A Sequent Calculus Prover for First-Order Logic with Functions

Asta Halkjær From Frederik Krogsdal Jacobsen

March 29, 2023

Abstract

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

Contents

Chapter 1

SeCaV

1.1 Sequent Calculus Verifier (SeCaV)

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

1.2 Syntax: Terms / Formulas

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

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

1.3 Semantics: Terms / Formulas

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

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

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

— Test

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

⟨*proof*⟩

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

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

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

— Simplifications

lemma *member* [*iff*]: ‹*member p z* ⟷ *p* ∈ *set z*›
⟨*proof*⟩

lemma *ext* [*iff*]: ‹*ext y z* ⟷ *set z* ⊆ *set y*›
⟨*proof*⟩

1.5 Sequent Calculus

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

— Test

corollary ‹⊩ [*Imp* (*Pre 0* []) (*Pre 0* [])]›
⟨*proof*⟩

1.6 Shorthands

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

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

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

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

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

lemmas *Neg* = *sequent-calculus.intros*(*12*)

lemmas *Ext* = *sequent-calculus.intros*(*13*)

— Test

lemma ‹⊩
  [
    Imp (*Pre 0* []) (*Pre 0* [])
  ]
  ›
⟨*proof*⟩

1.7 Appendix: Soundness

1.7.1 Increment Function

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

1.7.2 Parameters: Terms

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

lemma *p0* [*simp*]: ‹*paramsts ts* = ⋃(*set* (*map paramst ts*))›
  ⟨*proof*⟩

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

lemma *p1* [*simp*]: ‹*paramst′ t* = *paramst t*›
  ⟨*proof*⟩

1.7.3 Parameters: Formulas

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

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

lemma *p2* [*simp*]: ‹*params′ p* = *params p*›
⟨*proof*⟩

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

lemma *p1′* [*simp*]: ‹*paramst″ t* = *paramst t*›
⟨*proof*⟩

fun *params″* :: ‹*fm* ⇒ *nat set*› **where**
‹*params″* (*Pre b ts*) = (⋃ *t* ∈ *set ts*. *paramst″ t*)› |
‹*params″* (*Imp p q*) = *params″ p* ∪ *params″ q*› |
‹*params″* (*Dis p q*) = *params″ p* ∪ *params″ q*› |
‹*params″* (*Con p q*) = *params″ p* ∪ *params″ q*› |
‹*params″* (*Exi p*) = *params″ p*› |
‹*params″* (*Uni p*) = *params″ p*› |
‹*params″* (*Neg p*) = *params″ p*›

lemma *p2′* [*simp*]: ‹*params″ p* = *params p*›
⟨*proof*⟩

1.7.4 Update Lemmas

lemma *upd-lemma′* [*simp*]:
‹*n* ∉ *paramst t* ⟹ *semantics-term e* (*f*(*n* := *z*)) *t* = *semantics-term e f t*›
‹*n* ∉ *paramsts ts* ⟹ *semantics-list e* (*f*(*n* := *z*)) *ts* = *semantics-list e f ts*›
⟨*proof*⟩

lemma *upd-lemma* [*iff*]: ‹*n* ∉ *params p* ⟹ *semantics e* (*f*(*n* := *z*)) *g p* ⟷

semantics e f g p›
⟨*proof*⟩

1.7.5 Substitution

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

primrec *subst* :: ‹*fm* ⇒ *tm* ⇒ *nat* ⇒ *fm*› **where**
‹*subst* (*Pre b ts*) *s k* = *Pre b* (*substs 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* (*substt t u i*) = *semantics-term* (*shift e i* (*semantics-term e f u*)) *f t*›
‹*semantics-list e f* (*substs 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*]: ‹*new-term c t* ⟷ (*c* ∉ *paramst t*)› ‹*new-list c l* ⟷ (*c* ∉ *paramsts l*)›
⟨*proof*⟩

lemma *s2* [*iff*]: ‹*new c p* ⟷ (*c* ∉ *params p*)›
⟨*proof*⟩

lemma *s3* [*iff*]: ‹*news c z* ⟷ *list-all* (λ*p*. *c* ∉ *params p*) *z*›
⟨*proof*⟩

lemma *s4* [*simp*]: ‹*inc-term t* = *liftt t*› ‹*inc-list l* = *liftts l*›
⟨*proof*⟩

lemma *s5* [*simp*]: ‹*sub-term v s t* = *substt t s v*› ‹*sub-list v s l* = *substs l s v*›
⟨*proof*⟩

lemma *s6* [*simp*]: ‹*sub v s p* = *subst p s v*›
⟨*proof*⟩

1.7.7 Soundness

theorem *sound*: ‹⊩ *z* ⟹ ∃ *p* ∈ *set z*. *semantics e f g p*›
⟨*proof*⟩

corollary ‹⊩ *z* ⟹ ∃ *p*. *member p z* ∧ *semantics e f g p*›
⟨*proof*⟩

corollary ‹⊩ [*p*] ⟹ *semantics e f g p*›
⟨*proof*⟩

corollary ‹¬ (⊩ [])›
⟨*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**
‹*from-tm* (*Var n*) = *FOL-Fitting.Var n*› |
‹*from-tm* (*Fun a ts*) = *App a* (*from-tm-list ts*)› |
‹*from-tm-list* [] = []› |
‹*from-tm-list* (*t* # *ts*) = *from-tm t* # *from-tm-list ts*›

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

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

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

theorem *to-from-tm* [*simp*]: ‹*to-tm* (*from-tm t*) = *t*› ‹*to-tm-list* (*from-tm-list ts*) = *ts*›
⟨*proof*⟩

theorem *to-from-fm* [*simp*]: ‹*to-fm* (*from-fm p*) = *p*›
⟨*proof*⟩

lemma *Truth* [*simp*]: ‹⊩ *Imp* (*Pre 0* []) (*Pre 0* []) # *z*›
⟨*proof*⟩

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

⟨*proof*⟩

lemma *params* [*iff*]: ‹*FOL-Fitting.new c p* ⟷ *new c* (*to-fm p*)›
⟨*proof*⟩

lemma *list-params* [*iff*]: ‹*FOL-Fitting.news c z* ⟷ *news c* (*map to-fm z*)›
⟨*proof*⟩

lemma *liftt* [*simp*]:
‹*to-tm* (*FOL-Fitting.liftt t*) = *inc-term* (*to-tm t*)›
‹*to-tm-list* (*FOL-Fitting.liftts l*) = *inc-list* (*to-tm-list l*)›
⟨*proof*⟩

lemma *substt* [*simp*]:
‹*to-tm* (*FOL-Fitting.substt t s v*) = *sub-term v* (*to-tm s*) (*to-tm t*)›
‹*to-tm-list* (*FOL-Fitting.substs l s v*) = *sub-list v* (*to-tm s*) (*to-tm-list l*)›
⟨*proof*⟩

lemma *subst* [*simp*]: ‹*to-fm* (*FOL-Fitting.subst A t s*) = *sub s* (*to-tm t*) (*to-fm A*)›
⟨*proof*⟩

lemma *sim*: ‹(⊢ *x*) ⟹ (⊩ (*map to-fm x*))›
⟨*proof*⟩

lemma *evalt* [*simp*]:
‹*semantics-term e f t* = *evalt e f* (*from-tm t*)›
‹*semantics-list e f ts* = *evalts e f* (*from-tm-list ts*)›
⟨*proof*⟩

lemma *shift* [*simp*]: ‹*shift e 0 x* = *e*⟨*0*:*x*⟩›
⟨*proof*⟩

lemma *semantics* [*iff*]: ‹*semantics e f g p* ⟷ *eval e f g* (*from-fm p*)›
⟨*proof*⟩

abbreviation *valid* (≫ - *0*) **where**
‹(≫ *p*) ≡ ∀ (*e* :: - ⇒ *nat hterm*) *f g*. *semantics e f g p*›

theorem *complete-sound*: ‹≫ *p* ⟹ ⊩ [*p*]› ‹⊩ [*q*] ⟹ *semantics e f g q*›
⟨*proof*⟩

corollary ‹(≫ *p*) ⟷ (⊩ [*p*])›
⟨*proof*⟩

1.10 Reference

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

end

Chapter 2

The prover

2.1 Proof search procedure

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

This theory defines the actual proof search procedure.

2.1.1 Datatypes

A sequent is a list of formulas

type-synonym *sequent* = ‹*fm list*›

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

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

2.1.2 Auxiliary functions

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

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

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

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

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

subtermTm returns a list of all terms occurring within a term

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

subtermFm returns a list of all terms occurring within a formula

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

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

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

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

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

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

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

2.1.3 Effects of rules

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

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

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

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

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

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

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

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

This function defines the effect of applying a rule to a proof state. If the sequent is an axiom, the effect is to end the branch of the proof tree, so an

empty set of child branches is returned. Otherwise, we compute the children generated by applying the rule to the current proof state, then add any new subterms to the proof states of the children.

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

2.1.4 The rule stream

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

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

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

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

2.1.5 Abstract completeness

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

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

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

lemma *all-rules-enabled*: ‹∀ *st*. ∀ *r* ∈ *i.R* (*cycle rulesList*). ∃ *sl*. *eff r st sl*›
⟨*proof*⟩

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

interpretation *RuleSystem eff rules UNIV*
⟨*proof*⟩

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*: ‹∀ *r*. *r* ∈ *R* ⟶ *per r*›
⟨*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* ≡ *mkTree rules*›

abbreviation ‹*rootSequent t* ≡ *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* ∨ *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* ⇀ (*Haskell*) *MaybeExt.fromJust*
| **constant** *Option.is-none* ⇀ (*Haskell*) *MaybeExt.isNothing*

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

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

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

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

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

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

We then export this version of the prover into Haskell.

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

end

2.3 Lemmas about the prover

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

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

2.3.1 SeCaV lemmas

We need a few lemmas about the SeCaV system.

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

lemma *paramst-liftt* [*simp*]:
  ‹*paramst* (*liftt t*) = *paramst t*› ‹*paramsts* (*liftts ts*) = *paramsts ts*›
  ⟨*proof*⟩

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

lemma *paramst-sub-term*:
  ‹*paramst* (*sub-term m s t*) ⊆ *paramst s* ∪ *paramst t*›
  ‹*paramsts* (*sub-list m s l*) ⊆ *paramst s* ∪ *paramsts l*›
  ⟨*proof*⟩

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

lemma *params-sub*: ‹*params* (*sub m t p*) ⊆ *paramst t* ∪ *params p*›
⟨*proof*⟩

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

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

lemma *news-paramss*: ‹*news i z* ⟷ *i* ∉ *paramss z*›
⟨*proof*⟩

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

lemma *paramsts-subset*: ‹*set A* ⊆ *set B* ⟹ *paramsts A* ⊆ *paramsts B*›
⟨*proof*⟩

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

lemma *size-sub* [*simp*]: ‹*size* (*sub i t p*) = *size p*›
⟨*proof*⟩

2.3.2 Fairness

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

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

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

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

primrec *rule-index* :: ‹*rule* ⇒ *nat*› **where**
‹*rule-index NegNeg* = *0*›
| ‹*rule-index AlphaImp* = *1*›
| ‹*rule-index AlphaDis* = *2*›
| ‹*rule-index AlphaCon* = *3*›
| ‹*rule-index DeltaExi* = *4*›
| ‹*rule-index DeltaUni* = *5*›
| ‹*rule-index BetaImp* = *6*›

| ‹*rule-index BetaDis* = *7*›
| ‹*rule-index BetaCon* = *8*›
| ‹*rule-index GammaExi* = *9*›
| ‹*rule-index GammaUni* = *10*›

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

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

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

lemma *cycle-nth*: ‹*xs* ≠ [] ⟹ *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 ‹∀ *i* < *k*. ∃ *n* > *m*. *n mod k* = *i*›
⟨*proof*⟩

lemma *mod-suff*:
assumes ‹∀ (*n* :: *nat*) > *m*. *P* (*n mod k*)› ‹*0* < *k*›
shows ‹∀ *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*: ‹∃ *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*: ‹∃ *n*. (*sdrop k rules*) !! *n* = *r*›
⟨*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*: ‹*t* ∈ *set* (*subtermTm s*) ⟹ *set* (*subtermTm t*) ⊆ *set* (*subtermTm s*)›
⟨*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* ≠ *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*⟩

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*))›
⟨*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 ‹*ts* ≡ ⋃ *p* ∈ *set z*. *set* (*subtermFm p*)›
shows ‹*set* (*subterms z*) = (*if ts* = {} *then* {*Fun 0* []} *else ts*)›
⟨*proof*⟩

The parameters and the subterm functions respect each other.

lemma *paramst-subtermTm*:

‹$\forall\, i \in \mathit{paramst}\ t.\ \exists\, l.\ \mathit{Fun}\ i\ l \in \mathit{set}\ (\mathit{subtermTm}\ t)$›
‹$\forall\, i \in \mathit{paramsts}\ \mathit{ts}.\ \exists\, l.\ \mathit{Fun}\ i\ l \in (\bigcup t \in \mathit{set}\ \mathit{ts}.\ \mathit{set}\ (\mathit{subtermTm}\ t))$›
⟨*proof*⟩

lemma *params-subtermFm*: ‹$\forall\, i \in \mathit{params}\ p.\ \exists\, l.\ \mathit{Fun}\ i\ l \in \mathit{set}\ (\mathit{subtermFm}\ p)$›
⟨*proof*⟩

lemma *subtermFm-subset-params*: ‹$\mathit{set}\ (\mathit{subtermFm}\ p) \subseteq \mathit{set}\ A \Longrightarrow \mathit{params}\ p \subseteq \mathit{paramsts}\ A$›
⟨*proof*⟩

2.3.4 Custom cases

Some proofs are more efficient with some custom case lemmas.

lemma *Neg-exhaust*
[*case-names Pre Imp Dis Con Exi Uni NegPre NegImp NegDis NegCon NegExi NegUni NegNeg*]:
assumes
‹$\bigwedge i\ \mathit{ts}.\ x = \mathit{Pre}\ i\ \mathit{ts} \Longrightarrow P$›
‹$\bigwedge p\ q.\ x = \mathit{Imp}\ p\ q \Longrightarrow P$›
‹$\bigwedge p\ q.\ x = \mathit{Dis}\ p\ q \Longrightarrow P$›
‹$\bigwedge p\ q.\ x = \mathit{Con}\ p\ q \Longrightarrow P$›
‹$\bigwedge p.\ x = \mathit{Exi}\ p \Longrightarrow P$›
‹$\bigwedge p.\ x = \mathit{Uni}\ p \Longrightarrow P$›
‹$\bigwedge i\ \mathit{ts}.\ x = \mathit{Neg}\ (\mathit{Pre}\ i\ \mathit{ts}) \Longrightarrow P$›
‹$\bigwedge p\ q.\ x = \mathit{Neg}\ (\mathit{Imp}\ p\ q) \Longrightarrow P$›
‹$\bigwedge p\ q.\ x = \mathit{Neg}\ (\mathit{Dis}\ p\ q) \Longrightarrow P$›
‹$\bigwedge p\ q.\ x = \mathit{Neg}\ (\mathit{Con}\ p\ q) \Longrightarrow P$›
‹$\bigwedge p.\ x = \mathit{Neg}\ (\mathit{Exi}\ p) \Longrightarrow P$›
‹$\bigwedge p.\ x = \mathit{Neg}\ (\mathit{Uni}\ p) \Longrightarrow P$›
‹$\bigwedge p.\ x = \mathit{Neg}\ (\mathit{Neg}\ p) \Longrightarrow P$›
shows P
⟨*proof*⟩

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

‹∀ A. parts A r x = [[x]] ⟹ P›
shows *P*
⟨*proof*⟩

2.3.5 Unaffected formulas

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

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

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

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

lemma *parts-preserves-unaffected*:
assumes ‹¬ *affects r p*› ‹*z'* ∈ *set* (*parts A r p*)›
shows ‹*p* ∈ *set z'*›
⟨*proof*⟩

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

lemma *list-prod-is-cartesian*:
‹*set* (*list-prod hs ts*) = {*h* @ *t* |*h t*. *h* ∈ *set hs* ∧ *t* ∈ *set ts*}›
⟨*proof*⟩

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

lemma *set-children-Cons*:
‹*set* (*children A r* (*p* # *z*)) =
{*hs* @ *ts* |*hs ts*. *hs* ∈ *set* (*parts A r p*) ∧
ts ∈ *set* (*children* (*remdups* (*A* @ *subtermFms* (*concat* (*parts A r p*)))) *r z*)}›
⟨*proof*⟩

The *children* function does not change unaffected formulas.

lemma *children-preserves-unaffected*:

assumes ‹$p \in set\ z$› ‹$\neg\ affects\ r\ p$› ‹$z' \in set\ (children\ A\ r\ z)$›
shows ‹$p \in set\ z'$›
⟨*proof*⟩

The *effect* function does not change unaffected formulas.

lemma *effect-preserves-unaffected*:
assumes ‹$p \in set\ z$› **and** ‹$\neg\ affects\ r\ p$› **and** ‹$(B,\ z')\ |\in|\ effect\ r\ (A,\ z)$›
shows ‹$p \in set\ z'$›
⟨*proof*⟩

2.3.6 Affected formulas

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

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

lemma *parts-in-children*:
assumes ‹$p \in set\ z$› ‹$z' \in set\ (children\ A\ r\ z)$›
shows ‹$\exists\ B\ xs.\ set\ A \subseteq set\ B \wedge xs \in set\ (parts\ B\ r\ p) \wedge set\ xs \subseteq set\ z'$›
⟨*proof*⟩

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

lemma *ne-effect-not-branchDone*: ‹$(B,\ z')\ |\in|\ effect\ r\ (A,\ z) \Longrightarrow \neg\ branchDone\ z$›
⟨*proof*⟩

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

lemma *parts-in-effect*:
assumes ‹$p \in set\ z$› **and** ‹$(B,\ z')\ |\in|\ effect\ r\ (A,\ z)$›
shows ‹$\exists\ C\ xs.\ set\ A \subseteq set\ C \wedge xs \in set\ (parts\ C\ r\ p) \wedge set\ xs \subseteq set\ z'$›
⟨*proof*⟩

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

corollary ‹$Neg\ (Neg\ p) \in set\ z \Longrightarrow (B,\ z')\ |\in|\ effect\ NegNeg\ (A,\ z) \Longrightarrow p \in set\ z'$›
⟨*proof*⟩

corollary ‹$Neg\ (Uni\ p) \in set\ z \Longrightarrow (B,\ z')\ |\in|\ effect\ GammaUni\ (A,\ z) \Longrightarrow$
$set\ (map\ (\lambda t.\ Neg\ (sub\ 0\ t\ p))\ A) \subseteq set\ z'$›
⟨*proof*⟩

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 ‹¬ *branchDone z*› ‹*eff r* (*A*, *z*) *ss*›
shows ‹∀ *z′* ∈ *set* (*children* (*remdups* (*A* @ *subtermFms z*)) *r z*). ∃ *B*. (*B*, *z′*) |∈| *ss*›
⟨*proof*⟩

2.3.7 Generating new function names

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

lemma *foldr-max*:
fixes *xs* :: ‹*nat list*›
shows ‹*foldr max xs 0* = (*if xs* = [] *then 0 else Max* (*set xs*))›
⟨*proof*⟩

lemma *Suc-max-new*:
fixes *xs* :: ‹*nat list*›
shows ‹*Suc* (*foldr max xs 0*) ∉ *set xs*›
⟨*proof*⟩

lemma *listFunTm-paramst*: ‹*set* (*listFunTm t*) = *paramst t*› ‹*set* (*listFunTms ts*) = *paramsts ts*›
⟨*proof*⟩

2.3.8 Finding axioms

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

lemma *branchDone-contradiction*: ‹*branchDone z* ⟷ (∃ *p*. *p* ∈ *set z* ∧ *Neg p* ∈ *set z*)›
⟨*proof*⟩

2.3.9 Subterms

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

Any term is a subterm of itself.

lemma *subtermTm-refl* [*simp*]: ‹*t* ∈ *set* (*subtermTm t*)›
⟨*proof*⟩

The arguments of a predicate are subterms of it.

lemma *subterm-Pre-refl*: ‹*set ts* ⊆ *set* (*subtermFm* (*Pre n ts*))›
⟨*proof*⟩

The arguments of function are subterms of it.

lemma *subterm-Fun-refl*: ‹*set ts* ⊆ *set* (*subtermTm* (*Fun n ts*))›
⟨*proof*⟩

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: ‹*Pre n ts* ∈ *H* ⟹ *Neg* (*Pre n ts*) ∉ *H*› **and**
AlphaDis: ‹*Dis p q* ∈ *H* ⟹ *p* ∈ *H* ∧ *q* ∈ *H*› **and**
AlphaImp: ‹*Imp p q* ∈ *H* ⟹ *Neg p* ∈ *H* ∧ *q* ∈ *H*› **and**
AlphaCon: ‹*Neg* (*Con p q*) ∈ *H* ⟹ *Neg p* ∈ *H* ∧ *Neg q* ∈ *H*› **and**
BetaCon: ‹*Con p q* ∈ *H* ⟹ *p* ∈ *H* ∨ *q* ∈ *H*› **and**
BetaImp: ‹*Neg* (*Imp p q*) ∈ *H* ⟹ *p* ∈ *H* ∨ *Neg q* ∈ *H*› **and**
BetaDis: ‹*Neg* (*Dis p q*) ∈ *H* ⟹ *Neg p* ∈ *H* ∨ *Neg q* ∈ *H*› **and**
GammaExi: ‹*Exi p* ∈ *H* ⟹ ∀ *t* ∈ *terms H*. *sub 0 t p* ∈ *H*› **and**
GammaUni: ‹*Neg* (*Uni p*) ∈ *H* ⟹ ∀ *t* ∈ *terms H*. *Neg* (*sub 0 t p*) ∈ *H*› **and**
DeltaUni: ‹*Uni p* ∈ *H* ⟹ ∃ *t* ∈ *terms H*. *sub 0 t p* ∈ *H*› **and**
DeltaExi: ‹*Neg* (*Exi p*) ∈ *H* ⟹ ∃ *t* ∈ *terms H*. *Neg* (*sub 0 t p*) ∈ *H*› **and**
Neg: ‹*Neg* (*Neg p*) ∈ *H* ⟹ *p* ∈ *H*›

end

2.5 Escape path formulas are Hintikka

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

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

2.5.1 Definitions

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

pseq is the sequent in a node.

definition *pseq* :: ‹*state* × *rule* ⇒ *sequent*› **where**
‹*pseq z* = *snd* (*fst z*)›

ptms is the list of terms in a node.

definition *ptms* :: ‹*state* × *rule* ⇒ *tm list*› **where**
‹*ptms z* = *fst* (*fst z*)›

2.5.2 Facts about streams

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

lemma *epath-sdrop*: ‹*epath steps* ⟹ *epath* (*sdrop n steps*)›
⟨*proof*⟩

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

lemma *sset-sdrop*: ‹*sset* (*sdrop n s*) ⊆ *sset s*›
⟨*proof*⟩

2.5.3 Transformation of states on an escape path

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

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

lemma *epath-eff*:
assumes ‹*epath steps*› ‹*eff* (*snd* (*shd steps*)) (*fst* (*shd steps*)) *ss*›
shows ‹*fst* (*shd* (*stl steps*)) |∈| *ss*›
⟨*proof*⟩

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

lemma *effect-tms*:
assumes ‹(*B*, *z'*) |∈| *effect r* (*A*, *z*)›
shows ‹*B* = *remdups* (*A* @ *subterms z* @ *subterms z'*)›
⟨*proof*⟩

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

lemma *epath-effect*:
assumes ‹*epath steps*› ‹*shd steps* = ((*A*, *z*), *r*)›
shows ‹∃ *B z' r'*. (*B*, *z'*) |∈| *effect r* (*A*, *z*) ∧ *shd* (*stl steps*) = ((*B*, *z'*), *r'*) ∧
(*B* = *remdups* (*A* @ *subterms z* @ *subterms z'*))›
⟨*proof*⟩

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

lemma *epath-stl-ptms*:
assumes ‹*epath steps*›
shows ‹*ptms* (*shd* (*stl steps*)) = *remdups* (*ptms* (*shd steps*) @
subterms (*pseq* (*shd steps*)) @ *subterms* (*pseq* (*shd* (*stl steps*))))›
⟨*proof*⟩

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*)))›
⟨*proof*⟩

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*)›

⟨*proof*⟩

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*›
⟨*proof*⟩

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

lemma *always-enabledAtStep*: ‹*enabledAtStep r xs*›
⟨*proof*⟩

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

lemma *epath-preserves-unaffected*:
assumes ‹*p* ∈ *set* (*pseq* (*shd steps*))› **and** ‹*epath steps*› **and** ‹*steps* = *pre* @− *suf*› **and**
‹*list-all* (*not* (λ*step*. *affects* (*snd step*) *p*)) *pre*›
shows ‹*p* ∈ *set* (*pseq* (*shd suf*))›
⟨*proof*⟩

2.5.5 Formulas on an escape path form a Hintikka set

This definition captures the set of formulas on an entire path

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

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

lemma *pseq-in-tree-fms*: ‹⟦*x* ∈ *sset steps*; *p* ∈ *set* (*pseq x*)⟧ ⟹ *p* ∈ *tree-fms steps*›
⟨*proof*⟩

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

lemma *tree-fms-in-pseq*: ‹*p* ∈ *tree-fms steps* ⟹ ∃ *n*. *p* ∈ *set* (*pseq* (*steps* !! *n*))›
⟨*proof*⟩

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

lemma *Saturated-sdrop*: ‹*Saturated steps* ⟹ *Saturated* (*sdrop n steps*)›
⟨*proof*⟩

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

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

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

lemma *Saturated-ev-rule*:
  assumes ‹*Saturated steps*›
  shows ‹*ev* (*holds* (*is-rule r*)) (*sdrop n steps*)›
⟨*proof*⟩

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

lemma *epath-never-branchDone*:
  assumes ‹*epath steps*›
  shows ‹*alw* (*holds* (*not* (*branchDone o pseq*))) *steps*›
⟨*proof*⟩

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

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

lemma *escape-path-Hintikka*:
  assumes ‹*epath steps*› **and** ‹*Saturated steps*›
  shows ‹*Hintikka* (*tree-fms steps*)›
    (**is** ‹*Hintikka ?H*›)
⟨*proof*⟩

end

2.6 Bounded semantics

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

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

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

primrec *usemantics* **where**
  ‹*usemantics u e f g* (*Pre i l*) = *g i* (*semantics-list e f l*)›
| ‹*usemantics u e f g* (*Imp p q*) = (*usemantics u e f g p* ⟶ *usemantics u e f g q*)›

| ‹*usemantics u e f g* (*Dis p q*) = (*usemantics u e f g p* ∨ *usemantics u e f g q*)›
| ‹*usemantics u e f g* (*Con p q*) = (*usemantics u e f g p* ∧ *usemantics u e f g q*)›
| ‹*usemantics u e f g* (*Exi p*) = (∃ *x* ∈ *u*. *usemantics u* (*SeCaV.shift e 0 x*) *f g p*)›
| ‹*usemantics u e f g* (*Uni p*) = (∀ *x* ∈ *u*. *usemantics u* (*SeCaV.shift e 0 x*) *f g p*)›
| ‹*usemantics u e f g* (*Neg p*) = (¬ *usemantics u e f g p*)›

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

definition *is-env* :: ‹'*a set* ⇒ (*nat* ⇒ '*a*) ⇒ *bool*› **where**
‹*is-env u e* ≡ ∀ *n*. *e n* ∈ *u*›

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

definition *is-fdenot* :: ‹'*a set* ⇒ (*nat* ⇒ '*a list* ⇒ '*a*) ⇒ *bool*› **where**
‹*is-fdenot u f* ≡ ∀ *i l*. *list-all* (λ*x*. *x* ∈ *u*) *l* ⟶ *f i l* ∈ *u*›

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

lemma *usemantics-UNIV*: ‹*usemantics UNIV e f g p* ⟷ *semantics e f g p*›
⟨*proof*⟩

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

lemma *uupd-lemma* [*iff*]: ‹*n* ∉ *params p* ⟹ *usemantics u e* (*f*(*n* := *x*)) *g p* ⟷ *usemantics u e f g p*›
⟨*proof*⟩

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

lemma *usubst-lemma* [*iff*]:
‹*usemantics u e f g* (*subst a t i*) ⟷ *usemantics u* (*SeCaV.shift e i* (*semantics-term e f t*)) *f g a*›
⟨*proof*⟩

Soundness of SeCaV with regards to the bounded semantics

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

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

lemma *usemantics-term* [*simp*]:
assumes ‹*is-env u e*› ‹*is-fdenot u f*›
shows ‹*semantics-term e f t* ∈ *u*› ‹*list-all* (λ*x*. *x* ∈ *u*) (*semantics-list e f ts*)›
⟨*proof*⟩

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

lemma *is-fdenot-shift* [*simp*]: ‹*is-fdenot u f* ⟹ *x* ∈ *u* ⟹ *is-fdenot u* (*f*(*i* := λ-. *x*))›
⟨*proof*⟩

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

theorem *sound-usemantics*:
assumes ‹⊩ *z*› **and** ‹*is-env u e*› **and** ‹*is-fdenot u f*›
shows ‹∃ *p* ∈ *set z*. *usemantics u e f g p*›
⟨*proof*⟩

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* ≠ {}›
⟨*proof*⟩

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*))›

⟨*proof*⟩

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

lemma *terms-downwards-closed*: ‹$t \in terms\ S \Longrightarrow set\ (subtermTm\ t) \subseteq terms\ S$›
⟨*proof*⟩

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

lemma *usemantics-E*:
‹$t \in terms\ S \Longrightarrow semantics\text{-}term\ (E\ S)\ (F\ S)\ t = t$›
‹$list\text{-}all\ (\lambda t.\ t \in terms\ S)\ ts \Longrightarrow semantics\text{-}list\ (E\ S)\ (F\ S)\ ts = ts$›
⟨*proof*⟩

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

lemma *is-env-E*:
‹$is\text{-}env\ (terms\ S)\ (E\ S)$›
⟨*proof*⟩

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

lemma *is-fdenot-F*:
‹$is\text{-}fdenot\ (terms\ S)\ (F\ S)$›
⟨*proof*⟩

abbreviation
‹$M\ S \equiv usemantics\ (terms\ S)\ (E\ S)\ (F\ S)\ (G\ S)$›

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

theorem *Hintikka-counter-model*:
assumes ‹*Hintikka S*›
shows ‹$(p \in S \longrightarrow \neg\ M\ S\ p) \wedge (Neg\ p \in S \longrightarrow M\ S\ p)$›
⟨*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 ‹$\forall z' \in set\ (children\ A\ r\ z).\ (\Vdash pre\ @\ z')$›
and ‹$paramss\ (pre\ @\ z) \subseteq paramsts\ A$›
shows ‹$\Vdash pre\ @\ z$›
⟨*proof*⟩

As a special case, the prefix can be empty.

corollary *SeCaV-children*:
assumes ‹$\forall z' \in set\ (children\ A\ r\ z).\ (\Vdash z')$› **and** ‹$paramss\ z \subseteq paramsts\ A$›
shows ‹$\Vdash z$›
⟨*proof*⟩

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

interpretation *Soundness eff rules UNIV* ‹λ- $(A, z).\ (\Vdash z)$›
⟨*proof*⟩

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

theorem *prover-soundness-SeCaV*:
assumes ‹*tfinite t*› **and** ‹*wf t*›
shows ‹$\Vdash rootSequent\ t$›
⟨*proof*⟩

end

2.9 Completeness

theory *Completeness*
imports *Countermodel EPathHintikka*
begin

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

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

theorem *epath-prover-completeness*:
fixes A :: ‹*tm list*› **and** z :: ‹*fm list*›
defines ‹$t \equiv secavProver\ (A, z)$›
shows ‹$(fst\ (root\ t) = (A, z) \wedge wf\ t \wedge tfinite\ t) \vee$
$(\exists\ steps.\ fst\ (shd\ steps) = (A, z) \wedge epath\ steps \wedge Saturated\ steps)$›
(**is** ‹*?A* $\vee$ *?B*›)

⟨*proof*⟩

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

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

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

lemma *epath-countermodel*:
  assumes ‹*fst* (*shd steps*) = (*A*, *z*)› **and** ‹*epath steps*› **and** ‹*Saturated steps*›
  shows ‹¬ *uvalid z*›
⟨*proof*⟩

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

theorem *prover-completeness-usemantics*:
  fixes *A* :: ‹*tm list*›
  assumes ‹*uvalid z*›
  defines ‹*t* ≡ *secavProver* (*A*, *z*)›
  shows ‹*fst* (*root t*) = (*A*, *z*) ∧ *wf t* ∧ *tfinite t*›
  ⟨*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 ‹⊩ *z*›
  defines ‹*t* ≡ *secavProver* (*A*, *z*)›
  shows ‹*fst* (*root t*) = (*A*, *z*) ∧ *wf t* ∧ *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 ‹*tfinite t*› ‹*wf t*› ‹*is-env u e*› ‹*is-fdenot u f*›
shows ‹∃ *p* ∈ *set* (*rootSequent t*). *usemantics u e f g p*›
⟨*proof*⟩

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

theorem *prover-usemantics*:
fixes *A* :: ‹*tm list*› **and** *z* :: ‹*fm list*›
defines ‹*t* ≡ *secavProver* (*A*, *z*)›
shows ‹*tfinite t* ∧ *wf t* ⟷ *uvalid z*›
⟨*proof*⟩

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

corollary
fixes *p* :: *fm*
defines ‹*t* ≡ *secavProver* ([], [*p*])›
shows ‹*tfinite t* ∧ *wf t* ⟷ *uvalid* [*p*]›
⟨*proof*⟩

2.10.2 SeCaV

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

theorem *prover-SeCaV*:
fixes *A* :: ‹*tm list*› **and** *z* :: ‹*fm list*›
defines ‹*t* ≡ *secavProver* (*A*, *z*)›
shows ‹*tfinite t* ∧ *wf t* ⟷ (⊩ *z*)›
⟨*proof*⟩

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

corollary
fixes *p* :: *fm*
defines ‹*t* ≡ *secavProver* ([], [*p*])›
shows ‹*tfinite t* ∧ *wf t* ⟷ (⊩ [*p*])›
⟨*proof*⟩

2.10.3 Semantics

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

corollary *prover-soundness-semantics*:
assumes ‹*tfinite t*› ‹*wf t*›
shows ‹∃ *p* ∈ *set* (*rootSequent t*). *semantics e f g p*›
⟨*proof*⟩

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

corollary
assumes ‹*tfinite t*› ‹*wf t*› ‹*snd* (*fst* (*root t*)) = [*p*]›
shows ‹*semantics e f g p*›
⟨*proof*⟩

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

corollary *prover-completeness-semantics*:
fixes *A* :: ‹*tm list*›
assumes ‹∀ (*e* :: *nat* ⇒ *nat hterm*) *f g*. *semantics e f g p*›
defines ‹*t* ≡ *secavProver* (*A*, [*p*])›
shows ‹*fst* (*root t*) = (*A*, [*p*]) ∧ *wf t* ∧ *tfinite t*›
⟨*proof*⟩

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

theorem *prover-semantics*:
fixes *A* :: ‹*tm list*› **and** *p* :: *fm*
defines ‹*t* ≡ *secavProver* (*A*, [*p*])›
shows ‹*tfinite t* ∧ *wf t* ⟷ (∀ (*e* :: *nat* ⇒ *nat hterm*) *f g*. *semantics e f g p*)›
⟨*proof*⟩

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

theorem *semantics-usemantics*:
‹(∀ (*e* :: *nat* ⇒ *nat hterm*) *f g*. *semantics e f g p*) ⟷
(∀ (*u* :: *tm set*) *e f g*. *is-env u e* ⟶ *is-fdenot u f* ⟶ *usemantics u e f g p*)›
⟨*proof*⟩

end