

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

1	SeCaV	2
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	5
1.7.3	Parameters: Formulas	6
1.7.4	Update Lemmas	6
1.7.5	Substitution	7
1.7.6	Auxiliary Lemmas	8
1.7.7	Soundness	8
1.8	Reference	8
1.9	Appendix: Completeness	9
1.10	Reference	10
2	The prover	12
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	20
2.3.4	Custom cases	21
2.3.5	Unaffected formulas	22

2.3.6	Affected formulas	23
2.3.7	Generating new function names	24
2.3.8	Finding axioms	24
2.3.9	Subterms	24
2.4	Hintikka sets for SeCaV	25
2.5	Escape path formulas are Hintikka	26
2.5.1	Definitions	26
2.5.2	Facts about streams	26
2.5.3	Transformation of states on an escape path	27
2.5.4	Preservation of formulas on escape paths	27
2.5.5	Formulas on an escape path form a Hintikka set	28
2.6	Bounded semantics	29
2.7	Countermodels from Hintikka sets	31
2.8	Soundness	32
2.9	Completeness	33
2.10	Results	34
2.10.1	Alternate semantics	35
2.10.2	SeCaV	35
2.10.3	Semantics	35

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

⟨proof⟩

lemma $\langle \neg$ semantics $e f g (Neg (Imp (Pre 0 [])) (Pre 0 [])) \rangle$
⟨proof⟩

1.4 Auxiliary Functions

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

⟨new-term $c (Var n) = True$ ⟩ |
⟨new-term $c (Fun i l) = (if i = c then False else new-list c l)$ ⟩ |
⟨new-list $c [] = True$ ⟩ |
⟨new-list $c (t \# l) = (if new-term c t then new-list c l else False)$ ⟩

primrec *new* where

⟨new $c (Pre i l) = new-list c l$ ⟩ |
⟨new $c (Imp p q) = (if new c p then new c q else False)$ ⟩ |
⟨new $c (Dis p q) = (if new c p then new c q else False)$ ⟩ |
⟨new $c (Con p q) = (if new c p then new c q else False)$ ⟩ |
⟨new $c (Exi p) = new c p$ ⟩ |
⟨new $c (Uni p) = new c p$ ⟩ |
⟨new $c (Neg p) = new c p$ ⟩

primrec *news* where

⟨news $c [] = True$ ⟩ |
⟨news $c (p \# z) = (if new c p then news c z else False)$ ⟩

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

⟨inc-term $(Var n) = Var (n + 1)$ ⟩ |
⟨inc-term $(Fun i l) = Fun i (inc-list l)$ ⟩ |
⟨inc-list $[] = []$ ⟩ |
⟨inc-list $(t \# l) = inc-term t \# inc-list l$ ⟩

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

⟨sub-term $v s (Var n) = (if n < v then Var n else if n = v then s else Var (n - 1))$ ⟩ |
⟨sub-term $v s (Fun i l) = Fun i (sub-list v s l)$ ⟩ |
⟨sub-list $v s [] = []$ ⟩ |
⟨sub-list $v s (t \# l) = sub-term v s t \# sub-list v s l$ ⟩

primrec *sub* where

⟨sub $v s (Pre i l) = Pre i (sub-list v s l)$ ⟩ |
⟨sub $v s (Imp p q) = Imp (sub v s p) (sub v s q)$ ⟩ |
⟨sub $v s (Dis p q) = Dis (sub v s p) (sub v s q)$ ⟩ |
⟨sub $v s (Con p q) = Con (sub v s p) (sub v s q)$ ⟩ |
⟨sub $v s (Exi p) = Exi (sub (v + 1) (inc-term s) p)$ ⟩ |
⟨sub $v s (Uni p) = Uni (sub (v + 1) (inc-term s) p)$ ⟩ |
⟨sub $v s (Neg p) = Neg (sub v s p)$ ⟩

primrec *member* where

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

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]*: $\langle \text{member } p z \longleftrightarrow p \in \text{set } z \rangle$
 $\langle \text{proof} \rangle$

lemma *ext [iff]*: $\langle \text{ext } y z \longleftrightarrow \text{set } z \subseteq \text{set } y \rangle$
 $\langle \text{proof} \rangle$

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 $\langle \vdash [\text{Imp } (\text{Pre } 0 []) (\text{Pre } 0 [])] \rangle$
 $\langle \text{proof} \rangle$

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 $\langle \Vdash$

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

1.7 Appendix: Soundness

1.7.1 Increment Function

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

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

lemma $p0$ [simp]: $\langle \text{paramsts } ts = \bigcup (\text{set } (\text{map } \text{paramst } ts)) \rangle$
 $\langle \text{proof} \rangle$

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

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

lemma $p1$ [simp]: $\langle \text{paramst}' t = \text{paramst } t \rangle$
 $\langle \text{proof} \rangle$

1.7.3 Parameters: Formulas

primrec *params* :: $\langle fm \Rightarrow nat\ set \rangle$ **where**
 $\langle params\ (Pre\ b\ ts) = paramsts\ ts \rangle$ |
 $\langle params\ (Imp\ p\ q) = params\ p \cup params\ q \rangle$ |
 $\langle params\ (Dis\ p\ q) = params\ p \cup params\ q \rangle$ |
 $\langle params\ (Con\ p\ q) = params\ p \cup params\ q \rangle$ |
 $\langle params\ (Exi\ p) = params\ p \rangle$ |
 $\langle params\ (Uni\ p) = params\ p \rangle$ |
 $\langle params\ (Neg\ p) = params\ p \rangle$

primrec *params'* :: $\langle fm \Rightarrow nat\ set \rangle$ **where**
 $\langle params'\ (Pre\ b\ ts) = \bigcup (set\ (map\ paramst'\ ts)) \rangle$ |
 $\langle params'\ (Imp\ p\ q) = params'\ p \cup params'\ q \rangle$ |
 $\langle params'\ (Dis\ p\ q) = params'\ p \cup params'\ q \rangle$ |
 $\langle params'\ (Con\ p\ q) = params'\ p \cup params'\ q \rangle$ |
 $\langle params'\ (Exi\ p) = params'\ p \rangle$ |
 $\langle params'\ (Uni\ p) = params'\ p \rangle$ |
 $\langle params'\ (Neg\ p) = params'\ p \rangle$

lemma *p2* [*simp*]: $\langle params'\ p = params\ p \rangle$
 $\langle proof \rangle$

fun *paramst''* :: $\langle tm \Rightarrow nat\ set \rangle$ **where**
 $\langle paramst''\ (Var\ n) = \{ \} \rangle$ |
 $\langle paramst''\ (Fun\ a\ ts) = \{ a \} \cup (\bigcup t \in set\ ts.\ paramst''\ t) \rangle$

lemma *p1'* [*simp*]: $\langle paramst''\ t = paramst\ t \rangle$
 $\langle proof \rangle$

fun *params''* :: $\langle fm \Rightarrow nat\ set \rangle$ **where**
 $\langle params''\ (Pre\ b\ ts) = (\bigcup t \in set\ ts.\ paramst''\ t) \rangle$ |
 $\langle params''\ (Imp\ p\ q) = params''\ p \cup params''\ q \rangle$ |
 $\langle params''\ (Dis\ p\ q) = params''\ p \cup params''\ q \rangle$ |
 $\langle params''\ (Con\ p\ q) = params''\ p \cup params''\ q \rangle$ |
 $\langle params''\ (Exi\ p) = params''\ p \rangle$ |
 $\langle params''\ (Uni\ p) = params''\ p \rangle$ |
 $\langle params''\ (Neg\ p) = params''\ p \rangle$

lemma *p2'* [*simp*]: $\langle params''\ p = params\ p \rangle$
 $\langle proof \rangle$

1.7.4 Update Lemmas

lemma *upd-lemma'* [*simp*]:
 $\langle n \notin paramst\ t \implies semantics\ term\ e\ (f(n := z))\ t = semantics\ term\ e\ f\ t \rangle$
 $\langle n \notin paramsts\ ts \implies semantics\ list\ e\ (f(n := z))\ ts = semantics\ list\ e\ f\ ts \rangle$
 $\langle proof \rangle$

lemma *upd-lemma* [*iff*]: $\langle n \notin params\ p \implies semantics\ e\ (f(n := z))\ g\ p \longleftrightarrow$

semantics e f g p
 ⟨proof⟩

1.7.5 Substitution

primrec *subst* :: ⟨*tm* ⇒ *tm* ⇒ *nat* ⇒ *tm*⟩ and *substts* :: ⟨*tm list* ⇒ *tm* ⇒ *nat* ⇒ *tm list*⟩ **where**

⟨*subst* (*Var i*) *s k* = (if *k* < *i* then *Var (i - 1)* else if *i* = *k* then *s* else *Var i*)⟩ |
 ⟨*subst* (*Fun a ts*) *s k* = *Fun a (substts ts s k)*⟩ |
 ⟨*substts* [] *s k* = []⟩ |
 ⟨*substts* (*t # ts*) *s k* = *subst t s k # substts ts s k*⟩

primrec *subst* :: ⟨*fm* ⇒ *tm* ⇒ *nat* ⇒ *fm*⟩ **where**

⟨*subst* (*Pre b ts*) *s k* = *Pre b (substts ts s k)*⟩ |
 ⟨*subst* (*Imp p q*) *s k* = *Imp (subst p s k) (subst q s k)*⟩ |
 ⟨*subst* (*Dis p q*) *s k* = *Dis (subst p s k) (subst q s k)*⟩ |
 ⟨*subst* (*Con p q*) *s k* = *Con (subst p s k) (subst q s k)*⟩ |
 ⟨*subst* (*Exi p*) *s k* = *Exi (subst p (liftt s) (Suc k))*⟩ |
 ⟨*subst* (*Uni p*) *s k* = *Uni (subst p (liftt s) (Suc k))*⟩ |
 ⟨*subst* (*Neg p*) *s k* = *Neg (subst p s k)*⟩

lemma *shift-eq* [*simp*]: ⟨*i* = *j* ⇒ (shift *e i T*) *j* = *T*⟩ **for** *i* :: *nat*
 ⟨proof⟩

lemma *shift-gt* [*simp*]: ⟨*j* < *i* ⇒ (shift *e i T*) *j* = *e j*⟩ **for** *i* :: *nat*
 ⟨proof⟩

lemma *shift-lt* [*simp*]: ⟨*i* < *j* ⇒ (shift *e i T*) *j* = *e (j - 1)*⟩ **for** *i* :: *nat*
 ⟨proof⟩

lemma *shift-commute* [*simp*]: ⟨shift (shift *e i U*) 0 *T* = shift (shift *e* 0 *T*) (Suc *i*) *U*⟩
 ⟨proof⟩

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

⟨*semantics-term e f (subst t u i)* = *semantics-term (shift e i (semantics-term e f u)) f t*⟩
 ⟨*semantics-list e f (substts ts u i)* = *semantics-list (shift e i (semantics-term e f u)) f ts*⟩
 ⟨proof⟩

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

⟨*semantics-term (shift e 0 x) f (liftt t)* = *semantics-term e f t*⟩
 ⟨*semantics-list (shift e 0 x) f (liftts ts)* = *semantics-list e f ts*⟩
 ⟨proof⟩

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

⟨*semantics e f g (subst a t i)* ⇔ *semantics (shift e i (semantics-term e f t)) f g a*⟩

<proof>

1.7.6 Auxiliary Lemmas

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

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

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

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

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

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

1.7.7 Soundness

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

corollary $\langle \vdash z \implies \exists p. \text{ member } p \ z \wedge \text{ semantics } e \ f \ g \ p \rangle$
<proof>

corollary $\langle \vdash [p] \implies \text{ semantics } e \ f \ g \ p \rangle$
<proof>

corollary $\langle \neg (\vdash []) \rangle$
<proof>

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**

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

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

lemma *paramst* [*simp*]:

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

$\langle \text{proof} \rangle$

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

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

lemma *liftt* [*simp*]:
 $\langle \text{to-tm } (\text{FOL-Fitting.liftt } t) = \text{inc-term } (\text{to-tm } t) \rangle$
 $\langle \text{to-tm-list } (\text{FOL-Fitting.liftts } l) = \text{inc-list } (\text{to-tm-list } l) \rangle$
 $\langle \text{proof} \rangle$

lemma *substt* [*simp*]:
 $\langle \text{to-tm } (\text{FOL-Fitting.substt } t \ s \ v) = \text{sub-term } v \ (\text{to-tm } s) \ (\text{to-tm } t) \rangle$
 $\langle \text{to-tm-list } (\text{FOL-Fitting.substts } l \ s \ v) = \text{sub-list } v \ (\text{to-tm } s) \ (\text{to-tm-list } l) \rangle$
 $\langle \text{proof} \rangle$

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

lemma *sim*: $\langle (\vdash x) \implies (\Vdash (\text{map to-fm } x)) \rangle$
 $\langle \text{proof} \rangle$

lemma *evalt* [*simp*]:
 $\langle \text{semantics-term } e \ f \ t = \text{evalt } e \ f \ (\text{from-tm } t) \rangle$
 $\langle \text{semantics-list } e \ f \ ts = \text{evalts } e \ f \ (\text{from-tm-list } ts) \rangle$
 $\langle \text{proof} \rangle$

lemma *shift* [*simp*]: $\langle \text{shift } e \ 0 \ x = e(0;x) \rangle$
 $\langle \text{proof} \rangle$

lemma *semantics* [*iff*]: $\langle \text{semantics } e \ f \ g \ p \longleftrightarrow \text{eval } e \ f \ g \ (\text{from-fm } p) \rangle$
 $\langle \text{proof} \rangle$

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*: $\langle \gg p \implies \Vdash [p] \rangle \langle \Vdash [q] \implies \text{semantics } e \ f \ g \ q \rangle$
 $\langle \text{proof} \rangle$

corollary $\langle (\gg p) \longleftrightarrow (\Vdash [p]) \rangle$
 $\langle \text{proof} \rangle$

1.10 Reference

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

end

Chapter 2

The prover

2.1 Proof search procedure

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

This theory defines the actual proof search procedure.

2.1.1 Datatypes

A sequent is a list of formulas

```
type-synonym sequent = <fm list>
```

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

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

2.1.2 Auxiliary functions

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

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

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

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 $\langle rulesList \equiv [$
 $\ NegNeg, AlphaImp, AlphaDis, AlphaCon,$
 $\ DeltaExi, DeltaUni,$
 $\ BetaImp, BetaDis, BetaCon,$
 $\ GammaExi, GammaUni$
 $\] \rangle$

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

definition *rules* **where**
 $\langle rules = cycle\ rulesList \rangle$

2.1.5 Abstract completeness

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

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

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*: $\langle \forall st.\ \forall r \in i.R\ (cycle\ rulesList).\ \exists sl.\ eff\ r\ st\ sl \rangle$
 $\langle proof \rangle$

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

interpretation *RuleSystem* *eff* *rules* *UNIV*
 $\langle proof \rangle$

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*: $\langle \forall r. r \in R \longrightarrow per\ r \rangle$
<proof>

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

interpretation *PersistentRuleSystem eff rules UNIV*
<proof>

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

definition *<secavProver \equiv mkTree rules>*

abbreviation *<rootSequent t \equiv snd (fst (root t))>*

end

2.2 Export

theory *Export*
imports *Prover*
begin

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

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

code-lazy-type *stream*

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

declare *Stream.smember-code* [*code del*]
lemma [*code*]: *Stream.smember x (y ## s) = (x = y \vee Stream.smember x s)*
<proof>

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*)
<module MaybeExt(fromJust, isNothing) where
import Data.Maybe(fromJust, isNothing);>

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

```

code-identifier
  code-module Stream  $\rightarrow$  (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 $\langle \text{secavTreeCode} \equiv i.mkTree (\lambda r s. \text{Some } (effect\ r\ s))\ rules \rangle$

definition $\langle \text{secavProverCode} \equiv \lambda z. \text{secavTreeCode } ([], z) \rangle$

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-lifft* [*simp*]:

```

 $\langle paramst\ (lifft\ t) = paramst\ t \rangle$ 
 $\langle paramsts\ (liffts\ ts) = paramsts\ ts \rangle$ 
 $\langle proof \rangle$ 

```

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

lemma *paramst-sub-term*:

```

 $\langle paramst\ (sub-term\ m\ s\ t) \subseteq paramst\ s \cup paramst\ t \rangle$ 
 $\langle paramsts\ (sub-list\ m\ s\ l) \subseteq paramst\ s \cup paramsts\ l \rangle$ 
 $\langle proof \rangle$ 

```

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

lemma *params-sub*: $\langle \text{params } (sub\ m\ t\ p) \subseteq \text{paramst } t \cup \text{params } p \rangle$
 $\langle \text{proof} \rangle$

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*: $\langle \text{news } i\ z \longleftrightarrow i \notin \text{paramss } z \rangle$
 $\langle \text{proof} \rangle$

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

lemma *paramsts-subset*: $\langle \text{set } A \subseteq \text{set } B \implies \text{paramsts } A \subseteq \text{paramsts } B \rangle$
 $\langle \text{proof} \rangle$

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

lemma *size-sub [simp]*: $\langle \text{size } (sub\ i\ t\ p) = \text{size } p \rangle$
 $\langle \text{proof} \rangle$

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**
 $\langle \text{rule-index } \text{NegNeg} = 0 \rangle$
 $| \langle \text{rule-index } \text{AlphaImp} = 1 \rangle$
 $| \langle \text{rule-index } \text{AlphaDis} = 2 \rangle$
 $| \langle \text{rule-index } \text{AlphaCon} = 3 \rangle$
 $| \langle \text{rule-index } \text{DeltaExi} = 4 \rangle$
 $| \langle \text{rule-index } \text{DeltaUni} = 5 \rangle$
 $| \langle \text{rule-index } \text{BetaImp} = 6 \rangle$

```

| ⟨rule-index BetaDis = 7⟩
| ⟨rule-index BetaCon = 8⟩
| ⟨rule-index GammaExi = 9⟩
| ⟨rule-index GammaUni = 10⟩

```

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

lemma *distinct-rulesList*: ⟨distinct rulesList⟩
 ⟨proof⟩

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

lemma *cycle-nth*: ⟨ $xs \neq [] \implies cycle\ xs \ !!\ n = xs \ !\ (n \ mod\ length\ xs)$ ⟩
 ⟨proof⟩

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

lemma *nth-rule-index*: ⟨rulesList ! (rule-index r) = r⟩
 ⟨proof⟩

lemma *rule-index-bnd*: ⟨rule-index r < length rulesList⟩
 ⟨proof⟩

lemma *unique-rule-index*:
assumes ⟨ $n < length\ rulesList$ ⟩ ⟨rulesList ! $n = r$ ⟩
shows ⟨ $n = rule-index\ r$ ⟩
 ⟨proof⟩

The rule indices repeat in the stream each cycle.

lemma *rule-index-mod*:
assumes ⟨rules !! $n = r$ ⟩
shows ⟨ $n \ mod\ length\ rulesList = rule-index\ r$ ⟩
 ⟨proof⟩

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

lemma *mod-hit*:
fixes $k :: nat$
assumes ⟨ $0 < k$ ⟩
shows ⟨ $\forall i < k. \exists n > m. n \ mod\ k = i$ ⟩
 ⟨proof⟩

lemma *mod-suff*:
assumes ⟨ $\forall (n :: nat) > m. P\ (n \ mod\ k)$ ⟩ ⟨ $0 < k$ ⟩
shows ⟨ $\forall i < k. P\ i$ ⟩
 ⟨proof⟩

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

lemma *rules-repeat*: ⟨ $\exists n > m. rules \ !!\ n = r$ ⟩

⟨proof⟩

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

lemma *rules-repeat-sdrop*: $\langle \exists n. (sdrop\ k\ rules) !!\ n = r \rangle$
⟨proof⟩

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

lemma *fair-rules*: ⟨fair rules⟩
⟨proof⟩

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*: $\langle t \in set\ (subtermTm\ s) \implies set\ (subtermTm\ t) \subseteq set\ (subtermTm\ s) \rangle$
⟨proof⟩

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

lemma *sub-term-const-transfer*:
⟨sub-term $m\ (Fun\ a\ [])\ t \neq sub-term\ m\ s\ t \implies$
 $Fun\ a\ [] \in set\ (subtermTm\ (sub-term\ m\ (Fun\ a\ [])\ t)) \rangle$
⟨sub-list $m\ (Fun\ a\ [])\ ts \neq sub-list\ m\ s\ ts \implies$
 $Fun\ a\ [] \in (\bigcup t \in set\ (sub-list\ m\ (Fun\ a\ [])\ ts). set\ (subtermTm\ t)) \rangle$
⟨proof⟩

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

lemma *sub-const-transfer*:
assumes $\langle sub\ m\ (Fun\ a\ [])\ p \neq sub\ m\ t\ p \rangle$
shows $\langle Fun\ a\ [] \in set\ (subtermFm\ (sub\ m\ (Fun\ a\ [])\ p)) \rangle$
⟨proof⟩

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

lemma *set-subterms*:
fixes z
defines $\langle ts \equiv \bigcup p \in set\ z. set\ (subtermFm\ p) \rangle$
shows $\langle set\ (subterms\ z) = (if\ ts = \{\} then\ \{Fun\ 0\ []\} else\ ts) \rangle$
⟨proof⟩

The parameters and the subterm functions respect each other.

lemma *paramst-subtermTm*:

$\langle \forall i \in \text{paramst } t. \exists l. \text{Fun } i \ l \in \text{set } (\text{subtermTm } t) \rangle$
 $\langle \forall i \in \text{paramsts } ts. \exists l. \text{Fun } i \ l \in (\bigcup t \in \text{set } ts. \text{set } (\text{subtermTm } t)) \rangle$
 $\langle \text{proof} \rangle$

lemma *params-subtermFm*: $\langle \forall i \in \text{params } p. \exists l. \text{Fun } i \ l \in \text{set } (\text{subtermFm } p) \rangle$
 $\langle \text{proof} \rangle$

lemma *subtermFm-subset-params*: $\langle \text{set } (\text{subtermFm } p) \subseteq \text{set } A \implies \text{params } p \subseteq \text{paramsts } A \rangle$
 $\langle \text{proof} \rangle$

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

$\langle \bigwedge i \ ts. x = \text{Pre } i \ ts \implies P \rangle$
 $\langle \bigwedge p \ q. x = \text{Imp } p \ q \implies P \rangle$
 $\langle \bigwedge p \ q. x = \text{Dis } p \ q \implies P \rangle$
 $\langle \bigwedge p \ q. x = \text{Con } p \ q \implies P \rangle$
 $\langle \bigwedge p. x = \text{Exi } p \implies P \rangle$
 $\langle \bigwedge p. x = \text{Uni } p \implies P \rangle$
 $\langle \bigwedge i \ ts. x = \text{Neg } (\text{Pre } i \ ts) \implies P \rangle$
 $\langle \bigwedge p \ q. x = \text{Neg } (\text{Imp } p \ q) \implies P \rangle$
 $\langle \bigwedge p \ q. x = \text{Neg } (\text{Dis } p \ q) \implies P \rangle$
 $\langle \bigwedge p \ q. x = \text{Neg } (\text{Con } p \ q) \implies P \rangle$
 $\langle \bigwedge p. x = \text{Neg } (\text{Exi } p) \implies P \rangle$
 $\langle \bigwedge p. x = \text{Neg } (\text{Uni } p) \implies P \rangle$
 $\langle \bigwedge p. x = \text{Neg } (\text{Neg } p) \implies P \rangle$

shows P

$\langle \text{proof} \rangle$

lemma *parts-exhaust*

[*case-names AlphaDis AlphaImp AlphaCon BetaDis BetaImp BetaCon DeltaUni DeltaExi NegNeg GammaExi GammaUni Other*]:

assumes

$\langle \bigwedge p \ q. r = \text{AlphaDis} \implies x = \text{Dis } p \ q \implies P \rangle$
 $\langle \bigwedge p \ q. r = \text{AlphaImp} \implies x = \text{Imp } p \ q \implies P \rangle$
 $\langle \bigwedge p \ q. r = \text{AlphaCon} \implies x = \text{Neg } (\text{Con } p \ q) \implies P \rangle$
 $\langle \bigwedge p \ q. r = \text{BetaDis} \implies x = \text{Neg } (\text{Dis } p \ q) \implies P \rangle$
 $\langle \bigwedge p \ q. r = \text{BetaImp} \implies x = \text{Neg } (\text{Imp } p \ q) \implies P \rangle$
 $\langle \bigwedge p \ q. r = \text{BetaCon} \implies x = \text{Con } p \ q \implies P \rangle$
 $\langle \bigwedge p. r = \text{DeltaUni} \implies x = \text{Uni } p \implies P \rangle$
 $\langle \bigwedge p. r = \text{DeltaExi} \implies x = \text{Neg } (\text{Exi } p) \implies P \rangle$
 $\langle \bigwedge p. r = \text{NegNeg} \implies x = \text{Neg } (\text{Neg } p) \implies P \rangle$
 $\langle \bigwedge p. r = \text{GammaExi} \implies x = \text{Exi } p \implies P \rangle$
 $\langle \bigwedge p. r = \text{GammaUni} \implies x = \text{Neg } (\text{Uni } p) \implies P \rangle$

$\langle \forall A. \text{parts } A \ r \ x = [[x]] \implies P \rangle$
shows P
 $\langle \text{proof} \rangle$

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**

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

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 $\langle p \in \text{set } z' \rangle$
 $\langle \text{proof} \rangle$

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

lemma *list-prod-is-cartesian*:

$\langle \text{set } (\text{list-prod } \text{hs } \text{ts}) = \{h \ @ \ t \mid h \ t. h \in \text{set } \text{hs} \wedge t \in \text{set } \text{ts}\} \rangle$
 $\langle \text{proof} \rangle$

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*:

$\langle \text{set } (\text{children } A \ r \ (p \ # \ z)) =$
 $\{ \text{hs } \ @ \ \text{ts} \mid \text{hs } \ \text{ts}. \text{hs} \in \text{set } (\text{parts } A \ r \ p) \wedge$
 $\text{ts} \in \text{set } (\text{children } (\text{remdups } (A \ @ \ \text{subtermFms } (\text{concat } (\text{parts } A \ r \ p)))) \ r \ z) \} \rangle$
 $\langle \text{proof} \rangle$

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 $\langle p \in \text{set } z' \rangle$
 $\langle \text{proof} \rangle$

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 $\langle p \in \text{set } z' \rangle$
 $\langle \text{proof} \rangle$

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 $\langle \exists B \ xs. \text{set } A \subseteq \text{set } B \wedge xs \in \text{set } (\text{parts } B \ r \ p) \wedge \text{set } xs \subseteq \text{set } z' \rangle$
 $\langle \text{proof} \rangle$

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

lemma *ne-effect-not-branchDone*: $\langle (B, z') \mid \in \mid \text{effect } r \ (A, z) \rangle \implies \neg \text{branchDone } z \rangle$
 $\langle \text{proof} \rangle$

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 $\langle \exists C \ xs. \text{set } A \subseteq \text{set } C \wedge xs \in \text{set } (\text{parts } C \ r \ p) \wedge \text{set } xs \subseteq \text{set } z' \rangle$
 $\langle \text{proof} \rangle$

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

corollary $\langle \text{Neg } (\text{Neg } p) \in \text{set } z \implies (B, z') \mid \in \mid \text{effect } \text{NegNeg } (A, z) \implies p \in \text{set } z' \rangle$
 $\langle \text{proof} \rangle$

corollary $\langle \text{Neg } (\text{Uni } p) \in \text{set } z \implies (B, z') \mid \in \mid \text{effect } \text{GammaUni } (A, z) \implies \text{set } (\text{map } (\lambda t. \text{Neg } (\text{sub } 0 \ t \ p)) \ A) \subseteq \text{set } z' \rangle$
 $\langle \text{proof} \rangle$

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 $\langle \forall z' \in \text{set } (\text{children } (\text{remdups } (A @ \text{subtermFms } z))) \text{ r } z \rangle. \exists B. (B, z') \in |$
 $\text{ss} \rangle$
 $\langle \text{proof} \rangle$

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

lemma *Suc-max-new*:
fixes $xs :: \langle \text{nat list} \rangle$
shows $\langle \text{Suc } (\text{foldr } \text{max } xs \ 0) \notin \text{set } xs \rangle$
 $\langle \text{proof} \rangle$

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

2.3.8 Finding axioms

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

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

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

The arguments of a predicate are subterms of it.

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

The arguments of function are subterms of it.

lemma *subterm-Fun-refl*: $\langle \text{set } ts \subseteq \text{set } (\text{subtermTm } (\text{Fun } n \ ts)) \rangle$
 $\langle \text{proof} \rangle$

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

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

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

```
lemma subtermFm-preds: ⟨t ∈ set (subtermFm p) ⟷ (∃ pre ∈ preds p. t ∈ set
(subtermFm pre))⟩
  ⟨proof⟩
```

```
lemma preds-shape: ⟨pre ∈ preds p ⟹ ∃ n ts. pre = Pre n ts⟩
  ⟨proof⟩
```

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

```
lemma fun-arguments-subterm:
  assumes ⟨Fun n ts ∈ set (subtermFm p)⟩
  shows ⟨set ts ⊆ set (subtermFm p)⟩
  ⟨proof⟩
```

end

2.4 Hintikka sets for SeCaV

```
theory Hintikka
  imports Prover
begin
```

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

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

```
definition
  ⟨terms H ≡ if (∪ p ∈ H. set (subtermFm p)) = {} then {Fun 0 []}
  else (∪ p ∈ H. set (subtermFm p))⟩
```

```
locale Hintikka =
  fixes H :: ⟨fm set⟩
  assumes
```

Basic: $\langle \text{Pre } n \text{ ts} \in H \implies \text{Neg } (\text{Pre } n \text{ ts}) \notin H \rangle$ **and**
AlphaDis: $\langle \text{Dis } p \ q \in H \implies p \in H \wedge q \in H \rangle$ **and**
AlphaImp: $\langle \text{Imp } p \ q \in H \implies \text{Neg } p \in H \wedge q \in H \rangle$ **and**
AlphaCon: $\langle \text{Neg } (\text{Con } p \ q) \in H \implies \text{Neg } p \in H \wedge \text{Neg } q \in H \rangle$ **and**
BetaCon: $\langle \text{Con } p \ q \in H \implies p \in H \vee q \in H \rangle$ **and**
BetaImp: $\langle \text{Neg } (\text{Imp } p \ q) \in H \implies p \in H \vee \text{Neg } q \in H \rangle$ **and**
BetaDis: $\langle \text{Neg } (\text{Dis } p \ q) \in H \implies \text{Neg } p \in H \vee \text{Neg } q \in H \rangle$ **and**
GammaExi: $\langle \text{Exi } p \in H \implies \forall t \in \text{terms } H. \text{sub } 0 \ t \ p \in H \rangle$ **and**
GammaUni: $\langle \text{Neg } (\text{Uni } p) \in H \implies \forall t \in \text{terms } H. \text{Neg } (\text{sub } 0 \ t \ p) \in H \rangle$ **and**
DeltaUni: $\langle \text{Uni } p \in H \implies \exists t \in \text{terms } H. \text{sub } 0 \ t \ p \in H \rangle$ **and**
DeltaExi: $\langle \text{Neg } (\text{Exi } p) \in H \implies \exists t \in \text{terms } H. \text{Neg } (\text{sub } 0 \ t \ p) \in H \rangle$ **and**
Neg: $\langle \text{Neg } (\text{Neg } p) \in H \implies p \in H \rangle$

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* :: $\langle \text{state} \times \text{rule} \Rightarrow \text{sequent} \rangle$ **where**
 $\langle \text{pseq } z = \text{snd } (\text{fst } z) \rangle$

ptms is the list of terms in a node.

definition *ptms* :: $\langle \text{state} \times \text{rule} \Rightarrow \text{tm list} \rangle$ **where**
 $\langle \text{ptms } z = \text{fst } (\text{fst } z) \rangle$

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*: $\langle \text{epath } \text{steps} \implies \text{epath } (\text{sdrop } n \ \text{steps}) \rangle$
 $\langle \text{proof} \rangle$

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

lemma *sset-sdrop*: $\langle \text{sset } (\text{sdrop } n \ s) \subseteq \text{sset } s \rangle$
 $\langle \text{proof} \rangle$

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 $\langle \text{epath steps} \rangle \langle \text{eff} (\text{snd} (\text{shd steps})) (\text{fst} (\text{shd steps})) \text{ ss} \rangle$
shows $\langle \text{fst} (\text{shd} (\text{stl steps})) \mid \in \mid \text{ss} \rangle$
 $\langle \text{proof} \rangle$

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 $\langle B = \text{remdups} (A @ \text{subterms } z @ \text{subterms } z') \rangle$
 $\langle \text{proof} \rangle$

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 $\langle \exists B z' r'. (B, z') \mid \in \mid \text{effect } r (A, z) \wedge \text{shd} (\text{stl steps}) = ((B, z'), r') \wedge$
 $(B = \text{remdups} (A @ \text{subterms } z @ \text{subterms } z')) \rangle$
 $\langle \text{proof} \rangle$

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 $\langle \text{ptms} (\text{shd} (\text{stl steps})) = \text{remdups} (\text{ptms} (\text{shd steps}) @$
 $\text{subterms} (\text{pseq} (\text{shd steps}) @ \text{subterms} (\text{pseq} (\text{shd} (\text{stl steps})))) \rangle$
 $\langle \text{proof} \rangle$

The list of terms never decreases on an escape path.

lemma *epath-sdrop-ptms*:

assumes $\langle \text{epath steps} \rangle$
shows $\langle \text{set} (\text{ptms} (\text{shd steps})) \subseteq \text{set} (\text{ptms} (\text{shd} (\text{sdrop } n \text{ steps})) \rangle$
 $\langle \text{proof} \rangle$

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 $\langle \text{ev} (\text{holds } P) \text{ xs} \rangle$
shows $\langle \exists n. \text{list-all} (\text{not } P) (\text{stake } n \text{ xs}) \wedge \text{holds } P (\text{sdrop } n \text{ xs}) \rangle$

$\langle proof \rangle$

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 $\langle ev \ (holds \ P) \ xs \rangle$

shows $\langle \exists pre \ suf. \ list\text{-}all \ (not \ P) \ pre \wedge \ holds \ P \ suf \wedge \ xs = pre \ @- \ suf \rangle$

$\langle proof \rangle$

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

lemma *always-enabledAtStep*: $\langle enabledAtStep \ r \ xs \rangle$

$\langle proof \rangle$

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 $\langle p \in set \ (pseq \ (shd \ steps)) \rangle$ **and** $\langle epath \ steps \rangle$ **and** $\langle steps = pre \ @- \ suf \rangle$ **and**

$\langle list\text{-}all \ (not \ (\lambda step. \ affects \ (snd \ step) \ p)) \ pre \rangle$

shows $\langle p \in set \ (pseq \ (shd \ suf)) \rangle$

$\langle proof \rangle$

2.5.5 Formulas on an escape path form a Hintikka set

This definition captures the set of formulas on an entire path

definition $\langle tree\text{-}fms \ steps \equiv \bigcup ss \in sset \ steps. \ set \ (pseq \ ss) \rangle$

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

lemma *pseq-in-tree-fms*: $\langle \llbracket x \in sset \ steps; \ p \in set \ (pseq \ x) \rrbracket \implies p \in tree\text{-}fms \ steps \rangle$

$\langle proof \rangle$

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*: $\langle p \in tree\text{-}fms \ steps \implies \exists n. \ p \in set \ (pseq \ (steps \ !! \ n)) \rangle$

$\langle proof \rangle$

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*: $\langle Saturated \ steps \implies Saturated \ (sdrop \ n \ steps) \rangle$

$\langle proof \rangle$

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 $\langle \text{Saturated steps} \rangle$

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

$\langle \text{proof} \rangle$

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 $\langle \text{epath steps} \rangle$

shows $\langle \text{alw (holds (not (branchDone } o\ pseq))) steps} \rangle$

$\langle \text{proof} \rangle$

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** $\langle \text{Hintikka } ?H \rangle$)

$\langle \text{proof} \rangle$

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**

$\langle \text{usemantics } u\ e\ f\ g\ (\text{Pre } i\ l) = g\ i\ (\text{semantics-list } e\ f\ l) \rangle$

| $\langle \text{usemantics } u\ e\ f\ g\ (\text{Imp } p\ q) = (\text{usemantics } u\ e\ f\ g\ p \longrightarrow \text{usemantics } u\ e\ f\ g\ q) \rangle$

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

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

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

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

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

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

lemma *usemantics-UNIV*: $\langle \text{usemantics } \text{UNIV } e \text{ f } g \text{ p} \longleftrightarrow \text{semantics } e \text{ f } g \text{ p} \rangle$
 $\langle \text{proof} \rangle$

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*]: $\langle n \notin \text{params } p \implies \text{usemantics } u \text{ e } (f(n := x)) \text{ g } p \longleftrightarrow \text{usemantics } u \text{ e } f \text{ g } p \rangle$
 $\langle \text{proof} \rangle$

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

lemma *usubst-lemma* [*iff*]:
 $\langle \text{usemantics } u \text{ e } f \text{ g } (\text{subst } a \text{ t } i) \longleftrightarrow \text{usemantics } u \text{ (SeCaV.shift e } i \text{ (semantics-term } e \text{ f } t)) \text{ f } g \text{ a} \rangle$
 $\langle \text{proof} \rangle$

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 $\langle \text{is-env } u \text{ e} \rangle \langle \text{is-fdenot } u \text{ f} \rangle$
shows $\langle \text{semantics-term } e \text{ f } t \in u \rangle \langle \text{list-all } (\lambda x. x \in u) \text{ (semantics-list } e \text{ f } t) \rangle$
 $\langle \text{proof} \rangle$

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

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

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*)

$\langle G\ S\ n\ ts \equiv Neg\ (Pre\ n\ ts) \in S \rangle$

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

abbreviation

$\langle E\ S\ n \equiv if\ Var\ n \in terms\ S\ then\ Var\ n\ else\ SOME\ t.\ t \in terms\ S \rangle$

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

abbreviation

$\langle F\ S\ i\ l \equiv if\ Fun\ i\ l \in terms\ S\ then\ Fun\ i\ l\ else\ SOME\ t.\ t \in terms\ S \rangle$

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

lemma *terms-ne* [simp]: $\langle terms\ S \neq \{\} \rangle$

$\langle proof \rangle$

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*: $\langle t \in terms\ S \implies t = Fun\ 0\ [] \vee (\exists p \in S.\ t \in set\ (subtermFm\ p)) \rangle$

<proof>

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>

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>

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

lemma *is-env-E*:

$\langle \text{is-env } (\text{terms } S) (E S) \rangle$

<proof>

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

lemma *is-fdenot-F*:

$\langle \text{is-fdenot } (\text{terms } S) (F S) \rangle$

<proof>

abbreviation

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

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 $\langle \text{Hintikka } S \rangle$

shows $\langle (p \in S \longrightarrow \neg M S p) \wedge (\text{Neg } p \in S \longrightarrow M S p) \rangle$

<proof>

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 $\langle \Vdash \text{pre } @ \ z \rangle$
 $\langle \text{proof} \rangle$

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 $\langle \Vdash \ z \rangle$
 $\langle \text{proof} \rangle$

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

interpretation *Soundness eff rules UNIV* $\langle \lambda \cdot (A, z). (\Vdash \ z) \rangle$
 $\langle \text{proof} \rangle$

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 $\langle \Vdash \text{rootSequent } t \rangle$
 $\langle \text{proof} \rangle$

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 $A :: \langle \text{tm list} \rangle$ **and** $z :: \langle \text{fm list} \rangle$
defines $\langle t \equiv \text{secavProver } (A, z) \rangle$
shows $\langle (\text{fst } (\text{root } t) = (A, z) \wedge \text{wf } t \wedge \text{tfinite } t) \vee$
 $\langle (\exists \text{ steps. } \text{fst } (\text{shd } \text{steps}) = (A, z) \wedge \text{epath } \text{steps} \wedge \text{Saturated } \text{steps}) \rangle$
is $\langle ?A \vee ?B \rangle$

⟨*proof*⟩

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

abbreviation

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

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

lemma *epath-countermodel*:

assumes ⟨*fst* (*shd steps*) = (*A*, *z*)⟩ **and** ⟨*epath steps*⟩ **and** ⟨*Saturated steps*⟩
shows ⟨ $\neg \text{uvalid } z$ ⟩

⟨*proof*⟩

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

theorem *prover-completeness-usemantics*:

fixes *A* :: ⟨*tm list*⟩
assumes ⟨*uvalid z*⟩
defines ⟨*t* $\equiv \text{secavProver } (A, z)$ ⟩
shows ⟨*fst* (*root t*) = (*A*, *z*) \wedge *wf t* \wedge *tfinite t*⟩
⟨*proof*⟩

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

corollary *prover-completeness-SeCaV*:

fixes *A* :: ⟨*tm list*⟩
assumes ⟨ $\Vdash z$ ⟩
defines ⟨*t* $\equiv \text{secavProver } (A, z)$ ⟩
shows ⟨*fst* (*root t*) = (*A*, *z*) \wedge *wf t* \wedge *tfinite t*⟩
⟨*proof*⟩

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 $\langle \exists p \in set\ (rootSequent\ t).\ usemantics\ u\ e\ f\ g\ p \rangle$
 $\langle proof \rangle$

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 $\langle tfinite\ t \wedge wf\ t \longleftrightarrow uvalid\ z \rangle$
 $\langle proof \rangle$

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 $\langle tfinite\ t \wedge wf\ t \longleftrightarrow uvalid\ [p] \rangle$
 $\langle proof \rangle$

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 $\langle tfinite\ t \wedge wf\ t \longleftrightarrow (\Vdash\ z) \rangle$
 $\langle proof \rangle$

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 $\langle tfinite\ t \wedge wf\ t \longleftrightarrow (\Vdash\ [p]) \rangle$
 $\langle proof \rangle$

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 $\langle \exists p \in set\ (rootSequent\ t). semantics\ e\ f\ g\ p \rangle$
 $\langle proof \rangle$

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 $\langle semantics\ e\ f\ g\ p \rangle$
 $\langle proof \rangle$

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 $\langle fst\ (root\ t) = (A, [p]) \wedge wf\ t \wedge tfinite\ t \rangle$
 $\langle proof \rangle$

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 $\langle tfinite\ t \wedge wf\ t \longleftrightarrow (\forall (e :: nat \Rightarrow nat\ hterm) f\ g. semantics\ e\ f\ g\ p) \rangle$
 $\langle proof \rangle$

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

theorem *semantics-usemantics:*

$\langle (\forall (e :: nat \Rightarrow nat\ hterm) f\ g. semantics\ e\ f\ g\ p) \longleftrightarrow$
 $(\forall (u :: tm\ set) e\ f\ g. is-env\ u\ e \longrightarrow is-fdenot\ u\ f \longrightarrow usemantics\ u\ e\ f\ g\ p) \rangle$
 $\langle proof \rangle$

end