A Sequent Calculus for First-Order Logic

Asta Halkjær From

March 17, 2025

Abstract

This work formalizes soundness and completeness of a one-sided sequent calculus for first-order logic. The completeness is shown via a translation from a complete semantic tableau calculus, the proof of which is based on the First-Order Logic According to Fitting theory. The calculi and proof techniques are taken from Ben-Ari's Mathematical Logic for Computer Science [1].

Contents

1 Common Notation **2**

2 Tableau Calculus **2**
2.1 Soundness . . . 3
2.2 Completeness for Closed Formulas . . . 4
2.3 Open Formulas . . . 8
2.4 Completeness . . . 19

3 Sequent Calculus **20**
3.1 Soundness . . . 21
3.2 Tableau Calculus Equivalence . . . 22
3.3 Completeness . . . 23

4 Completeness Revisited **24**

1 Common Notation

theory *Common* **imports** *FOL−Fitting.FOL-Fitting* **begin**

notation *FF* (‹⊥›)
notation *TT* (‹⊤›)

end

2 Tableau Calculus

theory *Tableau* **imports** *Common* **begin**

inductive *TC* :: ‹('a, 'b) form list ⇒ bool› (‹⊣ -› *0*) **where**
Basic: ‹⊣ Pred i l # Neg (Pred i l) # G›
| *BasicFF*: ‹⊣ ⊥ # G›
| *BasicNegTT*: ‹⊣ Neg ⊤ # G›
| *AlphaNegNeg*: ‹⊣ A # G ⟹ ⊣ Neg (Neg A) # G›
| *AlphaAnd*: ‹⊣ A # B # G ⟹ ⊣ And A B # G›
| *AlphaNegOr*: ‹⊣ Neg A # Neg B # G ⟹ ⊣ Neg (Or A B) # G›
| *AlphaNegImpl*: ‹⊣ A # Neg B # G ⟹ ⊣ Neg (Impl A B) # G›
| *BetaNegAnd*: ‹⊣ Neg A # G ⟹ ⊣ Neg B # G ⟹ ⊣ Neg (And A B) # G›
| *BetaOr*: ‹⊣ A # G ⟹ ⊣ B # G ⟹ ⊣ Or A B # G›
| *BetaImpl*: ‹⊣ Neg A # G ⟹ ⊣ B # G ⟹ ⊣ Impl A B # G›
| *GammaForall*: ‹⊣ subst A t 0 # G ⟹ ⊣ Forall A # G›
| *GammaNegExists*: ‹⊣ Neg (subst A t 0) # G ⟹ ⊣ Neg (Exists A) # G›
| *DeltaExists*: ‹⊣ subst A (App n []) 0 # G ⟹ news n (A # G) ⟹ ⊣ Exists A # G›
| *DeltaNegForall*: ‹⊣ Neg (subst A (App n []) 0) # G ⟹ news n (A # G) ⟹ ⊣ Neg (Forall A) # G›
| *Order*: ‹⊣ G ⟹ set G = set G' ⟹ ⊣ G'›

lemma *Shift*: ‹⊣ rotate1 G ⟹ ⊣ G›
by (*simp add*: *Order*)

lemma *Swap*: ‹⊣ B # A # G ⟹ ⊣ A # B # G›
by (*simp add*: *Order insert-commute*)

definition *tableauproof* :: ‹('a, 'b) form list ⇒ ('a, 'b) form ⇒ bool› **where**
‹tableauproof ps p ≡ (⊣ Neg p # ps)›

theorem *tableauNotAA*: ‹⊣ [Neg (Pred ''A'' []), Pred ''A'' []]›
by (*rule Shift*, *simp*) (*rule Basic*)

theorem *AndAnd*:
‹⊣ [And (Pred ''A'' []) (Pred ''B'' []), Neg (And (Pred ''B'' []) (Pred ''A'' []))]›
apply (*rule AlphaAnd*)
apply (*rule Shift*, *rule Shift*, *simp*)
apply (*rule BetaNegAnd*)

```
   apply (rule Shift, rule Shift, simp)
   apply (rule Basic)
  apply (rule Swap)
  apply (rule Basic)
  done
```

2.1 Soundness

```
lemma TC-soundness:
  ‹⊣ G ⟹ ∃ p ∈ set G. ¬ eval e f g p›
proof (induct G arbitrary: f rule: TC.induct)
  case (DeltaExists A n G)
  show ?case
  proof (rule ccontr)
    assume ‹¬ (∃ p ∈ set (Exists A # G). ¬ eval e f g p)›
    then have ∗: ‹∀ p ∈ set (Exists A # G). eval e f g p›
      by simp

    then obtain x where ‹eval (shift e 0 x) (f(n := λw. x)) g A›
      using ‹news n (A # G)› by auto
    then have ∗∗: ‹eval e (f(n := λw. x)) g (subst A (App n []) 0)›
      by simp

    have ‹∃ p ∈ set (subst A (App n []) 0 # G). ¬ eval e (f(n := λw. x)) g p›
      using DeltaExists by fast
    then consider
      ‹¬ eval e (f(n := λw. x)) g (subst A (App n []) 0)› |
      ‹∃ p ∈ set G. ¬ eval e (f(n := λw. x)) g p›
      by auto
    then show False
    proof cases
      case 1
      then show ?thesis
        using ∗∗ ..
    next
      case 2
      then obtain p where ‹¬ eval e (f(n := λw. x)) g p› ‹p ∈ set G›
        by blast
      then have ‹¬ eval e f g p›
           using ‹news n (A # G)› by (metis Ball-set set-subset-Cons subsetCE
upd-lemma)
      then show ?thesis
        using ∗ ‹p ∈ set G› by simp
    qed
  qed
next
  case (DeltaNegForall A n G)
  show ?case
  proof (rule ccontr)
```

```
    assume ‹¬ (∃ p ∈ set (Neg (Forall A) # G). ¬ eval e f g p)›
    then have *: ‹∀ p ∈ set (Neg (Forall A) # G). eval e f g p›
      by simp

    then obtain x where ‹eval (shift e 0 x) (f(n := λw. x)) g (Neg A)›
      using ‹news n (A # G)› by auto
    then have **: ‹eval e (f(n := λw. x)) g (Neg (subst A (App n []) 0))›
      by simp

    have ‹∃ p ∈ set (Neg (subst A (App n []) 0) # G). ¬ eval e (f(n := λw. x)) g
p›
      using DeltaNegForall by fast
    then consider
      ‹¬ eval e (f(n := λw. x)) g (Neg (subst A (App n []) 0))› |
      ‹∃ p ∈ set G. ¬ eval e (f(n := λw. x)) g p›
      by auto
    then show False
    proof cases
      case 1
      then show ?thesis
        using ** ..
    next
      case 2
      then obtain p where ‹¬ eval e (f(n := λw. x)) g p› ‹p ∈ set G›
        by blast
      then have ‹¬ eval e f g p›
          using ‹news n (A # G)› by (metis Ball-set set-subset-Cons subsetCE
upd-lemma)
      then show ?thesis
        using * ‹p ∈ set G› by simp
    qed
  qed
qed auto

theorem tableau-soundness:
  ‹tableauproof ps p ⟹ list-all (eval e f g) ps ⟹ eval e f g p›
  using TC-soundness unfolding tableauproof-def list-all-def by fastforce
```

2.2 Completeness for Closed Formulas

```
theorem infinite-nonempty: ‹infinite A ⟹ ∃ x. x ∈ A›
  by (simp add: ex-in-conv infinite-imp-nonempty)

theorem TCd-consistency:
  assumes inf-param: ‹infinite (UNIV::'a set)›
  shows ‹consistency {S::('a, 'b) form set. ∃ G. S = set G ∧ ¬ (⊣ G)}›
  unfolding consistency-def
proof (intro conjI allI impI notI)
  fix S :: ‹('a, 'b) form set›
```

```
assume ‹S ∈ {set G | G. ¬ (⊣ G)}› (is ‹S ∈ ?C›)
then obtain G :: ‹('a, 'b) form list›
  where *: ‹S = set G› and ‹¬ (⊣ G)›
  by blast

{ fix p ts
  assume ‹Pred p ts ∈ S ∧ Neg (Pred p ts) ∈ S›
  then show False
    using * Basic Order ‹¬ (⊣ G)› by fastforce }

{ assume ‹⊥ ∈ S›
  then show False
    using * BasicFF Order ‹¬ (⊣ G)› by fastforce }

{ assume ‹Neg ⊤ ∈ S›
  then show False
    using * BasicNegTT Order ‹¬ (⊣ G)› by fastforce }

{ fix Z
  assume ‹Neg (Neg Z) ∈ S›
  then have ‹¬ (⊣ Z # G)›
    using * AlphaNegNeg Order ‹¬ (⊣ G)›
    by (metis insert-absorb list.set(2))
  moreover have ‹S ∪ {Z} = set (Z # G)›
    using * by simp
  ultimately show ‹S ∪ {Z} ∈ ?C›
    by blast }

{ fix A B
  assume ‹And A B ∈ S›
  then have ‹¬ (⊣ A # B # G)›
    using * AlphaAnd Order ‹¬ (⊣ G)›
    by (metis insert-absorb list.set(2))
  moreover have ‹S ∪ {A, B} = set (A # B # G)›
    using * by simp
  ultimately show ‹S ∪ {A, B} ∈ ?C›
    by blast }

{ fix A B
  assume ‹Neg (Or A B) ∈ S›
  then have ‹¬ (⊣ Neg A # Neg B # G)›
    using * AlphaNegOr Order ‹¬ (⊣ G)›
    by (metis insert-absorb list.set(2))
  moreover have ‹S ∪ {Neg A, Neg B} = set (Neg A # Neg B # G)›
    using * by simp
  ultimately show ‹S ∪ {Neg A, Neg B} ∈ ?C›
    by blast }

{ fix A B
```

```
  assume ‹Neg (Impl A B) ∈ S›
  then have ‹¬ (⊣ A # Neg B # G)›
    using * AlphaNegImpl Order ‹¬ (⊣ G)›
    by (metis insert-absorb list.set(2))
  moreover have ‹{A, Neg B} ∪ S = set (A # Neg B # G)›
    using * by simp
  ultimately show ‹S ∪ {A, Neg B} ∈ ?C›
    by blast }

{ fix A B
  assume ‹Or A B ∈ S›
  then have ‹¬ (⊣ A # G) ∨ ¬ (⊣ B # G)›
    using * BetaOr Order ‹¬ (⊣ G)›
    by (metis insert-absorb list.set(2))
  then show ‹S ∪ {A} ∈ ?C ∨ S ∪ {B} ∈ ?C›
    using * by auto }

{ fix A B
  assume ‹Neg (And A B) ∈ S›
  then have ‹¬ (⊣ Neg A # G) ∨ ¬ (⊣ Neg B # G)›
    using * BetaNegAnd Order ‹¬ (⊣ G)›
    by (metis insert-absorb list.set(2))
  then show ‹S ∪ {Neg A} ∈ ?C ∨ S ∪ {Neg B} ∈ ?C›
    using * by auto }

{ fix A B
  assume ‹Impl A B ∈ S›
  then have ‹¬ (⊣ Neg A # G) ∨ ¬ (⊣ B # G)›
    using * BetaImpl Order ‹¬ (⊣ G)›
    by (metis insert-absorb list.set(2))
  then show ‹S ∪ {Neg A} ∈ ?C ∨ S ∪ {B} ∈ ?C›
    using * by auto }

{ fix P and t :: ‹'a term›
  assume ‹Forall P ∈ S›
  then have ‹¬ (⊣ subst P t 0 # G)›
    using * GammaForall Order‹¬ (⊣ G)›
    by (metis insert-absorb list.set(2))
  moreover have ‹S ∪ {subst P t 0} = set (subst P t 0 # G)›
    using * by simp
  ultimately show ‹S ∪ {subst P t 0} ∈ ?C›
    by blast }

{ fix P and t :: ‹'a term›
  assume ‹Neg (Exists P) ∈ S›
  then have ‹¬ (⊣ Neg (subst P t 0) # G)›
    using * GammaNegExists Order ‹¬ (⊣ G)›
    by (metis insert-absorb list.set(2))
  moreover have ‹S ∪ {Neg (subst P t 0)} = set (Neg (subst P t 0) # G)›
```

```
      using ∗ by simp
    ultimately show ‹S ∪ {Neg (subst P t 0)} ∈ ?C›
      by blast }

  { fix P
    assume ‹Exists P ∈ S›
    have ‹finite ((⋃p ∈ set G. params p) ∪ params P)›
      by simp
    then have ‹infinite (− ((⋃p ∈ set G. params p) ∪ params P))›
      using inf-param Diff-infinite-finite finite-compl infinite-UNIV-listI by blast
    then obtain x where ∗∗: ‹x ∈ − ((⋃p ∈ set G. params p) ∪ params P)›
      using infinite-imp-nonempty by blast
    then have ‹news x (P # G)›
      using Ball-set-list-all by auto
    then have ‹¬ (⊣ subst P (App x []) 0 # G)›
      using ∗ ‹Exists P ∈ S› Order DeltaExists ‹¬ (⊣ G)›
      by (metis insert-absorb list.set(2))
    moreover have ‹S ∪ {subst P (App x []) 0} = set (subst P (App x []) 0 #
G)›
      using ∗ by simp
    ultimately show ‹∃ x. S ∪ {subst P (App x []) 0} ∈ ?C›
      by blast }

  { fix P
    assume ‹Neg (Forall P) ∈ S›
    have ‹finite ((⋃p ∈ set G. params p) ∪ params P)›
      by simp
    then have ‹infinite (− ((⋃p ∈ set G. params p) ∪ params P))›
      using inf-param Diff-infinite-finite finite-compl infinite-UNIV-listI by blast
    then obtain x where ∗∗: ‹x ∈ − ((⋃p ∈ set G. params p) ∪ params P)›
      using infinite-imp-nonempty by blast
    then have ‹news x (P # G)›
      using Ball-set-list-all by auto
    then have ‹¬ (⊣ Neg (subst P (App x []) 0) # G)›
      using ∗ ‹Neg (Forall P) ∈ S› Order DeltaNegForall ‹¬ (⊣ G)›
      by (metis insert-absorb list.set(2))
    moreover have ‹S ∪ {Neg (subst P (App x []) 0)} = set (Neg (subst P (App
x []) 0) # G)›
      using ∗ by simp
    ultimately show ‹∃ x. S ∪ {Neg (subst P (App x []) 0)} ∈ ?C›
      by blast }
qed

theorem tableau-completeness':
  fixes p :: ‹(nat, nat) form›
  assumes ‹closed 0 p›
    and ‹list-all (closed 0) ps›
    and mod: ‹∀ (e :: nat ⇒ nat hterm) f g. list-all (eval e f g) ps ⟶ eval e f g p›
  shows ‹tableauproof ps p›
```

```
proof (rule ccontr)
  fix e
  assume ‹¬ tableauproof ps p›

  let ?S = ‹set (Neg p # ps)›
  let ?C = ‹{set (G :: (nat, nat) form list) | G. ¬ (⊣ G)}›
  let ?f = HApp
  let ?g = ‹(λa ts. Pred a (terms-of-hterms ts) ∈ Extend ?S
              (mk-finite-char (mk-alt-consistency (close ?C))) from-nat)›

  from ‹list-all (closed 0) ps›
  have ‹∀ p ∈ set ps. closed 0 p›
    by (simp add: list-all-iff)

  { fix x
    assume ‹x ∈ ?S›
    moreover have ‹consistency ?C›
      using TCd-consistency by blast
    moreover have ‹?S ∈ ?C›
      using ‹¬ tableauproof ps p› unfolding tableauproof-def by blast
    moreover have ‹infinite (− (⋃ p ∈ ?S. params p))›
      by (simp add: Compl-eq-Diff-UNIV infinite-UNIV-listI)
    moreover note ‹closed 0 p› ‹∀ p ∈ set ps. closed 0 p› ‹x ∈ ?S›
    then have ‹closed 0 x› by auto
    ultimately have ‹eval e ?f ?g x›
      using model-existence by blast }
  then have ‹list-all (eval e ?f ?g) (Neg p # ps)›
    by (simp add: list-all-iff)
  moreover have ‹eval e ?f ?g (Neg p)›
    using calculation by simp
  moreover have ‹list-all (eval e ?f ?g) ps›
    using calculation by simp
  then have ‹eval e ?f ?g p›
    using mod by blast
  ultimately show False by simp
qed
```

2.3 Open Formulas

```
lemma TC-psubst:
  fixes f :: ‹'a ⇒ 'a›
  assumes inf-params: ‹infinite (UNIV :: 'a set)›
  shows ‹⊣ G ⟹ ⊣ map (psubst f) G›
proof (induct G arbitrary: f rule: TC.induct)
  case (DeltaExists A n G)
  let ?params = ‹params A ∪ (⋃ p ∈ set G. params p)›

  have ‹finite ?params›
    by simp
```

then obtain *fresh* **where** $*$: ‹$fresh \notin$ *?params* $\cup$ $\{n\}$ $\cup$ *image f ?params*›
  using *ex-new-if-finite inf-params*
  by (*metis finite.emptyI finite.insertI finite-UnI finite-imageI*)

let *?f* = ‹$f(n := fresh)$›

have ‹*news n* (A # G)›
  using *DeltaExists* **by** *blast*
then have ‹*new fresh* (*psubst ?f A*)› ‹*news fresh* (*map* (*psubst ?f*) *G*)›
  using $*$ *new-psubst-image news-psubst* **by** (*fastforce simp add*: *image-Un*)+
then have *G*: ‹*map* (*psubst ?f*) *G* = *map* (*psubst f*) *G*›
  using *DeltaExists*
  by (*metis* (*mono-tags*, *lifting*) *Ball-set insertCI list.set*(*2*) *map-eq-conv psubst-upd*)

have ‹$\dashv$ *psubst ?f* (*subst A* (*App n* []) *0*) # *map* (*psubst ?f*) *G*›
  using *DeltaExists* **by** (*metis list.simps*(*9*))
then have ‹$\dashv$ *subst* (*psubst ?f A*) (*App fresh* []) *0* # *map* (*psubst ?f*) *G*›
  by *simp*
moreover have ‹*news fresh* (*map* (*psubst ?f*) (*A* # *G*))›
  using ‹*new fresh* (*psubst ?f A*)› ‹*news fresh* (*map* (*psubst ?f*) *G*)› **by** *simp*
then have ‹*news fresh* (*psubst ?f A* # *map* (*psubst ?f*) *G*)›
  by *simp*
ultimately have ‹$\dashv$ *map* (*psubst ?f*) (*Exists A* # *G*)›
  using *TC.DeltaExists* **by** *fastforce*
then show *?case*
  using *DeltaExists G* **by** *simp*
next
case (*DeltaNegForall A n G*)
let *?params* = ‹*params A* $\cup$ ($\bigcup p \in$ *set G*. *params p*)›

have ‹*finite ?params*›
  by *simp*
then obtain *fresh* **where** $*$: ‹$fresh \notin$ *?params* $\cup$ $\{n\}$ $\cup$ *image f ?params*›
  using *ex-new-if-finite inf-params*
  by (*metis finite.emptyI finite.insertI finite-UnI finite-imageI*)

let *?f* = ‹$f(n := fresh)$›

have ‹*news n* (A # G)›
  using *DeltaNegForall* **by** *blast*
then have ‹*new fresh* (*psubst ?f A*)› ‹*news fresh* (*map* (*psubst ?f*) *G*)›
  using $*$ *new-psubst-image news-psubst* **by** (*fastforce simp add*: *image-Un*)+
then have *G*: ‹*map* (*psubst ?f*) *G* = *map* (*psubst f*) *G*›
  using *DeltaNegForall*
  by (*metis* (*mono-tags*, *lifting*) *Ball-set insertCI list.set*(*2*) *map-eq-conv psubst-upd*)

have ‹$\dashv$ *psubst ?f* (*Neg* (*subst A* (*App n* []) *0*)) # *map* (*psubst ?f*) *G*›
  using *DeltaNegForall* **by** (*metis list.simps*(*9*))
then have ‹$\dashv$ *Neg* (*subst* (*psubst ?f A*) (*App fresh* []) *0*) # *map* (*psubst ?f*) *G*›

```
    by simp
  moreover have ‹news fresh (map (psubst ?f) (A # G))›
    using ‹new fresh (psubst ?f A)› ‹news fresh (map (psubst ?f) G)› by simp
  then have ‹news fresh (psubst ?f A # map (psubst ?f) G)›
    by simp
  ultimately have ‹⊣ map (psubst ?f) (Neg (Forall A) # G)›
    using TC.DeltaNegForall by fastforce
  then show ?case
    using DeltaNegForall G by simp
next
  case (Order G G')
  then show ?case
    using Order TC.Order set-map by metis
qed (auto intro: TC.intros)

lemma subcs-map: ‹subcs c s G = map (subc c s) G›
  by (induct G) simp-all

lemma TC-subcs:
  fixes G :: ‹('a, 'b) form list›
  assumes inf-params: ‹infinite (UNIV :: 'a set)›
  shows ‹⊣ G ⟹ ⊣ subcs c s G›
proof (induct G arbitrary: c s rule: TC.induct)
  case (GammaForall A t G)
  let ?params = ‹params A ∪ (⋃p ∈ set G. params p) ∪ paramst s ∪ paramst t ∪ {c}›

  have ‹finite ?params›
    by simp
  then obtain fresh where fresh: ‹fresh ∉ ?params›
    using ex-new-if-finite inf-params by metis

  let ?f = ‹id(c := fresh)›
  let ?g = ‹id(fresh := c)›
  let ?s = ‹psubstt ?f s›

  have s: ‹psubstt ?g ?s = s›
    using fresh psubst-new-away' by simp
  have ‹subc (?g c) (psubstt ?g ?s) (psubst ?g (Forall A)) = subc c s (Forall A)›
    using fresh by simp
  then have A: ‹psubst ?g (subc c ?s (Forall A)) = subc c s (Forall A)›
    using fun-upd-apply id-def subc-psubst UnCI fresh params.simps(8) by metis
  have ‹∀x ∈ (⋃p ∈ set (Forall A # G). params p). x ≠ c ⟶ ?g x ≠ ?g c›
    using fresh by auto
  moreover have ‹map (psubst ?g) (Forall A # G) = Forall A # G›
    using fresh by (induct G) simp-all
  ultimately have G: ‹map (psubst ?g) (subcs c ?s (Forall A # G)) = subcs c s (Forall A # G)›
    using s A by (simp add: subcs-psubst)
```

```
  have ‹new-term c ?s›
    using fresh psubst-new-free′ by fast
  then have ‹⊣ subc c ?s (subst A (subc-term c ?s t) 0) # subcs c ?s G›
    using GammaForall by (metis new-subc-put subcs.simps(2))
  moreover have ‹new-term c (subc-term c ?s t)›
    using ‹new-term c ?s› new-subc-same′ by simp
  ultimately have ‹⊣ subst (subc c (liftt ?s) A) (subc-term c ?s t) 0 # subcs c ?s
G›
    by simp
  moreover have ‹Forall (subc c (liftt ?s) A) ∈ set (subcs c ?s (Forall A # G))›
    by simp
  ultimately have ‹⊣ subcs c ?s (Forall A # G)›
    using TC.GammaForall by simp
  then have ‹⊣ map (psubst ?g) (subcs c ?s (Forall A # G))›
    using TC-psubst inf-params by blast
  then show ‹⊣ subcs c s (Forall A # G)›
    using G by simp
next
  case (GammaNegExists A t G)
  let ?params = ‹params A ∪ (⋃ p ∈ set G. params p) ∪ paramst s ∪ paramst t ∪
{c}›

  have ‹finite ?params›
    by simp
  then obtain fresh where fresh: ‹fresh ∉ ?params›
    using ex-new-if-finite inf-params by metis

  let ?f = ‹id(c := fresh)›
  let ?g = ‹id(fresh := c)›
  let ?s = ‹psubstt ?f s›

  have s: ‹psubstt ?g ?s = s›
    using fresh psubst-new-away′ by simp
  have ‹subc (?g c) (psubstt ?g ?s) (psubst ?g (Neg (Exists A))) = subc c s (Neg
(Exists A))›
    using fresh by simp
  then have A: ‹psubst ?g (subc c ?s (Neg (Exists A))) = subc c s (Neg (Exists
A))›
    using fun-upd-apply id-def subc-psubst UnCI fresh params.simps(7,9) by metis

  have ‹∀ x ∈ (⋃ p ∈ set (Neg (Exists A) # G). params p). x ≠ c ⟶ ?g x ≠ ?g
c›
    using fresh by auto
  moreover have ‹map (psubst ?g) (Neg (Exists A) # G) = Neg (Exists A) # G›
    using fresh by (induct G) simp-all
  ultimately have G: ‹map (psubst ?g) (subcs c ?s (Neg (Exists A) # G)) =
      subcs c s (Neg (Exists A) # G)›
    using s A by (simp add: subcs-psubst)
```

have ‹*new-term c ?s*›
  using *fresh psubst-new-free′* **by** *fast*
then have ‹⊣ *Neg* (*subc c ?s* (*subst A* (*subc-term c ?s t*) *0*)) # *subcs c ?s G*›
  using *GammaNegExists* **by** (*metis new-subc-put subc.simps*(*4*) *subcs.simps*(*2*))
moreover have ‹*new-term c* (*subc-term c ?s t*)›
  using ‹*new-term c ?s*› *new-subc-same′* **by** *simp*
ultimately have ‹⊣ *Neg* (*subst* (*subc c* (*liftt ?s*) *A*) (*subc-term c ?s t*) *0*) # *subcs c ?s G*›
  by *simp*
moreover have ‹*Neg* (*Exists* (*subc c* (*liftt ?s*) *A*)) ∈ *set* (*subcs c ?s* (*Neg* (*Exists A*) # *G*))›
  by *simp*
ultimately have ‹⊣ *subcs c ?s* (*Neg* (*Exists A*) # *G*)›
  using *TC.GammaNegExists* **by** *simp*
then have ‹⊣ *map* (*psubst ?g*) (*subcs c ?s* (*Neg* (*Exists A*) # *G*))›
  using *TC-psubst inf-params* **by** *blast*
then show ‹⊣ *subcs c s* (*Neg* (*Exists A*) # *G*)›
  using *G* **by** *simp*
next
case (*DeltaExists A n G*)
then show *?case*
proof (*cases* ‹*c* = *n*›)
  case *True*
  then have ‹⊣ *Exists A* # *G*›
    using *DeltaExists TC.DeltaExists* **by** *metis*
  moreover have ‹*new c A*› **and** ‹*news c G*›
    using *DeltaExists True* **by** *simp-all*
  ultimately show *?thesis*
    by (*simp add*: *subcs-news*)
next
  case *False*
  let *?params* = ‹*params A* ∪ (⋃*p* ∈ *set G*. *params p*) ∪ *paramst s* ∪ {*c*} ∪ {*n*}›

  have ‹*finite ?params*›
    by *simp*
  then obtain *fresh* **where** *fresh*: ‹*fresh* ∉ *?params*›
    using *ex-new-if-finite inf-params* **by** *metis*

  let *?s* = ‹*psubstt* (*id*(*n* := *fresh*)) *s*›
  let *?f* = ‹*id*(*n* := *fresh*, *fresh* := *n*)›

  have *f*: ‹∀ *x* ∈ *?params*. *x* ≠ *c* ⟶ *?f x* ≠ *?f c*›
    using *fresh* **by** *simp*

  have ‹*new-term n ?s*›
    using *fresh psubst-new-free′* **by** *fast*
  then have ‹*psubstt ?f ?s* = *psubstt* (*id*(*fresh* := *n*)) *?s*›
    by (*metis fun-upd-twist psubstt-upd*(*1*))

```
  then have psubst-s: ‹psubstt ?f ?s = s›
    using fresh psubst-new-away' by simp

  have ‹?f c = c› and ‹new-term c (App fresh [])›
    using False fresh by auto

  have ‹psubst ?f (subc c ?s (subst A (App n []) 0)) =
    subc (?f c) (psubstt ?f ?s) (psubst ?f (subst A (App n []) 0))›
    by (simp add: subc-psubst)
  also have ‹... = subc c s (subst (psubst ?f A) (App fresh []) 0)›
    using ‹?f c = c› psubst-subst psubst-s by simp
  also have ‹... = subc c s (subst A (App fresh []) 0)›
    using DeltaExists fresh by simp
  finally have psubst-A: ‹psubst ?f (subc c ?s (subst A (App n []) 0)) =
      subst (subc c (liftt s) A) (App fresh []) 0›
    using ‹new-term c (App fresh [])› by simp

  have ‹news n G›
    using DeltaExists by simp
  moreover have ‹news fresh G›
    using fresh by (induct G) simp-all
  ultimately have ‹map (psubst ?f) G = G›
    by (induct G) simp-all
  moreover have ‹∀ x ∈ ⋃ p ∈ set G. params p. x ≠ c ⟶ ?f x ≠ ?f c›
    by auto
  ultimately have psubst-G: ‹map (psubst ?f) (subcs c ?s G) = subcs c s G›
    using ‹?f c = c› psubst-s by (simp add: subcs-psubst)

  have ‹⊣ subc c ?s (subst A (App n []) 0) # subcs c ?s G›
    using DeltaExists by simp
  then have ‹⊣ psubst ?f (subc c ?s (subst A (App n []) 0)) # map (psubst ?f)
(subcs c ?s G)›
    using TC-psubst inf-params DeltaExists.hyps(3) by fastforce
  then have ‹⊣ psubst ?f (subc c ?s (subst A (App n []) 0)) # subcs c s G›
    using psubst-G by simp
  then have sub-A: ‹⊣ subst (subc c (liftt s) A) (App fresh []) 0 # subcs c s G›
    using psubst-A by simp

  have ‹new-term fresh s›
    using fresh by simp
  then have ‹new-term fresh (liftt s)›
    by simp
  then have ‹new fresh (subc c (liftt s) A)›
    using fresh new-subc by simp
  moreover have ‹news fresh (subcs c s G)›
    using ‹news fresh G› ‹new-term fresh s› news-subcs by fast
  ultimately show ‹⊣ subcs c s (Exists A # G)›
    using TC.DeltaExists sub-A by fastforce
qed
```

next
  case (*DeltaNegForall A n G*)
  then show *?case*
  proof (*cases* ‹*c* = *n*›)
    case *True*
    then have ‹⊣ *Neg* (*Forall A*) # *G*›
      using *DeltaNegForall TC.DeltaNegForall* **by** *metis*
    moreover have ‹*new c A*› **and** ‹*news c G*›
      using *DeltaNegForall True* **by** *simp-all*
    ultimately show *?thesis*
      by (*simp add*: *subcs-news*)
  next
    case *False*
    let *?params* = ‹*params A* ∪ (⋃ *p* ∈ *set G*. *params p*) ∪ *paramst s* ∪ {*c*} ∪ {*n*}›

    have ‹*finite ?params*›
      by *simp*
    then obtain *fresh* **where** *fresh*: ‹*fresh* ∉ *?params*›
      using *ex-new-if-finite inf-params* **by** *metis*

    let *?s* = ‹*psubstt* (*id*(*n* := *fresh*)) *s*›
    let *?f* = ‹*id*(*n* := *fresh*, *fresh* := *n*)›

    have *f*: ‹∀ *x* ∈ *?params*. *x* ≠ *c* ⟶ *?f x* ≠ *?f c*›
      using *fresh* **by** *simp*

    have ‹*new-term n ?s*›
      using *fresh psubst-new-free′* **by** *fast*
    then have ‹*psubstt ?f ?s* = *psubstt* (*id*(*fresh* := *n*)) *?s*›
      using *fun-upd-twist psubstt-upd*(*1*) **by** *metis*
    then have *psubst-s*: ‹*psubstt ?f ?s* = *s*›
      using *fresh psubst-new-away′* **by** *simp*

    have ‹*?f c* = *c*› **and** ‹*new-term c* (*App fresh* [])›
      using *False fresh* **by** *auto*

    have ‹*psubst ?f* (*subc c ?s* (*Neg* (*subst A* (*App n* []) *0*))) =
      subc (*?f c*) (*psubstt ?f ?s*) (*psubst ?f* (*Neg* (*subst A* (*App n* []) *0*)))›
      by (*simp add*: *subc-psubst*)
    also have ‹. . . = *subc c s* (*Neg* (*subst* (*psubst ?f A*)(*App fresh* []) *0*))›
      using ‹*?f c* = *c*› *psubst-subst psubst-s* **by** *simp*
    also have ‹. . . = *subc c s* (*Neg* (*subst A* (*App fresh* []) *0*))›
      using *DeltaNegForall fresh* **by** *simp*
    finally have *psubst-A*: ‹*psubst ?f* (*subc c ?s* (*Neg* (*subst A* (*App n* []) *0*))) =
        Neg (*subst* (*subc c* (*liftt s*) *A*) (*App fresh* []) *0*)›
      using ‹*new-term c* (*App fresh* [])› **by** *simp*

    have ‹*news n G*›
      using *DeltaNegForall* **by** *simp*

```
    moreover have ‹news fresh G›
      using fresh by (induct G) simp-all
    ultimately have ‹map (psubst ?f) G = G›
      by (induct G) simp-all
    moreover have ‹∀ x ∈ ⋃ p ∈ set G. params p. x ≠ c ⟶ ?f x ≠ ?f c›
      by auto
    ultimately have psubst-G: ‹map (psubst ?f) (subcs c ?s G) = subcs c s G›
      using ‹?f c = c› psubst-s by (simp add: subcs-psubst)

    have ‹⊣ subc c ?s (Neg (subst A (App n []) 0)) # subcs c ?s G›
      using DeltaNegForall by simp
    then have ‹⊣ psubst ?f (subc c ?s (Neg (subst A (App n []) 0)))
              # map (psubst ?f) (subcs c ?s G)›
      using TC-psubst inf-params DeltaNegForall.hyps(3) by fastforce
    then have ‹⊣ psubst ?f (subc c ?s (Neg (subst A (App n []) 0))) # subcs c s
G›
      using psubst-G by simp
    then have sub-A: ‹⊣ Neg (subst (subc c (liftt s) A) (App fresh []) 0) # subcs
c s G›
      using psubst-A by simp

    have ‹new-term fresh s›
      using fresh by simp
    then have ‹new-term fresh (liftt s)›
      by simp
    then have ‹new fresh (subc c (liftt s) A)›
      using fresh new-subc by simp
    moreover have ‹news fresh (subcs c s G)›
      using ‹news fresh G› ‹new-term fresh s› news-subcs by fast
    ultimately show ‹⊣ subcs c s (Neg (Forall A) # G)›
      using TC.DeltaNegForall sub-A by fastforce
  qed
next
  case (Order G G')
  then show ?case
    using TC.Order set-map subcs-map by metis
qed (auto intro: TC.intros)

lemma TC-map-subc:
  fixes G :: ‹('a, 'b) form list›
  assumes inf-params: ‹infinite (UNIV :: 'a set)›
  shows ‹⊣ G ⟹ ⊣ map (subc c s) G›
  using assms TC-subcs subcs-map by metis

lemma ex-all-closed: ‹∃ m. list-all (closed m) G›
proof (induct G)
  case Nil
  then show ?case
    by simp
```

```
next
  case (Cons a G)
  then show ?case
    unfolding list-all-def
    using ex-closed closed-mono
    by (metis Ball-set list-all-simps(1) nat-le-linear)
qed

primrec sub-consts :: ‹'a list ⇒ ('a, 'b) form ⇒ ('a, 'b) form› where
  ‹sub-consts [] p = p›
| ‹sub-consts (c # cs) p = sub-consts cs (subst p (App c []) (length cs))›

lemma valid-sub-consts:
  assumes ‹∀ (e :: nat ⇒ 'a) f g. eval e f g p›
  shows ‹eval (e :: nat => 'a) f g (sub-consts cs p)›
  using assms by (induct cs arbitrary: p) simp-all

lemma closed-sub' [simp]:
  assumes ‹k ≤ m› shows
    ‹closedt (Suc m) t = closedt m (substt t (App c []) k)›
    ‹closedts (Suc m) l = closedts m (substts l (App c []) k)›
  using assms by (induct t and l rule: closedt.induct closedts.induct) auto

lemma closed-sub: ‹k ≤ m ⟹ closed (Suc m) p = closed m (subst p (App c [])
k)›
  by (induct p arbitrary: m k) simp-all

lemma closed-sub-consts: ‹length cs = k ⟹ closed m (sub-consts cs p) = closed
(m + k) p›
proof (induct cs arbitrary: k p)
  case Nil
  then show ?case
    by simp
next
  case (Cons c cs)
  then show ?case
    using closed-sub by fastforce
qed

lemma map-sub-consts-Nil: ‹map (sub-consts []) G = G›
  by (induct G) simp-all

primrec conjoin :: ‹('a, 'b) form list ⇒ ('a, 'b) form› where
  ‹conjoin [] = Neg ⊥›
| ‹conjoin (p # ps) = And p (conjoin ps)›

lemma eval-conjoin: ‹list-all (eval e f g) G = eval e f g (conjoin G)›
  by (induct G) simp-all
```

lemma *valid-sub*:
  fixes e :: ‹$nat \Rightarrow 'a$›
  assumes ‹$\forall (e :: nat \Rightarrow 'a)\ f\ g.\ eval\ e\ f\ g\ p \longrightarrow eval\ e\ f\ g\ q$›
  shows ‹$eval\ e\ f\ g\ (subst\ p\ t\ m) \longrightarrow eval\ e\ f\ g\ (subst\ q\ t\ m)$›
  using *assms* **by** *simp*

lemma *eval-sub-consts*:
  fixes e :: ‹$nat \Rightarrow 'a$›
  assumes ‹$\forall (e :: nat \Rightarrow 'a)\ f\ g.\ eval\ e\ f\ g\ p \longrightarrow eval\ e\ f\ g\ q$›
    and ‹$eval\ e\ f\ g\ (sub\text{-}consts\ cs\ p)$›
  shows ‹$eval\ e\ f\ g\ (sub\text{-}consts\ cs\ q)$›
  using *assms*
proof (*induct cs arbitrary*: *p q*)
  case *Nil*
  then show *?case*
    by *simp*
next
  case (*Cons c cs*)
  then show *?case*
    by (*metis sub-consts.simps*(*2*) *subst-lemma*)
qed

lemma *sub-consts-And* [*simp*]: ‹$sub\text{-}consts\ cs\ (And\ p\ q) = And\ (sub\text{-}consts\ cs\ p)$ $(sub\text{-}consts\ cs\ q)$›
  by (*induct cs arbitrary*: *p q*) *simp-all*

lemma *sub-consts-conjoin*:
  ‹$eval\ e\ f\ g\ (sub\text{-}consts\ cs\ (conjoin\ G)) = eval\ e\ f\ g\ (conjoin\ (map\ (sub\text{-}consts\ cs)$ $G))$›
proof (*induct G*)
  case *Nil*
  then show *?case*
    by (*induct cs*) *simp-all*
next
  case (*Cons p G*)
  then show *?case*
    using *sub-consts-And* **by** *simp*
qed

lemma *all-sub-consts-conjoin*:
  ‹$list\text{-}all\ (eval\ e\ f\ g)\ (map\ (sub\text{-}consts\ cs)\ G) = eval\ e\ f\ g\ (sub\text{-}consts\ cs\ (conjoin$ $G))$›
  by (*induct G*) (*simp-all add*: *valid-sub-consts*)

lemma *valid-all-sub-consts*:
  fixes e :: ‹$nat \Rightarrow 'a$›
  assumes ‹$\forall (e :: nat \Rightarrow 'a)\ f\ g.\ list\text{-}all\ (eval\ e\ f\ g)\ G \longrightarrow eval\ e\ f\ g\ p$›
  shows ‹$list\text{-}all\ (eval\ e\ f\ g)\ (map\ (sub\text{-}consts\ cs)\ G) \longrightarrow eval\ e\ f\ g\ (sub\text{-}consts\ cs$ $p)$›

using *assms eval-conjoin eval-sub-consts all-sub-consts-conjoin* **by** *metis*

lemma *TC-vars-for-consts*:
fixes G :: ‹(′*a*, ′*b*) *form list*›
assumes ‹*infinite* (*UNIV* :: ′*a set*)›
shows ‹$\dashv$ $G \Longrightarrow \dashv$ *map* (λ*p*. *vars-for-consts p cs*) G›
proof (*induct cs*)
case *Nil*
then show *?case*
by *simp*
next
case (*Cons c cs*)
have ‹($\dashv$ *map* (λ*p*. *vars-for-consts p* (*c* # *cs*)) *G*) =
($\dashv$ *map* (λ*p*. *subc c* (*Var* (*length cs*)) (*vars-for-consts p cs*)) *G*)›
by *simp*
also have ‹. . . = ($\dashv$ *map* (*subc c* (*Var* (*length cs*)) *o* (λ*p*. *vars-for-consts p cs*))
G)›
unfolding *comp-def* **by** *simp*
also have ‹. . . = ($\dashv$ *map* (*subc c* (*Var* (*length cs*))) (*map* (λ*p*. *vars-for-consts p*
cs) *G*))›
by *simp*
finally show *?case*
using *Cons TC-map-subc assms* **by** *metis*
qed

lemma *vars-for-consts-sub-consts*:
‹*closed* (*length cs*) *p* $\Longrightarrow$ *list-all* (λ*c*. *new c p*) *cs* $\Longrightarrow$ *distinct cs* $\Longrightarrow$
vars-for-consts (*sub-consts cs p*) *cs* = *p*›
proof (*induct cs arbitrary*: *p*)
case (*Cons c cs*)
then show *?case*
using *subst-new-all closed-sub* **by** *force*
qed *simp*

lemma *all-vars-for-consts-sub-consts*:
‹*list-all* (*closed* (*length cs*)) *G* $\Longrightarrow$ *list-all* (λ*c*. *list-all* (*new c*) *G*) *cs* $\Longrightarrow$ *distinct*
cs $\Longrightarrow$
map (λ*p*. *vars-for-consts p cs*) (*map* (*sub-consts cs*) *G*) = *G*›
using *vars-for-consts-sub-consts* **unfolding** *list-all-def*
by (*induct G*) *fastforce+*

lemma *new-conjoin*: ‹*new c* (*conjoin G*) $\Longrightarrow$ *list-all* (*new c*) *G*›
by (*induct G*) *simp-all*

lemma *all-fresh-constants*:
fixes G :: ‹(′*a*, ′*b*) *form list*›
assumes ‹*infinite* (*UNIV* :: ′*a set*)›
shows ‹∃ *cs*. *length cs* = *m* ∧ *list-all* (λ*c*. *list-all* (*new c*) *G*) *cs* ∧ *distinct cs*›
proof −

obtain *cs* **where** ‹*length cs* = *m*› ‹*list-all* (*λc. new c* (*conjoin G*)) *cs*› ‹*distinct cs*›
  using *assms fresh-constants* **by** *blast*
 then show *?thesis*
  using *new-conjoin* **unfolding** *list-all-def* **by** *metis*
qed

lemma *sub-consts-Neg*: ‹*sub-consts cs* (*Neg p*) = *Neg* (*sub-consts cs p*)›
 by (*induct cs arbitrary*: *p*) *simp-all*

2.4 Completeness

theorem *tableau-completeness*:
 fixes *G* :: ‹(*nat*, *nat*) *form list*›
 assumes ‹∀ (*e* :: *nat* ⇒ *nat hterm*) *f g*. *list-all* (*eval e f g*) *G* ⟶ *eval e f g p*›
 shows ‹*tableauproof G p*›
proof −
 obtain *m* **where** ∗: ‹*list-all* (*closed m*) (*p* # *G*)›
  using *ex-all-closed* **by** *blast*
 moreover obtain *cs* **where** ∗∗:
  ‹*length cs* = *m*›
  ‹*distinct cs*›
  ‹*list-all* (*λc. list-all* (*new c*) (*p* # *G*)) *cs*›
  using *all-fresh-constants* **by** *blast*
 ultimately have ‹*closed 0* (*sub-consts cs p*)›
  using *closed-sub-consts* **by** *fastforce*
 moreover have ‹*list-all* (*closed 0*) (*map* (*sub-consts cs*) *G*)›
  using *closed-sub-consts* ∗ ‹*length cs* = *m*› **by** (*induct G*) *fastforce*+

 moreover have ‹∀ (*e* :: *nat* ⇒ *nat hterm*) *f g*. *list-all* (*eval e f g*) (*map* (*sub-consts cs*) *G*) ⟶
  eval e f g (*sub-consts cs p*)›
  using *assms valid-all-sub-consts* **by** *blast*
 ultimately have ‹⊣ *Neg* (*sub-consts cs p*) # *map* (*sub-consts cs*) *G*›
  using *tableau-completeness′* **unfolding** *tableauproof-def* **by** *simp*
 then have ‹⊣ *map* (*sub-consts cs*) (*Neg p* # *G*)›
  by (*simp add*: *sub-consts-Neg*)
 then have ‹⊣ *map* (*λp. vars-for-consts p cs*) (*map* (*sub-consts cs*) (*Neg p* # *G*))›
  using *TC-vars-for-consts* **by** *blast*
 then show *?thesis*
  unfolding *tableauproof-def*
  using *all-vars-for-consts-sub-consts*[**where** *G*=‹*Neg p* # *G*›] ∗ ∗∗ **by** *simp*
qed

corollary
 fixes *p* :: ‹(*nat*, *nat*) *form*›
 assumes ‹∀ (*e* :: *nat* ⇒ *nat hterm*) *f g*. *eval e f g p*›
 shows ‹⊣ [*Neg p*]›
 using *assms tableau-completeness* **unfolding** *tableauproof-def* **by** *simp*

end

3 Sequent Calculus

theory *Sequent* **imports** *Tableau* **begin**

inductive *SC* :: ‹('a, 'b) form list ⇒ bool› (‹⊢ -› 0) **where**
Basic: ‹⊢ *Pred i l* # *Neg* (*Pred i l*) # *G*›
| *BasicNegFF*: ‹⊢ *Neg* ⊥ # *G*›
| *BasicTT*: ‹⊢ ⊤ # *G*›
| *AlphaNegNeg*: ‹⊢ *A* # *G* ⟹ ⊢ *Neg* (*Neg A*) # *G*›
| *AlphaNegAnd*: ‹⊢ *Neg A* # *Neg B* # *G* ⟹ ⊢ *Neg* (*And A B*) # *G*›
| *AlphaOr*: ‹⊢ *A* # *B* # *G* ⟹ ⊢ *Or A B* # *G*›
| *AlphaImpl*: ‹⊢ *Neg A* # *B* # *G* ⟹ ⊢ *Impl A B* # *G*›
| *BetaAnd*: ‹⊢ *A* # *G* ⟹ ⊢ *B* # *G* ⟹ ⊢ *And A B* # *G*›
| *BetaNegOr*: ‹⊢ *Neg A* # *G* ⟹ ⊢ *Neg B* # *G* ⟹ ⊢ *Neg* (*Or A B*) # *G*›
| *BetaNegImpl*: ‹⊢ *A* # *G* ⟹ ⊢ *Neg B* # *G* ⟹ ⊢ *Neg* (*Impl A B*) # *G*›
| *GammaExists*: ‹⊢ *subst A t 0* # *G* ⟹ ⊢ *Exists A* # *G*›
| *GammaNegForall*: ‹⊢ *Neg* (*subst A t 0*) # *G* ⟹ ⊢ *Neg* (*Forall A*) # *G*›
| *DeltaForall*: ‹⊢ *subst A* (*App n* []) *0* # *G* ⟹ *news n* (*A* # *G*) ⟹ ⊢ *Forall A* # *G*›
| *DeltaNegExists*: ‹⊢ *Neg* (*subst A* (*App n* []) *0*) # *G* ⟹ *news n* (*A* # *G*) ⟹ ⊢ *Neg* (*Exists A*) # *G*›
| *Order*: ‹⊢ *G* ⟹ *set G* = *set G'* ⟹ ⊢ *G'*›

lemma *Shift*: ‹⊢ *rotate1 G* ⟹ ⊢ *G*›
  by (*simp add*: *Order*)

lemma *Swap*: ‹⊢ *B* # *A* # *G* ⟹ ⊢ *A* # *B* # *G*›
  by (*simp add*: *Order insert-commute*)

lemma ‹⊢ [*Neg* (*Pred* ''A'' []), *Pred* ''A'' []]›
  by (*rule Shift*, *simp*) (*rule Basic*)

lemma ‹⊢ [*And* (*Pred* ''A'' []) (*Pred* ''B'' []), *Neg* (*And* (*Pred* ''B'' []) (*Pred* ''A'' []))]›
  apply (*rule BetaAnd*)
   apply (*rule Swap*)
   apply (*rule AlphaNegAnd*)
   apply (*rule Shift*, *simp*, *rule Swap*)
   apply (*rule Basic*)
  apply (*rule Swap*)
  apply (*rule AlphaNegAnd*)
  apply (*rule Shift*, *rule Shift*, *simp*)
  apply (*rule Basic*)
  done

3.1 Soundness

lemma *SC-soundness*: ‹$\vdash$ $G \Longrightarrow \exists\, p \in$ *set* G. *eval e f g p*›
proof (*induct G arbitrary*: *f rule*: *SC.induct*)
  case (*DeltaForall A n G*)
  then consider
    ‹$\forall\, x$. *eval e* (f(n := λw. x)) g (*subst A* (*App n* []) *0*)› |
    ‹$\exists\, x$. $\exists\, p \in$ *set G*. *eval e* (f(n := λw. x)) *g p*›
    by *fastforce*
  then show *?case*
  proof *cases*
    case *1*
    then have ‹$\forall\, x$. *eval* (*shift e 0 x*) (f(n := λw. x)) *g A*›
      by *simp*
    then have ‹$\forall\, x$. *eval* (*shift e 0 x*) *f g A*›
      using ‹*news n* (*A* # *G*)› **by** *simp*
    then show *?thesis*
      by *simp*
  next
    case *2*
    then have ‹$\exists\, p \in$ *set G*. *eval e f g p*›
      using ‹*news n* (*A* # *G*)› **using** *Ball-set insert-iff list.set*(*2*) *upd-lemma* **by**
metis
    then show *?thesis*
      by *simp*
  qed
next
  case (*DeltaNegExists A n G*)
  then consider
    ‹$\forall\, x$. *eval e* (f(n := λw. x)) g (*Neg* (*subst A* (*App n* []) *0*))› |
    ‹$\exists\, x$. $\exists\, p \in$ *set G*. *eval e* (f(n := λw. x)) *g p*›
    by *fastforce*
  then show *?case*
  proof *cases*
    case *1*
    then have ‹$\forall\, x$. *eval* (*shift e 0 x*) (f(n := λw. x)) *g* (*Neg A*)›
      by *simp*
    then have ‹$\forall\, x$. *eval* (*shift e 0 x*) *f g* (*Neg A*)›
      using ‹*news n* (*A* # *G*)› **by** *simp*
    then show *?thesis*
      by *simp*
  next
    case *2*
    then have ‹$\exists\, p \in$ *set G*. *eval e f g p*›
      using ‹*news n* (*A* # *G*)› **using** *Ball-set insert-iff list.set*(*2*) *upd-lemma* **by**
metis
    then show *?thesis*
      by *simp*
  qed
qed *auto*

3.2 Tableau Calculus Equivalence

```
fun compl :: ‹('a, 'b) form ⇒ ('a, 'b) form› where
  ‹compl (Neg p) = p›
| ‹compl p = Neg p›

lemma compl: ‹compl p = Neg p ∨ (∃ q. compl p = q ∧ p = Neg q)›
  by (cases p rule: compl.cases) simp-all

lemma new-compl: ‹new n p ⟹ new n (compl p)›
  by (cases p rule: compl.cases) simp-all

lemma news-compl: ‹news n G ⟹ news n (map compl G)›
  using new-compl by (induct G) fastforce+

theorem TC-SC: ‹⊣ G ⟹ ⊢ map compl G›
proof (induct G rule: TC.induct)
  case (Basic i l G)
  then show ?case
    using SC.Basic Swap by fastforce
next
  case (AlphaNegNeg A G)
  then show ?case
    using SC.AlphaNegNeg compl by (metis compl.simps(1) list.simps(9))
next
  case (AlphaAnd A B G)
  then have *: ‹⊢ compl A # compl B # map compl G›
    by simp
  then have ‹⊢ Neg A # Neg B # map compl G›
    using compl AlphaNegNeg Swap by metis
  then show ?case
    using AlphaNegAnd by simp
next
  case (AlphaNegImpl A B G)
  then have ‹⊢ compl A # B # map compl G›
    by simp
  then have ‹⊢ Neg A # B # map compl G›
    using compl AlphaNegNeg by metis
  then show ?case
    using AlphaImpl by simp
next
  case (BetaOr A G B)
  then have ‹⊢ compl A # map compl G› ‹⊢ compl B # map compl G›
    by simp-all
  then have ‹⊢ Neg A # map compl G› ‹⊢ Neg B # map compl G›
    using compl AlphaNegNeg by metis+
  then show ?case
    using BetaNegOr by simp
next
  case (BetaImpl A G B)
```

```
  then have ‹⊢ A # map compl G› ‹⊢ compl B # map compl G›
    by simp-all
  then have ‹⊢ A # map compl G› ‹⊢ Neg B # map compl G›
    by − (assumption, metis compl AlphaNegNeg)
  then have ‹⊢ Neg (Impl A B) # map compl G›
    using BetaNegImpl by blast
  then have ‹⊢ compl (Impl A B) # map compl G›
    using ‹⊢ A # map compl G› compl by simp
  then show ?case
    by simp
next
  case (GammaForall A t G)
  then have ‹⊢ compl (subst A t 0) # map compl G›
    by simp
  then have ‹⊢ Neg (subst A t 0) # map compl G›
    using compl AlphaNegNeg by metis
  then show ?case
    using GammaNegForall by simp
next
  case (DeltaExists A n G)
  then have ‹⊢ compl (subst A (App n []) 0) # map compl G›
    by simp
  then have ‹⊢ Neg (subst A (App n []) 0) # map compl G›
    using compl AlphaNegNeg by metis
  moreover have ‹news n (A # map compl G)›
    using DeltaExists news-compl by fastforce
  ultimately show ?case
    using DeltaNegExists by simp
next
  case (DeltaNegForall A n G)
  then have ‹⊢ subst A (App n []) 0 # map compl G›
    by simp
  moreover have ‹news n (A # map compl G)›
    using DeltaNegForall news-compl by fastforce
  ultimately show ?case
    using DeltaForall by simp
qed (simp-all add: SC.intros)
```

3.3 Completeness

```
theorem SC-completeness:
  fixes p :: ‹(nat, nat) form›
  assumes ‹∀ (e :: nat ⇒ nat hterm) f g. list-all (eval e f g) ps ⟶ eval e f g p›
  shows ‹⊢ p # map compl ps›
proof −
  have ‹⊣ Neg p # ps›
    using assms tableau-completeness unfolding tableauproof-def by simp
  then show ?thesis
    using TC-SC by fastforce
```

qed

corollary
fixes *p* :: ‹(*nat*, *nat*) *form*›
assumes ‹∀ (*e* :: *nat* ⇒ *nat hterm*) *f g*. *eval e f g p*›
shows ‹⊢ [*p*]›
using *assms SC-completeness list.map*(*1*) **by** *metis*

end
theory *Sequent2* **imports** *Sequent* **begin**

4 Completeness Revisited

lemma ‹∃ *p*. *q* = *compl p*›
by (*metis compl.simps*(*1*))

definition *compl′* **where**
‹*compl′* = (λ*q*. (*SOME p*. *q* = *compl p*))›

lemma *comp′-sem*:
‹*eval e f g* (*compl′ p*) ⟷ ¬ *eval e f g p*›
by (*smt compl′-def compl.simps*(*1*) *compl eval.simps*(*7*) *someI-ex*)

lemma *comp′-sem-list*: ‹*list-ex* (λ*p*. ¬ *eval e f g p*) (*map compl′ ps*) ⟷ *list-ex* (*eval e f g*) *ps*›
by (*induct ps*) (*use comp′-sem* **in** *auto*)

theorem *SC-completeness′*:
fixes *ps* :: ‹(*nat*, *nat*) *form list*›
assumes ‹∀ (*e* :: *nat* ⇒ *nat hterm*) *f g*. *list-ex* (*eval e f g*) (*p* # *ps*)›
shows ‹⊢ *p* # *ps*›
proof −
define *ps′* **where** ‹*ps′* = *map compl′ ps*›
then have ‹*ps* = *map compl ps′*›
by (*induct ps arbitrary*: *ps′*) (*simp*, *smt* (*verit*) *compl′-def compl.simps*(*1*) *list.simps*(*9*) *someI-ex*)
from *assms* **have** ‹∀ (*e* :: *nat* ⇒ *nat hterm*) *f g*. (*list-ex* (*eval e f g*) *ps*) ∨ *eval e f g p*›
by *auto*
then have ‹∀ (*e* :: *nat* ⇒ *nat hterm*) *f g*. (*list-ex* (λ*p*. ¬ *eval e f g p*) *ps′*) ∨ *eval e f g p*›
unfolding *ps′-def* **using** *comp′-sem-list* **by** *blast*
then have ‹∀ (*e* :: *nat* ⇒ *nat hterm*) *f g*. *list-all* (*eval e f g*) *ps′* ⟶ *eval e f g p*›
by (*metis Ball-set Bex-set*)
then have ‹⊢ *p* # *map compl ps′*›
using *SC-completeness* **by** *blast*
then show *?thesis*
using ‹*ps* = *map compl ps′*› **by** *auto*
qed

corollary
fixes *ps* :: ‹(*nat*, *nat*) *form list*›
assumes ‹$\forall$ (*e* :: *nat* $\Rightarrow$ *nat hterm*) *f g*. *list-ex* (*eval e f g*) *ps*›
shows ‹$\vdash$ *ps*›
using *assms SC-completeness′* **by** (*cases ps*) *auto*

end

References

[1] M. Ben-Ari. *Mathematical Logic for Computer Science, 3rd Edition.* Springer, 2012.