stream.smember } x (y \#\# s) = (x = y \vee \text{Stream.smember } x s)$   
**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*  $\rightarrow$  (*Haskell*) *MaybeExt.fromJust*  
| **constant** *Option.is-none*  $\rightarrow$  (*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  $\rightarrow$  (Haskell)
   $\langle$ module MaybeExt(fromJust, isNothing) where
    import Data.Maybe(fromJust, isNothing); $\rangle$ 

```

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  $\rightarrow$  (Haskell) Prover
| code-module Prover  $\rightarrow$  (Haskell) Prover
| code-module Export  $\rightarrow$  (Haskell) Prover
| code-module Option  $\rightarrow$  (Haskell) Prover
| code-module MaybeExt  $\rightarrow$  (Haskell) Prover
| code-module Abstract-Completeness  $\rightarrow$  (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  $\langle$ secavTreeCode  $\equiv$  i.mkTree ( $\lambda r s$ . Some (effect r s)) rules $\rangle$ 
definition  $\langle$ secavProverCode  $\equiv$   $\lambda z$  . secavTreeCode ([], z) $\rangle$ 

```

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]:
   $\langle$ paramst (liftt t) = paramst t $\rangle$   $\langle$ paramsts (liftts ts) = paramsts ts $\rangle$ 
  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:
   $\langle$ paramst (sub-term m s t)  $\subseteq$  paramst s  $\cup$  paramst t $\rangle$ 

```

$\langle \text{paramsts } (\text{sub-list } m \ s \ l) \subseteq \text{paramst } s \cup \text{paramsts } l \rangle$   
**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*:  $\langle \text{params } (\text{sub } m \ t \ p) \subseteq \text{paramst } t \cup \text{params } p \rangle$   
**proof** (*induct p arbitrary: m t*)  
**case** (*Pre x1 x2*)  
**then show** *?case*  
**using** *paramst-sub-term(2)* **by** *simp*  
**qed** *fastforce+*

**abbreviation**  $\langle \text{paramss } z \equiv \bigcup p \in \text{set } z. \text{params } p \rangle$

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

**lemma** *news-paramss*:  $\langle \text{news } i \ z \longleftrightarrow i \notin \text{paramss } z \rangle$   
**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*:  $\langle \text{set } A \subseteq \text{set } B \Longrightarrow \text{paramsts } A \subseteq \text{paramsts } B \rangle$   
**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]*:  $\langle \text{size } (\text{sub } i \ t \ p) = \text{size } p \rangle$   
**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* ::  $\langle \text{rule} \Rightarrow \text{rule} \rangle$  **where**  
 $\langle \text{next-rule } \text{NegNeg} = \text{AlphaImp} \rangle$   
 $| \langle \text{next-rule } \text{AlphaImp} = \text{AlphaDis} \rangle$   
 $| \langle \text{next-rule } \text{AlphaDis} = \text{AlphaCon} \rangle$   
 $| \langle \text{next-rule } \text{AlphaCon} = \text{DeltaExi} \rangle$   
 $| \langle \text{next-rule } \text{DeltaExi} = \text{DeltaUni} \rangle$   
 $| \langle \text{next-rule } \text{DeltaUni} = \text{BetaImp} \rangle$   
 $| \langle \text{next-rule } \text{BetaImp} = \text{BetaDis} \rangle$   
 $| \langle \text{next-rule } \text{BetaDis} = \text{BetaCon} \rangle$   
 $| \langle \text{next-rule } \text{BetaCon} = \text{GammaExi} \rangle$   
 $| \langle \text{next-rule } \text{GammaExi} = \text{GammaUni} \rangle$   
 $| \langle \text{next-rule } \text{GammaUni} = \text{NegNeg} \rangle$

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

**primrec** *rule-index* ::  $\langle \text{rule} \Rightarrow \text{nat} \rangle$  **where**

```

   $\langle \text{rule-index } \text{NegNeg} = 0 \rangle$ 
|  $\langle \text{rule-index } \text{AlphaImp} = 1 \rangle$ 
|  $\langle \text{rule-index } \text{AlphaDis} = 2 \rangle$ 
|  $\langle \text{rule-index } \text{AlphaCon} = 3 \rangle$ 
|  $\langle \text{rule-index } \text{DeltaExi} = 4 \rangle$ 
|  $\langle \text{rule-index } \text{DeltaUni} = 5 \rangle$ 
|  $\langle \text{rule-index } \text{BetaImp} = 6 \rangle$ 
|  $\langle \text{rule-index } \text{BetaDis} = 7 \rangle$ 
|  $\langle \text{rule-index } \text{BetaCon} = 8 \rangle$ 
|  $\langle \text{rule-index } \text{GammaExi} = 9 \rangle$ 
|  $\langle \text{rule-index } \text{GammaUni} = 10 \rangle$ 

```

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*:  $\langle \text{distinct rulesList} \rangle$   
**unfolding** *rulesList-def* **by** *simp*

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

**lemma** *cycle-nth*:  $\langle \text{xs} \neq [] \implies \text{cycle xs} !! n = \text{xs} ! (n \bmod \text{length xs}) \rangle$   
**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*:  $\langle \text{rulesList} ! (\text{rule-index } r) = r \rangle$   
**unfolding** *rulesList-def* **by** (*cases r*) *simp-all*

**lemma** *rule-index-bnd*:  $\langle \text{rule-index } r < \text{length rulesList} \rangle$   
**unfolding** *rulesList-def* **by** (*cases r*) *simp-all*

**lemma** *unique-rule-index*:  
**assumes**  $\langle n < \text{length rulesList} \rangle$   $\langle \text{rulesList} ! n = r \rangle$   
**shows**  $\langle n = \text{rule-index } r \rangle$   
**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**  $\langle \text{rules} !! n = r \rangle$   
**shows**  $\langle n \bmod \text{length rulesList} = \text{rule-index } r \rangle$   
**proof** –  
**have**  $*$ :  $\langle \text{rulesList} ! (n \bmod \text{length rulesList}) = r \rangle$   
**using** *assms cycle-nth* **unfolding** *rules-def rulesList-def* **by** (*metis list.distinct(1)*)  
**then show** *?thesis*  
**using** *distinct-rulesList \* unique-rule-index*  
**by** (*cases r*) (*metis length-greater-0-conv list.discI rulesList-def unique-euclidean-semiring-numeral-class.pos-mod-bound*)  
**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  $\langle 0 < k \rangle$ 
  shows  $\langle \forall i < k. \exists n > m. n \bmod k = i \rangle$ 
proof safe
  fix  $i$ 
  let  $?n = \langle (1 + m) * k + i \rangle$ 
  assume  $\langle i < k \rangle$ 
  then have  $\langle ?n \bmod k = i \rangle$ 
    by (metis mod-less mod-mult-self3)
  moreover have  $\langle ?n > m \rangle$ 
    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  $\langle \exists n > m. n \bmod k = i \rangle$ 
    by fast
qed

```

```

lemma mod-suff:
  assumes  $\langle \forall (n :: nat) > m. P (n \bmod k) \rangle \langle 0 < k \rangle$ 
  shows  $\langle \forall i < k. P i \rangle$ 
  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:  $\langle \exists n > m. \text{rules} !! n = r \rangle$ 
proof (rule ccontr)
  assume  $\langle \neg (\exists n > m. \text{rules} !! n = r) \rangle$ 
  then have  $\langle \neg (\exists n > m. n \bmod \text{length rulesList} = \text{rule-index } r) \rangle$ 
    using rule-index-mod nth-rule-index by metis
  then have  $\langle \forall n > m. n \bmod \text{length rulesList} \neq \text{rule-index } r \rangle$ 
    by blast
  moreover have  $\langle \text{length rulesList} > 0 \rangle$ 
    unfolding rulesList-def by simp
  ultimately have  $\langle \forall k < \text{length rulesList}. k \neq \text{rule-index } r \rangle$ 
    using mod-suff [where  $P = \langle \lambda a. a \neq \text{rule-index } r \rangle$ ] 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:  $\langle \exists n. (\text{sdrop } k \text{ rules}) !! n = r \rangle$ 
  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 = ⟨λn. 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) ⊆ set A ⟹ params p ⊆
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
  ⟨∧ i ts. x = Pre i ts ⟹ P⟩
  ⟨∧ p q. x = Imp p q ⟹ P⟩
  ⟨∧ p q. x = Dis p q ⟹ P⟩
  ⟨∧ p q. x = Con p q ⟹ P⟩
  ⟨∧ p. x = Exi p ⟹ P⟩
  ⟨∧ p. x = Uni p ⟹ P⟩
  ⟨∧ i ts. x = Neg (Pre i ts) ⟹ P⟩
  ⟨∧ p q. x = Neg (Imp p q) ⟹ P⟩
  ⟨∧ p q. x = Neg (Dis p q) ⟹ P⟩
  ⟨∧ p q. x = Neg (Con p q) ⟹ P⟩
  ⟨∧ p. x = Neg (Exi p) ⟹ P⟩
  ⟨∧ p. x = Neg (Uni p) ⟹ P⟩
  ⟨∧ p. x = Neg (Neg p) ⟹ 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
  ⟨∧ p q. r = AlphaDis ⟹ x = Dis p q ⟹ P⟩
  ⟨∧ p q. r = AlphaImp ⟹ x = Imp p q ⟹ P⟩
  ⟨∧ p q. r = AlphaCon ⟹ x = Neg (Con p q) ⟹ P⟩
  ⟨∧ p q. r = BetaDis ⟹ x = Neg (Dis p q) ⟹ P⟩
  ⟨∧ p q. r = BetaImp ⟹ x = Neg (Imp p q) ⟹ P⟩
  ⟨∧ p q. r = BetaCon ⟹ x = Con p q ⟹ P⟩
  ⟨∧ p. r = DeltaUni ⟹ x = Uni p ⟹ P⟩
  ⟨∧ p. r = DeltaExi ⟹ x = Neg (Exi p) ⟹ P⟩

```

```

  ⟨ $\bigwedge p. r = \text{NegNeg} \implies x = \text{Neg} (\text{Neg } p) \implies P$ ⟩
  ⟨ $\bigwedge p. r = \text{GammaExi} \implies x = \text{Exi } p \implies P$ ⟩
  ⟨ $\bigwedge p. r = \text{GammaUni} \implies x = \text{Neg} (\text{Uni } p) \implies P$ ⟩
  ⟨ $\forall A. \text{parts } A \ r \ x = [[x]] \implies P$ ⟩
shows  $P$ 
using  $\text{assms}$ 
proof ( $\text{cases } r$ )
  case  $\text{BetaCon}$ 
  then show  $?thesis$ 
    using  $\text{assms}$ 
    proof ( $\text{cases } x \ \text{rule: } \text{Neg-exhaust}$ )
      case ( $\text{Con } p \ q$ )
      then show  $?thesis$ 
        using  $\text{BetaCon } \text{assms}$  by  $\text{blast}$ 
      qed ( $\text{simp-all add: parts-def}$ )
  next
  case  $\text{BetaImp}$ 
  then show  $?thesis$ 
    using  $\text{assms}$ 
    proof ( $\text{cases } x \ \text{rule: } \text{Neg-exhaust}$ )
      case ( $\text{NegImp } p \ q$ )
      then show  $?thesis$ 
        using  $\text{BetaImp } \text{assms}$  by  $\text{blast}$ 
      qed ( $\text{simp-all add: parts-def}$ )
  next
  case  $\text{DeltaUni}$ 
  then show  $?thesis$ 
    using  $\text{assms}$ 
    proof ( $\text{cases } x \ \text{rule: } \text{Neg-exhaust}$ )
      case ( $\text{Uni } p$ )
      then show  $?thesis$ 
        using  $\text{DeltaUni } \text{assms}$  by  $\text{fast}$ 
      qed ( $\text{simp-all add: parts-def}$ )
  next
  case  $\text{DeltaExi}$ 
  then show  $?thesis$ 
    using  $\text{assms}$ 
    proof ( $\text{cases } x \ \text{rule: } \text{Neg-exhaust}$ )
      case ( $\text{NegExi } p$ )
      then show  $?thesis$ 
        using  $\text{DeltaExi } \text{assms}$  by  $\text{fast}$ 
      qed ( $\text{simp-all add: parts-def}$ )
  next
  case  $n: \text{NegNeg}$ 
  then show  $?thesis$ 
    using  $\text{assms}$ 
    proof ( $\text{cases } x \ \text{rule: } \text{Neg-exhaust}$ )
      case ( $\text{NegNeg } p$ )
      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* ::  $\langle \text{rule} \Rightarrow \text{fm} \Rightarrow \text{bool} \rangle$  **where**

```

  affects r p  $\equiv$  case (r, p) of
    (AlphaDis, Dis -)  $\Rightarrow$  True
  | (AlphaImp, Imp -)  $\Rightarrow$  True
  | (AlphaCon, Neg (Con -))  $\Rightarrow$  True
  | (BetaCon, Con -)  $\Rightarrow$  True
  | (BetaImp, Neg (Imp -))  $\Rightarrow$  True
  | (BetaDis, Neg (Dis -))  $\Rightarrow$  True
  | (DeltaUni, Uni -)  $\Rightarrow$  True
  | (DeltaExi, Neg (Exi -))  $\Rightarrow$  True
  | (NegNeg, Neg (Neg -))  $\Rightarrow$  True
  | (GammaExi, Exi -)  $\Rightarrow$  False
  | (GammaUni, Neg (Uni -))  $\Rightarrow$  False
  | (-, -)  $\Rightarrow$  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  $\langle \neg \text{affects } r \ p \rangle \langle z' \in \text{set } (\text{parts } A \ r \ p) \rangle$ 

```

**shows**  $\langle p \in \text{set } z' \rangle$   
**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*:  
 $\langle \text{set } (\text{list-prod } hs \ ts) = \{h \ @ \ t \mid h \ t. \ h \in \text{set } hs \ \wedge \ t \in \text{set } ts\} \rangle$   
**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*:  
 $\langle \text{set } (\text{children } A \ r \ (p \ # \ z)) =$   
 $\{hs \ @ \ ts \mid hs \ ts. \ hs \in \text{set } (\text{parts } A \ r \ p) \ \wedge$   
 $ts \in \text{set } (\text{children } (\text{remdups } (A \ @ \ \text{subtermFms } (\text{concat } (\text{parts } A \ r \ p)))) \ r \ z)\} \rangle$   
**using** *list-prod-is-cartesian* **by** (*metis* *children.simps(2)*)

The *children* function does not change unaffected formulas.

**lemma** *children-preserves-unaffected*:  
**assumes**  $\langle p \in \text{set } z \rangle \ \langle \neg \text{affects } r \ p \rangle \ \langle z' \in \text{set } (\text{children } A \ r \ z) \rangle$   
**shows**  $\langle p \in \text{set } z' \rangle$   
**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**  $\langle p \in \text{set } z \rangle$  **and**  $\langle \neg \text{affects } r \ p \rangle$  **and**  $\langle (B, z') \mid \in \mid \text{effect } r \ (A, z) \rangle$   
**shows**  $\langle p \in \text{set } z' \rangle$   
**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**  $\langle p \in \text{set } z \rangle \ \langle z' \in \text{set } (\text{children } A \ r \ z) \rangle$   
**shows**  $\langle \exists B \ xs. \ \text{set } A \subseteq \text{set } B \ \wedge \ xs \in \text{set } (\text{parts } B \ r \ p) \ \wedge \ \text{set } xs \subseteq \text{set } z' \rangle$   
**using** *assms*  
**proof** (*induct* *z arbitrary: A z'*)  
**case** (*Cons a -*)  
**then show** *?case*  
**proof** (*cases*  $\langle a = p \rangle$ )

```

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*:  $\langle (B, z') \mid \in \mid \text{effect } r (A, z) \rangle \implies \neg \text{branchDone } z \rangle$   
**by** (*cases*  $\langle \text{branchDone } z \rangle$ ) *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  $\langle p \in \text{set } z \rangle$  and  $\langle (B, z') \mid \in \mid \text{effect } r (A, z) \rangle$ 
shows  $\langle \exists C \text{ } xs. \text{set } A \subseteq \text{set } C \wedge xs \in \text{set } (\text{parts } C \text{ } r \text{ } p) \wedge \text{set } xs \subseteq \text{set } z' \rangle$ 
using assms parts-in-children ne-effect-not-branchDone
by (smt (verit, cfv-threshold) Pair-inject effect.simps fimageE fset-of-list-elim
le-sup-iff
set-append set-remdups)

```

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

**corollary**  $\langle \text{Neg } (\text{Neg } p) \in \text{set } z \rangle \implies \langle (B, z') \mid \in \mid \text{effect } \text{NegNeg } (A, z) \rangle \implies p \in \text{set } z' \rangle$   
**using** *parts-in-effect unfolding parts-def* **by** *fastforce*

**corollary**  $\langle \text{Neg } (\text{Uni } p) \in \text{set } z \rangle \implies \langle (B, z') \mid \in \mid \text{effect } \text{GammaUni } (A, z) \rangle \implies$   
 $\text{set } (\text{map } (\lambda t. \text{Neg } (\text{sub } 0 \text{ } t \text{ } p)) \text{ } A) \subseteq \text{set } z' \rangle$   
**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  $\langle \neg \text{branchDone } z \rangle$   $\langle \text{eff } r (A, z) \text{ } ss \rangle$ 
shows  $\langle \forall z' \in \text{set } (\text{children } (\text{remdups } (A \text{ } @ \text{ subtermFms } z)) \text{ } r \text{ } z). \exists B. (B, z') \mid \in \mid$ 
 $ss \rangle$ 
using assms unfolding eff-def using fset-of-list-elim 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**  $xs :: \langle \text{nat list} \rangle$

**shows**  $\langle \text{foldr max } xs \ 0 = (\text{if } xs = [] \text{ then } 0 \text{ else } \text{Max } (\text{set } xs)) \rangle$

**by**  $(\text{induct } xs) \text{ simp-all}$

**lemma** *Suc-max-new*:

**fixes**  $xs :: \langle \text{nat list} \rangle$

**shows**  $\langle \text{Suc } (\text{foldr max } xs \ 0) \notin \text{set } xs \rangle$

**proof**  $(\text{cases } xs)$

**case**  $(\text{Cons } x \ xs)$

**then have**  $\langle \text{foldr max } (x \# \ xs) \ 0 = \text{Max } (\text{set } (x \# \ xs)) \rangle$

**using** *foldr-max* **by** *simp*

**then show** *?thesis*

**using** *Cons* **by**  $(\text{metis } \text{List.finite-set } \text{Max.insert } \text{add-0 } \text{empty-iff } \text{list.set}(2) \text{max-0-1}(2))$

$n\text{-not-Suc-}n \text{ nat-add-max-left plus-1-eq-Suc remdups.simps}(2) \text{ set-remdups}$

**qed** *simp*

**lemma** *listFunTm-paramst*:  $\langle \text{set } (\text{listFunTm } t) = \text{paramst } t \rangle \langle \text{set } (\text{listFunTms } ts) = \text{paramsts } ts \rangle$

**by**  $(\text{induct } t \ \text{and } ts \ \text{rule: } \text{paramst.induct } \text{paramsts.induct}) \ \text{auto}$

### 2.3.8 Finding axioms

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

**lemma** *branchDone-contradiction*:  $\langle \text{branchDone } z \longleftrightarrow (\exists p. p \in \text{set } z \wedge \text{Neg } p \in \text{set } z) \rangle$

**by**  $(\text{induct } z \ \text{rule: } \text{branchDone.induct}) \ \text{auto}$

### 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*]:  $\langle t \in \text{set } (\text{subtermTm } t) \rangle$

**by**  $(\text{induct } t) \ \text{simp-all}$

The arguments of a predicate are subterms of it.

**lemma** *subterm-Pre-refl*:  $\langle \text{set } ts \subseteq \text{set } (\text{subtermFm } (\text{Pre } n \ ts)) \rangle$

**by**  $(\text{induct } ts) \ \text{auto}$

The arguments of function are subterms of it.

**lemma** *subterm-Fun-refl*:  $\langle \text{set } ts \subseteq \text{set } (\text{subtermTm } (\text{Fun } n \text{ } ts)) \rangle$   
**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* ::  $\langle \text{fm} \Rightarrow \text{fm set} \rangle$  **where**  
 $\langle \text{preds } (\text{Pre } n \text{ } ts) = \{\text{Pre } n \text{ } ts\} \rangle$   
 $\langle \text{preds } (\text{Imp } p \text{ } q) = \text{preds } p \cup \text{preds } q \rangle$   
 $\langle \text{preds } (\text{Dis } p \text{ } q) = \text{preds } p \cup \text{preds } q \rangle$   
 $\langle \text{preds } (\text{Con } p \text{ } q) = \text{preds } p \cup \text{preds } q \rangle$   
 $\langle \text{preds } (\text{Exi } p) = \text{preds } p \rangle$   
 $\langle \text{preds } (\text{Uni } p) = \text{preds } p \rangle$   
 $\langle \text{preds } (\text{Neg } p) = \text{preds } p \rangle$

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

**lemma** *subtermFm-preds*:  $\langle t \in \text{set } (\text{subtermFm } p) \longleftrightarrow (\exists \text{pre} \in \text{preds } p. t \in \text{set } (\text{subtermFm } \text{pre})) \rangle$   
**by** (*induct* *p*) *auto*

**lemma** *preds-shape*:  $\langle \text{pre} \in \text{preds } p \Longrightarrow \exists n \text{ } ts. \text{pre} = \text{Pre } n \text{ } ts \rangle$   
**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**  $\langle \text{Fun } n \text{ } ts \in \text{set } (\text{subtermFm } p) \rangle$   
**shows**  $\langle \text{set } ts \subseteq \text{set } (\text{subtermFm } p) \rangle$

**proof** –

**obtain** *pre* **where** *pre*:  $\langle \text{pre} \in \text{preds } p \rangle \langle \text{Fun } n \text{ } ts \in \text{set } (\text{subtermFm } \text{pre}) \rangle$   
**using** *assms* *subtermFm-preds* **by** *blast*  
**then obtain** *n'* *ts'* **where**  $\langle \text{pre} = \text{Pre } n' \text{ } ts' \rangle$   
**using** *preds-shape* **by** *blast*  
**then have**  $\langle \text{set } ts \subseteq \text{set } (\text{subtermFm } \text{pre}) \rangle$   
**using** *subtermTm-le* *pre* **by** *force*  
**then have**  $\langle \text{set } ts \subseteq \text{set } (\text{subtermFm } p) \rangle$   
**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**

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

**locale** *Hintikka* =

**fixes**  $H :: \langle fm\ set \rangle$

**assumes**

*Basic*:  $\langle Pre\ n\ ts \in H \implies Neg\ (Pre\ n\ ts) \notin H \rangle$  **and**

*AlphaDis*:  $\langle Dis\ p\ q \in H \implies p \in H \wedge q \in H \rangle$  **and**

*AlphaImp*:  $\langle Imp\ p\ q \in H \implies Neg\ p \in H \wedge q \in H \rangle$  **and**

*AlphaCon*:  $\langle Neg\ (Con\ p\ q) \in H \implies Neg\ p \in H \wedge Neg\ q \in H \rangle$  **and**

*BetaCon*:  $\langle Con\ p\ q \in H \implies p \in H \vee q \in H \rangle$  **and**

*BetaImp*:  $\langle Neg\ (Imp\ p\ q) \in H \implies p \in H \vee Neg\ q \in H \rangle$  **and**

*BetaDis*:  $\langle Neg\ (Dis\ p\ q) \in H \implies Neg\ p \in H \vee Neg\ q \in H \rangle$  **and**

*GammaExi*:  $\langle Exi\ p \in H \implies \forall t \in terms\ H. sub\ 0\ t\ p \in H \rangle$  **and**

*GammaUni*:  $\langle Neg\ (Uni\ p) \in H \implies \forall t \in terms\ H. Neg\ (sub\ 0\ t\ p) \in H \rangle$  **and**

*DeltaUni*:  $\langle Uni\ p \in H \implies \exists t \in terms\ H. sub\ 0\ t\ p \in H \rangle$  **and**

*DeltaExi*:  $\langle Neg\ (Exi\ p) \in H \implies \exists t \in terms\ H. Neg\ (sub\ 0\ t\ p) \in H \rangle$  **and**

*Neg*:  $\langle Neg\ (Neg\ p) \in H \implies p \in H \rangle$

**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* ::  $\langle state \times rule \Rightarrow sequent \rangle$  **where**

$\langle pseq\ z = snd\ (fst\ z) \rangle$

*ptms* is the list of terms in a node.

**definition** *ptms* ::  $\langle state \times rule \Rightarrow tm\ list \rangle$  **where**

$\langle ptms\ z = fst\ (fst\ z) \rangle$

## 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*:  $\langle \text{epath steps} \implies \text{epath (sdrop } n \text{ steps)} \rangle$   
**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*:  $\langle \text{sset (sdrop } n \text{ s)} \subseteq \text{sset s} \rangle$   
**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**  $\langle \text{epath steps} \rangle \langle \text{eff (snd (shd steps)) (fst (shd steps)) ss} \rangle$   
**shows**  $\langle \text{fst (shd (stl steps))} \mid \in \mid \text{ss} \rangle$   
**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**  $\langle (B, z') \mid \in \mid \text{effect } r (A, z) \rangle$   
**shows**  $\langle B = \text{remdups } (A @ \text{subterms } z @ \text{subterms } z') \rangle$   
**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**  $\langle \text{epath steps} \rangle \langle \text{shd steps} = ((A, z), r) \rangle$   
**shows**  $\langle \exists B z' r'. (B, z') \mid \in \mid \text{effect } r (A, z) \wedge \text{shd (stl steps)} = ((B, z'), r') \wedge (B = \text{remdups } (A @ \text{subterms } z @ \text{subterms } z')) \rangle$   
**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**  $\langle \text{epath steps} \rangle$

```

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-unaaffected*:

**assumes**  $\langle p \in \text{set } (\text{pseq } (\text{shd } \text{steps})) \rangle$  **and**  $\langle \text{epath } \text{steps} \rangle$  **and**  $\langle \text{steps} = \text{pre } @- \text{suf} \rangle$  **and**  
 $\langle \text{list-all } (\text{not } (\lambda \text{step. affects } (\text{snd } \text{step}) p)) \text{ pre} \rangle$   
**shows**  $\langle p \in \text{set } (\text{pseq } (\text{shd } \text{suf})) \rangle$   
**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-unaaffected 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**  $\langle \text{tree-fms } \text{steps} \equiv \bigcup \text{ss} \in \text{sset } \text{steps. set } (\text{pseq } \text{ss}) \rangle$

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

**lemma** *pseq-in-tree-fms*:  $\langle \llbracket x \in \text{sset } \text{steps}; p \in \text{set } (\text{pseq } x) \rrbracket \implies p \in \text{tree-fms } \text{steps} \rangle$   
**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*:  $\langle p \in \text{tree-fms } \text{steps} \implies \exists n. p \in \text{set } (\text{pseq } (\text{steps } !! n)) \rangle$   
**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*:  $\langle \text{Saturated } \text{steps} \implies \text{Saturated } (\text{sdrop } n \text{ steps}) \rangle$   
**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**  $\langle is\text{-}rule\ r\ step \equiv snd\ step = r \rangle$

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**  $\langle Saturated\ steps \rangle$

**shows**  $\langle ev\ (holds\ (is\text{-}rule\ r))\ (sdrop\ n\ steps) \rangle$

**proof** –

**have**  $\langle Saturated\ (sdrop\ n\ steps) \rangle$

**using**  $\langle Saturated\ steps \rangle$  *Saturated-sdrop* **by** *fast*

**moreover have**  $\langle r \in Prover.R \rangle$

**by** *(metis rules-repeat snth-sset)*

**ultimately have**  $\langle saturated\ r\ (sdrop\ n\ steps) \rangle$

**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**  $\langle epath\ steps \rangle$

**shows**  $\langle alw\ (holds\ (not\ (branchDone\ o\ pseq)))\ steps \rangle$

**proof** *(rule ccontr)*

**assume**  $\langle \neg\ ?thesis \rangle$

**then have**  $\langle ev\ (holds\ (branchDone\ o\ pseq))\ steps \rangle$

**by** *(simp add: alw-iff-sdrop ev-iff-sdrop)*

**then obtain**  $n$  **where**  $n$ :  $\langle holds\ (branchDone\ o\ pseq)\ (sdrop\ n\ steps) \rangle$

**using** *sdrop-wait* **by** *blast*

**let**  $?suf = \langle sdrop\ n\ steps \rangle$

**have**  $\langle \forall r\ A.\ effect\ r\ (A,\ pseq\ (shd\ ?suf)) = \{\|\} \rangle$

**unfolding** *effect-def* **using**  $n$  **by** *simp*

**moreover have**  $\langle epath\ ?suf \rangle$

**using**  $\langle epath\ steps \rangle$  *epath-sdrop* **by** *blast*

**then have**  $\langle \forall r\ A.\ \exists z'\ r'.\ z' \in effect\ r\ (A,\ pseq\ (shd\ ?suf)) \wedge shd\ (stl\ ?suf) = (z',\ r') \rangle$

**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**  $\langle \text{epath steps} \rangle$  **and**  $\langle \text{Saturated steps} \rangle$   
**shows**  $\langle \text{Hintikka (tree-fms steps)} \rangle$   
(is  $\langle \text{Hintikka } ?H \rangle$ )

**proof**

**fix**  $n\ ts$

**assume**  $\text{pre}: \langle \text{Pre } n\ ts \in ?H \rangle$

**then obtain**  $m$  **where**  $m: \langle \text{Pre } n\ ts \in \text{set (pseq (shd (sdrop } m\ \text{steps)))} \rangle$   
**using** *tree-fms-in-pseq* **by** *auto*

**show**  $\langle \text{Neg (Pre } n\ ts) \notin ?H \rangle$

**proof**

**assume**  $\langle \text{Neg (Pre } n\ ts) \in ?H \rangle$

**then obtain**  $k$  **where**  $k: \langle \text{Neg (Pre } n\ ts) \in \text{set (pseq (shd (sdrop } k\ \text{steps)))} \rangle$   
**using** *tree-fms-in-pseq* **by** *auto*

**let**  $?pre = \langle \text{stake (} m + k \text{) steps} \rangle$

**let**  $?suf = \langle \text{sdrop (} m + k \text{) steps} \rangle$

**have**

1:  $\langle \neg \text{affects } r\ (\text{Pre } n\ ts) \rangle$  **and**

2:  $\langle \neg \text{affects } r\ (\text{Neg (Pre } n\ ts)) \rangle$  **for**  $r$

**unfolding** *affects-def* **by** (*cases*  $r$ , *simp-all*)**+**

**have**  $\langle \text{list-all (not } (\lambda \text{step. affects (snd step) (Pre } n\ ts)))\ ?pre \rangle$

**unfolding** *list-all-def* **using** 1 **by** (*induct*  $?pre$ ) *simp-all*

**then have**  $p: \langle \text{Pre } n\ ts \in \text{set (pseq (shd } ?suf)) \rangle$

**using**  $\langle \text{epath steps} \rangle$  *epath-preserves-unaaffected*  $m$  *epath-sdrop*

**by** (*metis* (*no-types*, *lifting*) *list.pred-mono-strong* *list-all-append*  
*sdrop-add* *stake-add* *stake-sdrop*)

**have**  $\langle \text{list-all (not } (\lambda \text{step. affects (snd step) (Neg (Pre } n\ ts))))\ ?pre \rangle$

**unfolding** *list-all-def* **using** 2 **by** (*induct*  $?pre$ ) *simp-all*

**then have**  $np: \langle \text{Neg (Pre } n\ ts) \in \text{set (pseq (shd } ?suf)) \rangle$

**using**  $\langle \text{epath steps} \rangle$  *epath-preserves-unaaffected*  $k$  *epath-sdrop*

**by** (*smt* (*verit*, *best*) *add.commute* *list.pred-mono-strong* *list-all-append*  
*sdrop-add*  
*stake-add* *stake-sdrop*)

**have**  $\langle \text{holds (branchDone } o\ \text{pseq)}\ ?suf \rangle$

**using**  $p\ np$  *branchDone-contradiction* **by** *auto*

**moreover have**  $\langle \neg \text{holds (branchDone } o\ \text{pseq)}\ ?suf \rangle$

**using**  $\langle \text{epath steps} \rangle$  *epath-never-branchDone* **by** (*simp* *add: alw-iff-sdrop*)

```

    ultimately show False
      by blast
qed
next
fix p q
assume  $\langle \text{Dis } p \ q \in ?H \rangle$  (is  $\langle ?f \in ?H \rangle$ )
then obtain n where n:  $\langle ?f \in \text{set } (\text{pseq } (\text{shd } (\text{sdrop } n \ \text{steps}))) \rangle$ 
  using tree-fms-in-pseq by auto
let ?steps =  $\langle \text{sdrop } n \ \text{steps} \rangle$ 
let ?r = AlphaDis
have  $\langle \text{ev } (\text{holds } (\text{is-rule } ?r)) \ ?steps \rangle$ 
  using  $\langle \text{Saturated steps} \rangle$  Saturated-ev-rule by blast
then obtain pre suf where
  pre:  $\langle \text{list-all } (\text{not } (\text{is-rule } ?r)) \ \text{pre} \rangle$  and
  suf:  $\langle \text{holds } (\text{is-rule } ?r) \ \text{suf} \rangle$  and
  ori:  $\langle ?steps = \text{pre } @- \ \text{suf} \rangle$ 
  using ev-prefix by blast

have  $\langle \text{affects } r \ ?f \longleftrightarrow r = ?r \rangle$  for r
  unfolding affects-def by (cases r) simp-all
then have  $\langle \text{list-all } (\text{not } (\lambda \text{step. affects } (\text{snd } \text{step}) \ ?f)) \ \text{pre} \rangle$ 
  using pre by simp
moreover have  $\langle \text{epath } (\text{pre } @- \ \text{suf}) \rangle$ 
  using  $\langle \text{epath steps} \rangle$  epath-sdrop ori by metis
ultimately have  $\langle ?f \in \text{set } (\text{pseq } (\text{shd } \text{suf})) \rangle$ 
  using epath-preserves-unaaffected n ori by metis

moreover have  $\langle \text{epath } \ \text{suf} \rangle$ 
  using  $\langle \text{epath } (\text{pre } @- \ \text{suf}) \rangle$  epath-sdrop by (metis alwD alw-iff-sdrop alw-shift)
then obtain B z' r' where
  z':  $\langle (B, z') \mid \text{effect } ?r \ (\text{ptms } (\text{shd } \text{suf}), \ \text{pseq } (\text{shd } \text{suf})) \rangle$   $\langle \text{shd } (\text{stl } \text{suf}) = ((B,$ 
 $z'), r') \rangle$ 
  using suf epath-effect unfolding pseq-def ptms-def
  by (metis (mono-tags, lifting) holds.elims(2) prod.collapse)
ultimately have  $\langle p \in \text{set } z' \rangle$   $\langle q \in \text{set } z' \rangle$ 
  using parts-in-effect unfolding parts-def by fastforce+

then show  $\langle p \in ?H \wedge q \in ?H \rangle$ 
  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  $\langle \text{Imp } p \ q \in \text{tree-fms } \text{steps} \rangle$  (is  $\langle ?f \in ?H \rangle$ )
then obtain n where n:  $\langle ?f \in \text{set } (\text{pseq } (\text{shd } (\text{sdrop } n \ \text{steps}))) \rangle$ 
  using tree-fms-in-pseq by auto
let ?steps =  $\langle \text{sdrop } n \ \text{steps} \rangle$ 
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  $\longleftrightarrow$  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-unaaffected 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  $\longleftrightarrow$  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-unaaffected 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 (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-unaaffected 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

```

$z': \langle (B, z') \mid \in \mid \text{effect } ?r \text{ (ptms (shd suf), pseq (shd suf))} \rangle \langle \text{shd (stl suf)} \rangle = ((B, z'), r')$

**using** *suf epath-effect unfolding pseq-def ptms-def*  
**by** (*metis (mono-tags, lifting) holds.elims(2) prod.collapse*)  
**ultimately consider**  $\langle p \in \text{set } z' \mid \langle q \in \text{set } z' \rangle$   
**using** *parts-in-effect unfolding parts-def by fastforce*

**then show**  $\langle p \in ?H \vee q \in ?H \rangle$   
**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**  $\langle \text{Neg (Imp } p q) \in ?H \rangle$  (**is**  $\langle ?f \in ?H \rangle$ )  
**then obtain**  $n$  **where**  $n: \langle ?f \in \text{set (pseq (shd (sdrop n steps)))} \rangle$   
**using** *tree-fms-in-pseq by auto*  
**let**  $?steps = \langle \text{sdrop } n \text{ steps} \rangle$   
**let**  $?r = \text{BetaImp}$   
**have**  $\langle \text{ev (holds (is-rule } ?r)) ?steps \rangle$   
**using**  $\langle \text{Saturated steps} \rangle$  *Saturated-ev-rule by blast*  
**then obtain**  $\text{pre suf}$  **where**  
 $\text{pre}: \langle \text{list-all (not (is-rule } ?r)) \text{pre} \rangle$  **and**  
 $\text{suf}: \langle \text{holds (is-rule } ?r) \text{suf} \rangle$  **and**  
 $\text{ori}: \langle ?steps = \text{pre} @- \text{suf} \rangle$   
**using** *ev-prefix by blast*

**have**  $\langle \text{affects } r ?f \longleftrightarrow r = ?r \rangle$  **for**  $r$   
**unfolding** *affects-def by (cases r) simp-all*  
**then have**  $\langle \text{list-all (not } (\lambda \text{step. affects (snd step) ?f)) \text{pre} \rangle$   
**using** *pre by simp*  
**moreover have**  $\langle \text{epath (pre} @- \text{suf)} \rangle$   
**using**  $\langle \text{epath steps} \rangle$  *epath-sdrop ori by metis*  
**ultimately have**  $\langle ?f \in \text{set (pseq (shd suf))} \rangle$   
**using** *epath-preserves-unaaffected n ori by metis*

**moreover have**  $\langle \text{epath suf} \rangle$   
**using**  $\langle \text{epath (pre} @- \text{suf)} \rangle$  *epath-sdrop by (metis alwD alw-iff-sdrop alw-shift)*  
**then obtain**  $B z' r'$  **where**  
 $z': \langle (B, z') \mid \in \mid \text{effect } ?r \text{ (ptms (shd suf), pseq (shd suf))} \rangle \langle \text{shd (stl suf)} \rangle = ((B, z'), r')$

**using** *suf epath-effect unfolding pseq-def ptms-def*  
**by** (*metis (mono-tags, lifting) holds.elims(2) prod.collapse*)  
**ultimately consider**  $\langle p \in \text{set } z' \mid \langle \text{Neg } q \in \text{set } z' \rangle$   
**using** *parts-in-effect unfolding parts-def by fastforce*

**then show**  $\langle p \in ?H \vee \text{Neg } q \in ?H \rangle$   
**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  $\langle \text{Neg } (Dis\ p\ q) \in ?H \rangle$  (is  $\langle ?f \in ?H \rangle$ )
  then obtain n where n:  $\langle ?f \in \text{set } (pseq\ (shd\ (sdrop\ n\ steps))) \rangle$ 
    using tree-fms-in-pseq by auto
  let ?steps =  $\langle sdrop\ n\ steps \rangle$ 
  let ?r = BetaDis
  have  $\langle ev\ (holds\ (is-rule\ ?r))\ ?steps \rangle$ 
    using  $\langle \text{Saturated steps} \rangle$  Saturated-ev-rule by blast
  then obtain pre suf where
    pre:  $\langle list-all\ (not\ (is-rule\ ?r))\ pre \rangle$  and
    suf:  $\langle holds\ (is-rule\ ?r)\ suf \rangle$  and
    ori:  $\langle ?steps = pre\ @-\ suf \rangle$ 
    using ev-prefix by blast

  have  $\langle affects\ r\ ?f \longleftrightarrow r = ?r \rangle$  for r
    unfolding affects-def by (cases r) simp-all
  then have  $\langle list-all\ (not\ (\lambda step. affects\ (snd\ step)\ ?f))\ pre \rangle$ 
    using pre by simp
  moreover have  $\langle epath\ (pre\ @-\ suf) \rangle$ 
    using  $\langle epath\ steps \rangle$  epath-sdrop ori by metis
  ultimately have  $\langle ?f \in \text{set } (pseq\ (shd\ suf)) \rangle$ 
    using epath-preserves-unaaffected n ori by metis

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

  then show  $\langle \text{Neg } p \in ?H \vee \text{Neg } q \in ?H \rangle$ 
    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  $\langle Exi\ p \in ?H \rangle$  (is  $\langle ?f \in ?H \rangle$ )
  then obtain n where n:  $\langle ?f \in \text{set } (pseq\ (shd\ (sdrop\ n\ steps))) \rangle$ 
    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-unaaffected 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-unaaffected 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 (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-unaaffected 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  $\langle \text{Neg } (\text{Exi } p) \in \text{tree-fms steps} \rangle$  (is  $\langle ?f \in ?H \rangle$ )
then obtain  $n$  where  $n$ :  $\langle ?f \in \text{set } (\text{pseq } (\text{shd } (\text{sdrop } n \text{ steps}))) \rangle$ 
  using tree-fms-in-pseq by auto
let  $?steps = \langle \text{sdrop } n \text{ steps} \rangle$ 
let  $?r = \text{DeltaExi}$ 
have  $\langle \text{ev } (\text{holds } (\text{is-rule } ?r)) ?steps \rangle$ 
  using  $\langle \text{Saturated steps} \rangle$  Saturated-ev-rule by blast
then obtain  $\text{pre suf}$  where
   $\text{pre}$ :  $\langle \text{list-all } (\text{not } (\text{is-rule } ?r)) \text{pre} \rangle$  and
   $\text{suf}$ :  $\langle \text{holds } (\text{is-rule } ?r) \text{suf} \rangle$  and
   $\text{ori}$ :  $\langle ?steps = \text{pre } @- \text{suf} \rangle$ 
  using ev-prefix by blast

have  $\langle \text{affects } r ?f \longleftrightarrow r = ?r \rangle$  for  $r$ 
  unfolding affects-def by (cases  $r$ ) simp-all
then have  $\langle \text{list-all } (\text{not } (\lambda \text{step. affects } (\text{snd } \text{step}) ?f)) \text{pre} \rangle$ 
  using  $\text{pre}$  by simp
moreover have  $\langle \text{epath } (\text{pre } @- \text{suf}) \rangle$ 
  using  $\langle \text{epath steps} \rangle$  epath-sdrop ori by metis
ultimately have  $\langle ?f \in \text{set } (\text{pseq } (\text{shd } \text{suf})) \rangle$ 
  using epath-preserves-unaaffected n ori by metis

moreover have  $\langle \text{epath } \text{suf} \rangle$ 
  using  $\langle \text{epath } (\text{pre } @- \text{suf}) \rangle$  epath-sdrop by (metis alwD alw-iff-sdrop alw-shift)
then obtain  $B z' r'$  where
   $z'$ :  $\langle (B, z') \mid \text{effect } ?r (\text{ptms } (\text{shd } \text{suf}), \text{pseq } (\text{shd } \text{suf})) \rangle$   $\langle \text{shd } (\text{stl } \text{suf}) = ((B, z'), r') \rangle$ 
  using  $\text{suf}$  epath-effect unfolding pseq-def ptms-def
  by (metis (mono-tags, lifting) holds.elims(2) prod.collapse)
ultimately obtain  $C$  where
   $C$ :  $\langle \text{set } (\text{ptms } (\text{shd } \text{suf})) \subseteq \text{set } C \rangle$   $\langle \text{Neg } (\text{sub } 0 (\text{Fun } (\text{generateNew } C) [])) p \in \text{set } z' \rangle$ 
  using parts-in-effect [where  $B=B$  and  $z'=z'$  and  $z=\langle \text{pseq } (\text{shd } \text{suf}) \rangle$  and
 $r=\langle ?r \rangle$  and  $p=\langle \text{Neg } (\text{Exi } p) \rangle$ ]
  unfolding parts-def by auto
then have  $*$ :  $\langle \text{Neg } (\text{sub } 0 (\text{Fun } (\text{generateNew } C) [])) p \in ?H \rangle$ 
  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)
   $\text{sset-sdrop sset-shift stl-sset}$ 
let  $?t = \langle \text{Fun } (\text{generateNew } C) [] \rangle$ 
show  $\langle \exists t \in \text{terms } ?H. \text{Neg } (\text{sub } 0 t p) \in ?H \rangle$ 
proof (cases  $\langle ?t \in \text{set } (\text{subtermFm } (\text{Neg } (\text{sub } 0 ?t p))) \rangle$ )
  case True
  then have  $\langle ?t \in \text{terms } ?H \rangle$ 
  unfolding terms-def using  $*$  by (metis UN-I empty-iff)
then show  $?thesis$ 
  using  $*$  by blast

```

```

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

have  $\langle \text{affects } r \ ?f \longleftrightarrow r = ?r \rangle$  for  $r$ 
  unfolding affects-def by (cases  $r$ ) simp-all
then have  $\langle \text{list-all } (\text{not } (\lambda \text{step}. \text{affects } (\text{snd } \text{step}) \ ?f)) \ \text{pre} \rangle$ 
  using pre by simp
moreover have  $\langle \text{epath } (\text{pre } @- \ \text{suf}) \rangle$ 
  using  $\langle \text{epath steps} \rangle$  epath-sdrop ori by metis
ultimately have  $\langle ?f \in \text{set } (\text{pseq } (\text{shd } \ \text{suf})) \rangle$ 
  using epath-preserves-unaaffected n ori by metis

moreover have  $\langle \text{epath } \ \text{suf} \rangle$ 
  using  $\langle \text{epath } (\text{pre } @- \ \text{suf}) \rangle$  epath-sdrop by (metis alwD alw-iff-sdrop alw-shift)
then obtain  $B \ z' \ r'$  where
   $z'$ :  $\langle (B, z') \mid \in \text{effect } ?r \ (\text{ptms } (\text{shd } \ \text{suf}), \ \text{pseq } (\text{shd } \ \text{suf})) \rangle \langle \text{shd } (\text{stl } \ \text{suf}) = ((B, z'), r') \rangle$ 
  using suf epath-effect unfolding pseq-def ptms-def
  by (metis (mono-tags, lifting) holds.elims(2) prod.collapse)
ultimately have  $\langle p \in \text{set } z' \rangle$ 
  using parts-in-effect unfolding parts-def by fastforce

then show  $\langle p \in ?H \rangle$ 
  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**

$\langle \text{usemantics } u \ e \ f \ g \ (Pre \ i \ l) = g \ i \ (\text{semantics-list } e \ f \ l) \rangle$   
 $\langle \text{usemantics } u \ e \ f \ g \ (Imp \ p \ q) = (\text{usemantics } u \ e \ f \ g \ p \longrightarrow \text{usemantics } u \ e \ f \ g \ q) \rangle$   
 $\langle \text{usemantics } u \ e \ f \ g \ (Dis \ p \ q) = (\text{usemantics } u \ e \ f \ g \ p \ \vee \ \text{usemantics } u \ e \ f \ g \ q) \rangle$   
 $\langle \text{usemantics } u \ e \ f \ g \ (Con \ p \ q) = (\text{usemantics } u \ e \ f \ g \ p \ \wedge \ \text{usemantics } u \ e \ f \ g \ q) \rangle$   
 $\langle \text{usemantics } u \ e \ f \ g \ (Exi \ p) = (\exists x \in u. \ \text{usemantics } u \ (SeCaV.shift \ e \ 0 \ x) \ f \ g \ p) \rangle$   
 $\langle \text{usemantics } u \ e \ f \ g \ (Uni \ p) = (\forall x \in u. \ \text{usemantics } u \ (SeCaV.shift \ e \ 0 \ x) \ f \ g \ p) \rangle$   
 $\langle \text{usemantics } u \ e \ f \ g \ (Neg \ p) = (\neg \ \text{usemantics } u \ e \ f \ g \ p) \rangle$

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

**definition** *is-env* ::  $\langle 'a \ set \Rightarrow (nat \Rightarrow 'a) \Rightarrow bool \rangle$  **where**  
 $\langle is-env \ u \ e \equiv \forall n. \ e \ n \in u \rangle$

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

**definition** *is-fdenot* ::  $\langle 'a \ set \Rightarrow (nat \Rightarrow 'a \ list \Rightarrow 'a) \Rightarrow bool \rangle$  **where**  
 $\langle is-fdenot \ u \ f \equiv \forall i \ l. \ list-all \ (\lambda x. \ x \in u) \ l \longrightarrow f \ i \ l \in u \rangle$

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

**lemma** *usemantics-UNIV*:  $\langle \text{usemantics } UNIV \ e \ f \ g \ p \longleftrightarrow \text{semantics } e \ f \ g \ p \rangle$   
**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*]:  $\langle n \notin \text{params } p \Longrightarrow \text{usemantics } u \ e \ (f(n := x)) \ g \ p \longleftrightarrow \text{usemantics } u \ e \ f \ g \ p \rangle$   
**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*]:  
 $\langle \text{usemantics } u \ e \ f \ g \ (\text{subst } a \ t \ i) \longleftrightarrow \text{usemantics } u \ (SeCaV.shift \ e \ i \ (\text{semantics-term } e \ f \ t)) \ f \ g \ a \rangle$   
**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**  $\langle is-env\ u\ e \rangle \langle is-fdenot\ u\ f \rangle$

**shows**  $\langle semantics-term\ e\ f\ t \in u \rangle \langle list-all\ (\lambda x. x \in u)\ (semantics-list\ e\ f\ ts) \rangle$

**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*]:  $\langle is-fdenot\ u\ f \implies x \in u \implies is-fdenot\ u\ (f(i := \lambda-. x)) \rangle$

**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**  $\langle \Vdash z \rangle$  **and**  $\langle is-env\ u\ e \rangle$  **and**  $\langle is-fdenot\ u\ f \rangle$

**shows**  $\langle \exists p \in set\ z. usemantics\ u\ e\ f\ g\ p \rangle$

**using** *assms*

**proof** (*induct* *arbitrary: f* *rule: sequent-calculus.induct*)

**case** (*10* *i* *p* *z*)

**then show** *?case*

**proof** (*cases*  $\langle \forall x \in u. usemantics\ u\ e\ (f(i := \lambda-. x))\ g\ (sub\ 0\ (Fun\ i\ [])\ p) \rangle$ )

**case** *False*

**moreover have**  $\langle \forall x \in u. \exists p \in set\ (sub\ 0\ (Fun\ i\ [])\ p \# z). usemantics\ u\ e\ (f(i := \lambda-. x))\ g\ p \rangle$

**using** *10 is-fdenot-shift* **by** *metis*

**ultimately have**  $\langle \exists x \in u. \exists p \in set\ z. usemantics\ u\ e\ (f(i := \lambda-. x))\ g\ p \rangle$

**by** *fastforce*

**moreover have**  $\langle list-all\ (\lambda p. i \notin params\ p)\ z \rangle$

**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*  $\langle \forall x \in u. usemantics\ u\ e\ (f(i := \lambda-. x))\ g\ (Neg\ (sub\ 0\ (Fun\ i\ [])\ p)) \rangle$ )

**case** *False*

**moreover have**

```

  <∀ x ∈ u. ∃ p ∈ set (Neg (sub 0 (Fun i []) p) # z). usemantics u e (f(i := λ-.
x)) g p>
  using 11 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 (λp. i ∉ 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 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 ≡ Neg (Pre n ts) ∈ S>

```

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

```

abbreviation
  <E S n ≡ if Var n ∈ terms S then Var n else SOME t. t ∈ 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 ≡ if Fun i l ∈ terms S then Fun i l else SOME t. t ∈ 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 ≠ {}>
  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 ∈ terms S ⇒ t = Fun 0 [] ∨ (∃ p ∈ S. t ∈ 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*:  $\langle t \in \text{terms } S \implies \text{set } (\text{subtermTm } t) \subseteq \text{terms } S \rangle$

**proof** (*induct t*)

**case** (*Fun n ts*)

**moreover have**  $\langle \forall t \in \text{set } ts. t \in \text{set } ts \rangle$

**by** *simp*

**moreover have**  $\langle \forall t \in \text{set } ts. t \in \text{terms } S \rangle$

**proof**

**fix** *t*

**assume** \*:  $\langle t \in \text{set } ts \rangle$

**then show**  $\langle t \in \text{terms } S \rangle$

**proof** (*cases*  $\langle \text{terms } S = \{\text{Fun } 0 []\} \rangle$ )

**case** *True*

**then show** *?thesis*

**using** *Fun \** **by** *simp*

**next**

**case** *False*

**moreover obtain** *p* **where** *p*:  $\langle p \in S \rangle \langle \text{Fun } n \text{ ts} \in \text{set } (\text{subtermFm } p) \rangle$

**using** *Fun(2) terms-cases \** **by** *fastforce*

**then have**  $\langle \text{set } ts \subseteq \text{set } (\text{subtermFm } p) \rangle$

**using** *fun-arguments-subterm* **by** *blast*

**ultimately show**  $\langle t \in \text{terms } S \rangle$

**unfolding** *terms-def* **using** \* *p(1)* **by** (*metis UN-iff in-mono*)

**qed**

**qed**

**ultimately have**  $\langle \forall t \in \text{set } ts. \text{set } (\text{subtermTm } t) \subseteq \text{terms } S \rangle$

**using** *Fun* **by** *meson*

**moreover note**  $\langle \text{Fun } n \text{ ts} \in \text{terms } S \rangle$

**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*:

$\langle t \in \text{terms } S \implies \text{semantics-term } (E \ S) \ (F \ S) \ t = t \rangle$

$\langle \text{list-all } (\lambda t. t \in \text{terms } S) \ ts \implies \text{semantics-list } (E \ S) \ (F \ S) \ ts = ts \rangle$

**proof** (*induct t and ts arbitrary: ts rule: semantics-term.induct semantics-list.induct*)

**case** (*Fun i ts'*)

**moreover have**  $\langle \forall t' \in \text{set } ts'. t' \in \text{set } (\text{subtermTm } (\text{Fun } i \ ts')) \rangle$

**using** *subterm-Fun-refl* **by** *blast*

**ultimately have**  $\langle \text{list-all } (\lambda t. t \in \text{terms } S) \ ts' \rangle$

```

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:
   $\langle is\text{-env } (terms\ S) (E\ S) \rangle$ 
unfolding is-env-def
proof
  fix n
  show  $\langle E\ S\ n \in terms\ S \rangle$ 
  by (cases  $\langle Var\ n \in terms\ S \rangle$ ) (simp-all add: some-in-eq)
qed

```

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

```

lemma is-fdenot-F:
   $\langle is\text{-fdenot } (terms\ S) (F\ S) \rangle$ 
unfolding is-fdenot-def
proof (intro allI impI)
  fix i l
  assume  $\langle list\text{-all } (\lambda x. x \in terms\ S)\ l \rangle$ 
  then show  $\langle F\ S\ i\ l \in terms\ S \rangle$ 
  by (cases  $\langle \forall n. Var\ n \in terms\ S \rangle$ ) (simp-all add: some-in-eq)
qed

```

**abbreviation**

```

 $\langle M\ S \equiv usemantics\ (terms\ S)\ (E\ S)\ (F\ S)\ (G\ S) \rangle$ 

```

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  $\langle Hintikka\ S \rangle$ 
  shows  $\langle (p \in S \longrightarrow \neg M\ S\ p) \wedge (Neg\ p \in S \longrightarrow M\ S\ p) \rangle$ 
proof (induct p rule: wf-induct [where r=measure size])
  case 1
  then show ?case ..
next
  fix x
  assume wf:  $\langle \forall q. (q, x) \in measure\ size \longrightarrow$ 
     $(q \in S \longrightarrow \neg M\ S\ q) \wedge (Neg\ q \in S \longrightarrow M\ S\ q) \rangle$ 
  show  $\langle (x \in S \longrightarrow \neg M\ S\ x) \wedge (Neg\ x \in S \longrightarrow M\ S\ x) \rangle$ 
proof (cases x)
  case (Pre n ts)
  show ?thesis
proof (intro conjI impI)

```

```

assume  $\langle x \in S \rangle$ 
then have  $\langle \text{Neg } (Pre\ n\ ts) \notin S \rangle$ 
  using assms Pre Hintikka.Basic by blast
moreover have  $\langle \text{list-all } (\lambda t. t \in \text{terms } S) ts \rangle$ 
  using  $\langle x \in S \rangle$  Pre subterm-Pre-refl unfolding terms-def list-all-def by force
ultimately show  $\langle \neg M\ S\ x \rangle$ 
  using Pre usemantics-E
  by (metis (no-types, lifting) usemantics.simps(1))
next
assume  $\langle \text{Neg } x \in S \rangle$ 
then have  $\langle G\ S\ n\ ts \rangle$ 
  using assms Pre Hintikka.Basic by blast
moreover have  $\langle \text{list-all } (\lambda t. t \in \text{terms } S) ts \rangle$ 
  using  $\langle \text{Neg } x \in S \rangle$  Pre subterm-Pre-refl unfolding terms-def list-all-def by
force
ultimately show  $\langle M\ S\ x \rangle$ 
  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  $\langle x \in S \rangle$ 
  then have  $\langle \text{Neg } p \in S \rangle \langle q \in S \rangle$ 
    using Imp assms Hintikka.AlphaImp by blast+
  then show  $\langle \neg M\ S\ x \rangle$ 
    using wf Imp by fastforce
next
assume  $\langle \text{Neg } x \in S \rangle$ 
then have  $\langle p \in S \vee \text{Neg } q \in S \rangle$ 
  using Imp assms Hintikka.BetaImp by blast
then show  $\langle M\ S\ x \rangle$ 
  using wf Imp by fastforce
qed
next
case (Dis p q)
show ?thesis
proof (intro conjI impI)
  assume  $\langle x \in S \rangle$ 
  then have  $\langle p \in S \rangle \langle q \in S \rangle$ 
    using Dis assms Hintikka.AlphaDis by blast+
  then show  $\langle \neg M\ S\ x \rangle$ 
    using wf Dis by fastforce
next
assume  $\langle \text{Neg } x \in S \rangle$ 
then have  $\langle \text{Neg } p \in S \vee \text{Neg } q \in S \rangle$ 
  using Dis assms Hintikka.BetaDis by blast
then show  $\langle M\ S\ x \rangle$ 

```

```

    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 ∈ terms S⟩ by auto
  qed
next
case (Uni p)
show ?thesis
proof (intro conjI impI)
  assume ⟨x ∈ S⟩
  then obtain t where ⟨t ∈ terms S⟩ ⟨sub 0 t p ∈ S⟩
    using Uni assms Hintikka.DeltaUni by metis
  then have ⟨¬ 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 ∈ terms S⟩ usemantics-E(1) terms-downwards-closed unfolding
list-all-def by blast
  ultimately show ⟨¬ M S x⟩
    using Uni ⟨t ∈ terms S⟩ by auto
next
assume ⟨Neg x ∈ S⟩
then have ⟨∀ t ∈ terms S. Neg (sub 0 t p) ∈ S⟩
  using Uni assms Hintikka.GammaUni by blast
then have ⟨∀ t ∈ 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 ⟨∀ 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) (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 ∈ S⟩
  then show ⟨¬ M S x⟩
    using wf Neg by fastforce
next
assume ⟨Neg x ∈ S⟩
then have ⟨p ∈ 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  $\langle \forall z' \in \text{set } (\text{children } A \ r \ z). (\Vdash \text{pre } @ \ z') \rangle$ 
  and  $\langle \text{paramss } (\text{pre } @ \ z) \subseteq \text{paramsts } A \rangle$ 
  shows  $\langle \Vdash \text{pre } @ \ z \rangle$ 
  using assms
proof (induct z arbitrary: pre A)
  case Nil
  then show ?case
    by simp
next
  case (Cons p z)
  then have ih:  $\langle \forall z' \in \text{set } (\text{children } A \ r \ z). (\Vdash \text{pre } @ \ z') \implies (\Vdash \text{pre } @ \ z) \rangle$ 
    if  $\langle \text{paramss } (\text{pre } @ \ z) \subseteq \text{paramsts } A \rangle$  for pre A
    using that by simp

  let ?A =  $\langle \text{remdups } (A \ @ \ \text{subtermFms } (\text{concat } (\text{parts } A \ r \ p))) \rangle$ 

  have A:  $\langle \text{paramss } (\text{pre } @ \ p \ # \ z) \subseteq \text{paramsts } ?A \rangle$ 
    using paramsts-subset Cons.premis(2) by fastforce

  have  $\langle \forall z' \in \text{set } (\text{list-prod } (\text{parts } A \ r \ p) (\text{children } ?A \ r \ z)). (\Vdash \text{pre } @ \ z') \rangle$ 
    using Cons.premis by (metis children.simps(2))
  then have  $\langle \forall z' \in \{hs \ @ \ ts \mid hs \ ts. hs \in \text{set } (\text{parts } A \ r \ p) \wedge ts \in \text{set } (\text{children } ?A \ r \ z)\}. (\Vdash \text{pre } @ \ z') \rangle$ 
    using list-prod-is-cartesian by blast
  then have *:
     $\langle \forall hs \in \text{set } (\text{parts } A \ r \ p). \forall ts \in \text{set } (\text{children } ?A \ r \ z). (\Vdash \text{pre } @ \ hs \ @ \ ts) \rangle$ 
    by blast
  then show ?case

```

```

proof (cases r p rule: parts-exhaust)
  case (AlphaDis p q)
  then have  $\langle \forall z' \in \text{set } (\text{children } ?A \ r \ z). (\Vdash \text{pre } @ \ p \ \# \ q \ \# \ z') \rangle$ 
    using * unfolding parts-def by simp
  then have  $\langle \forall z' \in \text{set } (\text{children } ?A \ r \ z). (\Vdash (\text{pre } @ \ [p, q]) \ @ \ z') \rangle$ 
    by simp
  then have  $\langle \Vdash \text{pre } @ \ p \ \# \ q \ \# \ z \rangle$ 
    using AlphaDis ih[where pre= $\langle \text{pre } @ \ [p, q] \rangle$  and A= $\langle ?A \rangle$ ] A by simp
  then have  $\langle \Vdash p \ \# \ q \ \# \ \text{pre } @ \ z \rangle$ 
    using Ext by simp
  then have  $\langle \Vdash \text{Dis } p \ q \ \# \ \text{pre } @ \ z \rangle$ 
    using SeCaV.AlphaDis by blast
  then show ?thesis
    using AlphaDis Ext by simp
next
  case (AlphaImp p q)
  then have  $\langle \forall z' \in \text{set } (\text{children } ?A \ r \ z). (\Vdash \text{pre } @ \ \text{Neg } p \ \# \ q \ \# \ z') \rangle$ 
    using * unfolding parts-def by simp
  then have  $\langle \forall z' \in \text{set } (\text{children } ?A \ r \ z). (\Vdash (\text{pre } @ \ [\text{Neg } p, q]) \ @ \ z') \rangle$ 
    by simp
  then have  $\langle \Vdash \text{pre } @ \ \text{Neg } p \ \# \ q \ \# \ z \rangle$ 
    using AlphaImp ih[where pre= $\langle \text{pre } @ \ [\text{Neg } p, q] \rangle$  and A= $\langle ?A \rangle$ ] A by simp
  then have  $\langle \Vdash \text{Neg } p \ \# \ q \ \# \ \text{pre } @ \ z \rangle$ 
    using Ext by simp
  then have  $\langle \Vdash \text{Imp } p \ q \ \# \ \text{pre } @ \ z \rangle$ 
    using SeCaV.AlphaImp by blast
  then show ?thesis
    using AlphaImp Ext by simp
next
  case (AlphaCon p q)
  then have  $\langle \forall z' \in \text{set } (\text{children } ?A \ r \ z). (\Vdash \text{pre } @ \ \text{Neg } p \ \# \ \text{Neg } q \ \# \ z') \rangle$ 
    using * unfolding parts-def by simp
  then have  $\langle \forall z' \in \text{set } (\text{children } ?A \ r \ z). (\Vdash (\text{pre } @ \ [\text{Neg } p, \text{Neg } q]) \ @ \ z') \rangle$ 
    by simp
  then have  $\langle \Vdash \text{pre } @ \ \text{Neg } p \ \# \ \text{Neg } q \ \# \ z \rangle$ 
    using AlphaCon ih[where pre= $\langle \text{pre } @ \ [\text{Neg } p, \text{Neg } q] \rangle$  and A= $\langle ?A \rangle$ ] A by
simp
  then have  $\langle \Vdash \text{Neg } p \ \# \ \text{Neg } q \ \# \ \text{pre } @ \ z \rangle$ 
    using Ext by simp
  then have  $\langle \Vdash \text{Neg } (\text{Con } p \ q) \ \# \ \text{pre } @ \ z \rangle$ 
    using SeCaV.AlphaCon by blast
  then show ?thesis
    using AlphaCon Ext by simp
next
  case (BetaCon p q)
  then have
 $\langle \forall z' \in \text{set } (\text{children } ?A \ r \ z). (\Vdash \text{pre } @ \ p \ \# \ z') \rangle$ 
 $\langle \forall z' \in \text{set } (\text{children } ?A \ r \ z). (\Vdash \text{pre } @ \ q \ \# \ z') \rangle$ 
    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.premst(2) news-paramst 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.premst(2) news-paramst 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 GammaExt 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**  $\langle \forall z' \in \text{set } (\text{children } A \ r \ z). (\Vdash z') \rangle$  **and**  $\langle \text{paramss } z \subseteq \text{paramsts } A \rangle$   
**shows**  $\langle \Vdash z \rangle$   
**using** *SeCaV-children-pre assms* **by** (*metis append-Nil*)

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

**interpretation** *Soundness eff rules UNIV*  $\langle \lambda - (A, z). (\Vdash z) \rangle$

**unfolding** *Soundness-def*

**proof** *safe*

**fix**  $r \ A \ z \ ss \ S$

**assume**  $r\text{-enabled}: \langle \text{eff } r \ (A, z) \ ss \rangle$

**assume**  $\langle \forall s'. s' \mid \in \mid ss \longrightarrow (\forall S \in UNIV. \text{case } s' \text{ of } (A, z) \Rightarrow \Vdash z) \rangle$

**then have** *next-sound*:  $\langle \forall B \ z. (B, z) \mid \in \mid ss \longrightarrow (\Vdash z) \rangle$

**by** *simp*

**show**  $\langle \Vdash z \rangle$

**proof** (*cases*  $\langle \text{branchDone } z \rangle$ )

**case** *True*

**then obtain**  $p$  **where**  $\langle p \in \text{set } z \rangle \ \langle \text{Neg } p \in \text{set } z \rangle$

**using** *branchDone-contradiction* **by** *blast*

**then show** *?thesis*

**using** *Ext Basic* **by** *fastforce*

**next**

**case** *False*

**let**  $?A = \langle \text{remdups } (A \ @ \ \text{subtermFms } z) \rangle$

**have**  $\langle \forall z' \in \text{set } (\text{children } ?A \ r \ z). (\Vdash z') \rangle$

**using** *False r-enabled eff-children next-sound* **by** *blast*

**moreover have**  $\langle \text{set } (\text{subtermFms } z) \subseteq \text{set } ?A \rangle$

**by** *simp*

**then have**  $\langle \text{paramss } z \subseteq \text{paramsts } ?A \rangle$

**using** *subtermFm-subset-params* **by** *fastforce*

**ultimately show**  $\langle \Vdash z \rangle$

**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**  $\langle \text{tfinite } t \rangle$  **and**  $\langle \text{wf } t \rangle$

**shows**  $\langle \Vdash \text{rootSequent } t \rangle$

**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  $A :: \langle tm\ list \rangle$  and  $z :: \langle fm\ list \rangle$ 
  defines  $\langle t \equiv \text{secavProver } (A, z) \rangle$ 
  shows  $\langle \text{fst } (\text{root } t) = (A, z) \wedge \text{wf } t \wedge \text{tfinite } t \rangle \vee$ 
     $\langle \exists \text{ steps. } \text{fst } (\text{shd } \text{steps}) = (A, z) \wedge \text{epath } \text{steps} \wedge \text{Saturated } \text{steps} \rangle$ 
  (is  $\langle ?A \vee ?B \rangle$ 
proof –
  { assume  $\langle \neg ?A \rangle$ 
    with assms have  $\langle \neg \text{tfinite } (\text{mkTree rules } (A, z)) \rangle$ 
    unfolding secavProver-def using wf-mkTree fair-rules by simp
    then obtain steps where  $\langle \text{ipath } (\text{mkTree rules } (A, z)) \text{ steps} \rangle$  using Konig by blast
    with assms have  $\langle \text{fst } (\text{shd } \text{steps}) = (A, z) \wedge \text{epath } \text{steps} \wedge \text{Saturated } \text{steps} \rangle$ 
    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

```

 $\langle \text{uvalid } z \equiv \forall u (e :: \text{nat} \Rightarrow \text{tm}) f g. \text{is-env } u e \longrightarrow \text{is-fdenot } u f \longrightarrow$ 
   $(\exists p \in \text{set } z. \text{usemantics } u e f g p) \rangle$ 

```

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  $\langle \text{fst } (\text{shd } \text{steps}) = (A, z) \rangle$  and  $\langle \text{epath } \text{steps} \rangle$  and  $\langle \text{Saturated } \text{steps} \rangle$ 
shows  $\langle \neg \text{uvalid } z \rangle$ 
proof
  assume  $\langle \text{uvalid } z \rangle$ 
  moreover have  $\langle \text{Hintikka } (\text{tree-fms } \text{steps}) \rangle$  (is  $\langle \text{Hintikka } ?S \rangle$ 

```

```

    using assms escape-path-Hintikka assms by simp
  moreover have  $\langle \forall p \in \text{set } z. p \in \text{tree-fms steps} \rangle$ 
  using assms shd-sset by (metis Pair-inject prod.collapse pseq-def pseq-in-tree-fms)
  then have  $\langle \exists g. \forall p \in \text{set } z. \neg \text{usemantics } (\text{terms } ?S) (E ?S) (F ?S) g p \rangle$ 
    using calculation(2) Hintikka-counter-model assms by blast
  moreover have  $\langle \text{is-env } (\text{terms } ?S) (E ?S) \rangle \langle \text{is-fdenot } (\text{terms } ?S) (F ?S) \rangle$ 
    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 ::  $\langle \text{tm list} \rangle$ 
  assumes  $\langle \text{uvalid } z \rangle$ 
  defines  $\langle t \equiv \text{secavProver } (A, z) \rangle$ 
  shows  $\langle \text{fst } (\text{root } t) = (A, z) \wedge \text{wf } t \wedge \text{tfinite } t \rangle$ 
  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 ::  $\langle \text{tm list} \rangle$ 
  assumes  $\langle \Vdash z \rangle$ 
  defines  $\langle t \equiv \text{secavProver } (A, z) \rangle$ 
  shows  $\langle \text{fst } (\text{root } t) = (A, z) \wedge \text{wf } t \wedge \text{tfinite } t \rangle$ 
proof –
  have  $\langle \text{uvalid } z \rangle$ 
    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**  $\langle tfinite\ t \rangle \langle wf\ t \rangle \langle is-env\ u\ e \rangle \langle is-fdenot\ u\ f \rangle$   
**shows**  $\langle \exists p \in set\ (rootSequent\ t).\ usemantics\ u\ e\ f\ g\ p \rangle$   
**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**  $A :: \langle tm\ list \rangle$  **and**  $z :: \langle fm\ list \rangle$   
**defines**  $\langle t \equiv secavProver\ (A,\ z) \rangle$   
**shows**  $\langle tfinite\ t \wedge wf\ t \longleftrightarrow uvalid\ z \rangle$   
**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**  $\langle t \equiv secavProver\ ([],\ [p]) \rangle$   
**shows**  $\langle tfinite\ t \wedge wf\ t \longleftrightarrow uvalid\ [p] \rangle$   
**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**  $A :: \langle tm\ list \rangle$  **and**  $z :: \langle fm\ list \rangle$   
**defines**  $\langle t \equiv secavProver\ (A,\ z) \rangle$   
**shows**  $\langle tfinite\ t \wedge wf\ t \longleftrightarrow (\vdash\ z) \rangle$   
**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**  $\langle t \equiv secavProver\ ([],\ [p]) \rangle$   
**shows**  $\langle tfinite\ t \wedge wf\ t \longleftrightarrow (\vdash\ [p]) \rangle$   
**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**  $\langle tfinite\ t \rangle \langle wf\ t \rangle$   
**shows**  $\langle \exists p \in set\ (rootSequent\ t). semantics\ e\ f\ g\ p \rangle$   
**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**  $\langle tfinite\ t \rangle \langle wf\ t \rangle \langle snd\ (fst\ (root\ t)) = [p] \rangle$   
**shows**  $\langle semantics\ e\ f\ g\ p \rangle$   
**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**  $A :: \langle tm\ list \rangle$   
**assumes**  $\langle \forall (e :: nat \Rightarrow nat\ hterm)\ f\ g.\ semantics\ e\ f\ g\ p \rangle$   
**defines**  $\langle t \equiv secavProver\ (A, [p]) \rangle$   
**shows**  $\langle fst\ (root\ t) = (A, [p]) \wedge wf\ t \wedge tfinite\ t \rangle$

**proof** –

**have**  $\langle \Vdash [p] \rangle$   
**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**  $A :: \langle tm\ list \rangle$  **and**  $p :: fm$   
**defines**  $\langle t \equiv secavProver\ (A, [p]) \rangle$   
**shows**  $\langle tfinite\ t \wedge wf\ t \iff (\forall (e :: nat \Rightarrow nat\ hterm)\ f\ g.\ semantics\ e\ f\ g\ p) \rangle$   
**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:*

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

**end**