```
(*

Isabelle/HOL theory file FOL.thy by Jørgen Villadsen https://people.compute.dtu.dk/jovi/

Introduction to Type Theory and Higher-Order Logic - Advanced Course in Logic and Computation
European Summer School in Logic, Language and Information (ESSLLI), 5-16 August 2019, Riga, Latvia

Snapshot: https://github.com/logic-tools/nadea Natural_Deduction_Assistant.thy (28 July 2019)

*)

section ‹Formalization of Natural Deduction and Sequent Calculus for First-Order Logic›

text ‹

Project: Natural_Deduction_Assistant (NaDeA) https://nadea.compute.dtu.dk/

Authors: Andreas Halkjær From, Anders Schlichtkrull & Jørgen Villadsen

License: https://www.isa-afp.org/LICENSE

Intertwined Development of Sequent Calculus: https://www.isa-afp.org/entries/FOL_Seq_Calc1.html

›

theory FOL imports Main begin
```

```
section ‹Natural Deduction›

type_synonym id = ‹char list›

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

datatype fm = Falsity | Pre id ‹tm list› | Imp fm fm | Dis fm fm | Con fm fm | Exi fm | Uni fm

primrec
  semantics_term :: ‹(nat ⇒ 'a) ⇒ (id ⇒ 'a list ⇒ 'a) ⇒ tm ⇒ 'a› and
  semantics_list :: ‹(nat ⇒ 'a) ⇒ (id ⇒ 'a list ⇒ 'a) ⇒ tm list ⇒ 'a 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 :: ‹(nat ⇒ 'a) ⇒ (id ⇒ 'a list ⇒ 'a) ⇒ (id ⇒ 'a list ⇒ bool) ⇒ fm ⇒ bool› where
  ‹semantics e f g Falsity = False› |
  ‹semantics e f g (Pre i l) = g i (semantics_list e f l)› |
  ‹semantics e f g (Imp p q) = (if semantics e f g p then semantics e f g q else True)› |
  ‹semantics e f g (Dis p q) = (if semantics e f g p then True else semantics e f g q)› |
  ‹semantics e f g (Con p q) = (if semantics e f g p then semantics e f g q else False)› |
  ‹semantics e f g (Exi p) = (∃x. semantics (λn. if n = 0 then x else e (n - 1)) f g p)› |
  ‹semantics e f g (Uni p) = (∀x. semantics (λn. if n = 0 then x else e (n - 1)) f g p)›

primrec member :: ‹fm ⇒ fm list ⇒ bool› where
```

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

primrec
  new_term :: ‹id ⇒ tm ⇒ bool› and
  new_list :: ‹id ⇒ tm list ⇒ bool› 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 :: ‹id ⇒ fm ⇒ bool› where
  ‹new c Falsity = True› |
  ‹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›

primrec news :: ‹id ⇒ fm list ⇒ bool› where
  ‹news c [] = True› |
  ‹news c (p # z) = (if new c p then news c z else False)›

primrec
  inc_term :: ‹tm ⇒ tm› and
  inc_list :: ‹tm list ⇒ tm 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 :: ‹nat ⇒ tm ⇒ tm ⇒ tm› and
  sub_list :: ‹nat ⇒ tm ⇒ tm list ⇒ tm 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 :: ‹nat ⇒ tm ⇒ fm ⇒ fm› where
  ‹sub v s Falsity = Falsity› |
  ‹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)›

inductive OK :: ‹fm ⇒ fm list ⇒ bool› where
  Assume: ‹member p z ⟹ OK p z› |
  Boole: ‹OK Falsity (Imp p Falsity # z) ⟹ OK p z› |
  Imp_E: ‹OK (Imp p q) z ⟹ OK p z ⟹ OK q z› |
  Imp_I: ‹OK q (p # z) ⟹ OK (Imp p q) z› |
```

```
  Dis_E: ‹OK (Dis p q) z ⟹ OK r (p # z) ⟹ OK r (q # z) ⟹ OK r z› |
  Dis_I1: ‹OK p z ⟹ OK (Dis p q) z› |
  Dis_I2: ‹OK q z ⟹ OK (Dis p q) z› |
  Con_E1: ‹OK (Con p q) z ⟹ OK p z› |
  Con_E2: ‹OK (Con p q) z ⟹ OK q z› |
  Con_I: ‹OK p z ⟹ OK q z ⟹ OK (Con p q) z› |
  Exi_E: ‹OK (Exi p) z ⟹ OK q (sub 0 (Fun c []) p # z) ⟹ news c (p # q # z) ⟹ OK q z› |
  Exi_I: ‹OK (sub 0 t p) z ⟹ OK (Exi p) z› |
  Uni_E: ‹OK (Uni p) z ⟹ OK (sub 0 t p) z› |
  Uni_I: ‹OK (sub 0 (Fun c []) p) z ⟹ news c (p # z) ⟹ OK (Uni p) z›

section ‹Examples›

lemma ‹OK (Imp (Pre "p" []) (Pre "p" [])) []›
  by (rule Imp_I, rule Assume) simp

lemma ‹OK (Imp (Pre "p" []) (Pre "p" [])) []›
proof -
  have ‹OK (Pre "p" []) [(Pre "p" [])]› by (rule Assume) simp
  then show ‹OK (Imp (Pre "p" []) (Pre "p" [])) []› by (rule Imp_I)
qed

lemma modus_tollens: ‹OK (Imp
  (Con (Imp (Pre "p" []) (Pre "q" [])) (Imp (Pre "q" []) Falsity))
  (Imp (Pre "p" []) Falsity)) []›
  apply (rule Imp_I)
  apply (rule Imp_I)
```

```
  apply (rule Imp_E)
   apply (rule Con_E2)
   apply (rule Assume)
   apply simp
  apply (rule Imp_E)
   apply (rule Con_E1)
   apply (rule Assume)
   apply simp
  apply (rule Assume)
  apply simp
  done

lemma Socrates_is_mortal: ‹OK (Imp
  (Con (Uni (Imp (Pre "h" [Var 0]) (Pre "m" [Var 0])))
      (Pre "h" [Fun "s" []]))
  (Pre "m" [Fun "s" []])) []›
  apply (rule Imp_I)
  apply (rule Imp_E [where p=‹Pre "h" [Fun "s" []]›])
   apply (subgoal_tac ‹OK (sub 0 (Fun "s" [])
     (Imp (Pre "h" [Var 0]) (Pre "m" [Var 0]))) _›)
    apply simp
   apply (rule Uni_E)
   apply (rule Con_E1)
   apply (rule Assume)
   apply simp
  apply (rule Con_E2)
  apply (rule Assume)
```

```
  apply simp
  done

lemma grandfather: ‹OK (Imp
  (Uni (Imp (Imp (Pre "r" [Var 0]) Falsity) (Pre "r" [Fun "f" [Var 0]])))
  (Exi (Con (Pre "r" [Var 0]) (Pre "r" [Fun "f" [Fun "f" [Var 0]]])))) []›
proof -
  let ?a = ‹Fun "a" []›
  let ?fa = ‹Fun "f" [?a]›
  let ?ffa = ‹Fun "f" [?fa]›
  let ?fffa = ‹Fun "f" [?ffa]›
  let ?ffffa = ‹Fun "f" [?fffa]›

  let ?ra = ‹Pre "r" [?a]›
  let ?rfa = ‹Pre "r" [?fa]›
  let ?rffa = ‹Pre "r" [?ffa]›
  let ?rfffa = ‹Pre "r" [?fffa]›
  let ?rffffa = ‹Pre "r" [?ffffa]›

  show ?thesis
    apply (rule Boole)
    apply (rule Imp_E)
     apply (rule Assume)
     apply simp
    apply (rule Imp_I)
    apply (rule Imp_E [where p=‹Imp (Imp ?ra Falsity) ?rfa›])
     apply (rule Imp_I)
```

```
apply (rule Imp_E [where p=‹(Imp (Imp ?rfa Falsity) ?rffa)›])
 apply (rule Imp_I)
 apply (rule Imp_E [where p=‹Imp (Imp ?rffa Falsity) ?rfffa›])
  apply (rule Imp_I)
  apply (rule Imp_E [where p=‹Imp (Imp ?rfffa Falsity) ?rffffa›])
   apply (rule Imp_I)
   apply (rule Dis_E [where p=‹?ra› and q=‹Imp ?ra Falsity›])
     apply (rule Boole)
     apply (rule Imp_E [where p=‹Dis ?ra (Imp ?ra Falsity)›])
      apply (rule Assume)
      apply simp
     apply (rule Dis_I2)
     apply (rule Imp_I)
     apply (rule Imp_E [where p=‹Dis ?ra (Imp ?ra Falsity)›])
      apply (rule Assume)
      apply simp
     apply (rule Dis_I1)
     apply (rule Assume)
     apply simp
    apply (rule Dis_E [where p=‹?rffa› and q=‹Imp ?rffa Falsity›])
      apply (rule Boole)
      apply (rule Imp_E [where p=‹Dis ?rffa (Imp ?rffa Falsity)›])
       apply (rule Assume)
       apply simp
      apply (rule Dis_I2)
      apply (rule Imp_I)
      apply (rule Imp_E [where p=‹Dis ?rffa (Imp ?rffa Falsity)›])
```

```
   apply (rule Assume)
   apply simp
  apply (rule Dis_I1)
  apply (rule Assume)
  apply simp
 apply (rule Exi_I [where t=‹?a›])
 apply simp
 apply (rule Con_I)
  apply (rule Assume)
  apply simp
 apply (rule Assume)
 apply simp
apply (rule Imp_E [where p=‹Imp (Imp ?rffa Falsity) ?rfa›])
 apply (rule Imp_I)
 apply (rule Exi_I [where t=‹?fa›])
 apply simp
 apply (rule Con_I)
  apply (rule Imp_E [where p=‹Imp ?rffa Falsity›])
   apply (rule Assume)
   apply simp
  apply (rule Assume)
  apply simp
 apply (rule Imp_E [where p=‹Imp ?rffa Falsity›])
  apply (rule Assume)
  apply simp
 apply (rule Assume)
 apply simp
```

```
 apply (rule Imp_E [where p=‹Imp (Imp ?rfa Falsity) ?rffa›])
  apply (rule Imp_I)
  apply (rule Imp_I)
  apply (rule Boole)
  apply (rule Imp_E [where p=‹?rffa›])
   apply (rule Assume)
   apply simp
  apply (rule Imp_E [where p=‹Imp ?rfa Falsity›])
   apply (rule Assume)
   apply simp
  apply (rule Assume)
  apply simp
 apply (rule Assume)
 apply simp
apply (rule Dis_E [where p=‹?rfffa› and q=‹Imp ?rfffa Falsity›])
  apply (rule Boole)
  apply (rule Imp_E [where p=‹Dis ?rfffa (Imp ?rfffa Falsity)›])
   apply (rule Assume)
   apply simp
  apply (rule Dis_I2)
  apply (rule Imp_I)
  apply (rule Imp_E [where p=‹Dis ?rfffa (Imp ?rfffa Falsity)›])
   apply (rule Assume)
   apply simp
  apply (rule Dis_I1)
  apply (rule Assume)
  apply simp
```

```
 apply (rule Exi_I [where t=‹?fa›])
 apply simp
 apply (rule Con_I)
  apply (rule Imp_E [where p=‹Imp ?ra Falsity›])
   apply (rule Assume)
   apply simp
  apply (rule Assume)
  apply simp
 apply (rule Assume)
 apply simp
apply (rule Imp_E [where p=‹Imp (Imp ?rfffa Falsity) ?rffa›])
 apply (rule Imp_I)
 apply (rule Exi_I [where t=‹?ffa›])
 apply simp
 apply (rule Con_I)
  apply (rule Imp_E [where p=‹Imp ?rfffa Falsity›])
   apply (rule Assume)
   apply simp
  apply (rule Assume)
  apply simp
 apply (rule Imp_E [where p=‹Imp ?rfffa Falsity›])
  apply (rule Assume)
  apply simp
 apply (rule Assume)
 apply simp
apply (rule Imp_E [where p=‹Imp (Imp ?rffa Falsity) ?rfffa›])
 apply (rule Imp_I)
```

```
   apply (rule Imp_I)
   apply (rule Boole)
   apply (rule Imp_E [where p=‹?rfffa›])
    apply (rule Assume)
    apply simp
   apply (rule Imp_E [where p=‹Imp ?rffa Falsity›])
    apply (rule Assume)
    apply simp
   apply (rule Assume)
   apply simp
  apply (rule Assume)
  apply simp
 apply (subgoal_tac ‹OK (sub 0 ?fffa
  (Imp (Imp (Pre "r" [Var 0]) Falsity) (Pre "r" [Fun "f" [Var 0]]))) _›)
  apply simp
 apply (rule Uni_E)
 apply (rule Assume)
 apply simp
apply (subgoal_tac ‹OK (sub 0 ?ffa
  (Imp (Imp (Pre "r" [Var 0]) Falsity) (Pre "r" [Fun "f" [Var 0]]))) _›)
 apply simp
apply (rule Uni_E)
apply (rule Assume)
apply simp
apply (subgoal_tac ‹OK (sub 0 ?fa
(Imp (Imp (Pre "r" [Var 0]) Falsity) (Pre "r" [Fun "f" [Var 0]]))) _›)
apply simp
```

```
    apply (rule Uni_E)
    apply (rule Assume)
    apply simp
   apply (subgoal_tac ‹OK (sub 0 ?a
     (Imp (Imp (Pre "r" [Var 0]) Falsity) (Pre "r" [Fun "f" [Var 0]]))) _›)
    apply simp
   apply (rule Uni_E)
   apply (rule Assume)
   apply simp
   done
qed

lemma open_example: ‹OK (Dis (Pre "p" [Var x]) (Imp Falsity Falsity)) []›
  apply (rule Dis_I2)
  apply (rule Imp_I)
  apply (rule Assume)
  apply simp
  done

section ‹Soundness›

lemma symbols [simp]:
  ‹(if p then q else True) = (p ⟶ q)›
  ‹(if p then True else q) = (p ∨ q)›
  ‹(if p then q else False) = (p ∧ q)›
  by simp_all
```

```
fun put :: ‹(nat ⇒ 'a) ⇒ nat ⇒ 'a ⇒ nat ⇒ 'a› where
  ‹put e v x = (λn. if n < v then e n else if n = v then x else e (n - 1))›

lemma ‹put e 0 x = (λn. if n = 0 then x else e (n - 1))›
  by simp

lemma
  ‹semantics e f g (Exi p) = (∃x. semantics (put e 0 x) f g p)›
  ‹semantics e f g (Uni p) = (∀x. semantics (put e 0 x) f g p)›
  by simp_all

lemma increment:
  ‹semantics_term (put e 0 x) f (inc_term t) = semantics_term e f t›
  ‹semantics_list (put e 0 x) f (inc_list l) = semantics_list e f l›
  by (induct t and l rule: semantics_term.induct semantics_list.induct) simp_all

lemma commute: ‹put (put e v x) 0 y = put (put e 0 y) (v + 1) x›
  by fastforce

lemma allnew [simp]: ‹list_all (new c) z = news c z›
  by (induct z) simp_all

lemma map' [simp]:
  ‹new_term n t ⟹ semantics_term e (f(n := x)) t = semantics_term e f t›
  ‹new_list n l ⟹ semantics_list e (f(n := x)) l = semantics_list e f l›
  by (induct t and l rule: semantics_term.induct semantics_list.induct) auto
```

```
lemma map [simp]: ‹new n p ⟹ semantics e (f(n := x)) g p = semantics e f g p›
  by (induct p arbitrary: e) simp_all

lemma allmap [simp]: ‹news c z ⟹
    list_all (semantics e (f(c := m)) g) z = list_all (semantics e f g) z›
  by (induct z) simp_all

lemma substitute' [simp]:
  ‹semantics_term e f (sub_term v s t) = semantics_term (put e v (semantics_term e f s)) f t›
  ‹semantics_list e f (sub_list v s l) = semantics_list (put e v (semantics_term e f s)) f l›
  by (induct t and l rule: semantics_term.induct semantics_list.induct) simp_all

lemma substitute [simp]:
  ‹semantics e f g (sub v t p) = semantics (put e v (semantics_term e f t)) f g p›
proof (induct p arbitrary: e v t)
  case (Exi p)
  have ‹semantics e f g (sub v t (Exi p)) =
      (∃x. semantics (put e 0 x) f g (sub (v + 1) (inc_term t) p))›
    by simp
  also have ‹… = (∃x. semantics (put (put e 0 x) (v + 1)
      (semantics_term (put e 0 x) f (inc_term t))) f g p)›
    using Exi by simp
  also have ‹… = (∃x. semantics (put (put e v (semantics_term e f t)) 0 x) f g p)›
    using commute increment(1) by metis
  finally show ?case
    by simp
next
```

```
  case (Uni p)
  have ‹semantics e f g (sub v t (Uni p)) =
      (∀x. semantics (put e 0 x) f g (sub (v + 1) (inc_term t) p))›
    by simp
  also have ‹… =
      (∀x. semantics (put (put e 0 x) (v + 1) (semantics_term (put e 0 x) f (inc_term t))) f g p)›
    using Uni by simp
  also have ‹… = (∀x. semantics (put (put e v (semantics_term e f t)) 0 x) f g p)›
    using commute increment(1) by metis
  finally show ?case
    by simp
qed simp_all

lemma member_set [simp]: ‹p ∈ set z = member p z›
  by (induct z) simp_all

lemma soundness': ‹OK p z ⟹ list_all (semantics e f g) z ⟹ semantics e f g p›
proof (induct p z arbitrary: f rule: OK.induct)
  case (Exi_E p z q c)
  then obtain x where ‹semantics (put e 0 x) f g p›
    by auto
  then have ‹semantics (put e 0 x) (f(c := λw. x)) g p›
    using ‹news c (p # q # z)› by simp
  then have ‹semantics e (f(c := λw. x)) g (sub 0 (Fun c []) p)›
    by simp
  then have ‹list_all (semantics e (f(c := λw. x)) g) (sub 0 (Fun c []) p # z)›
    using Exi_E by simp
```

```
  then have ‹semantics e (f(c := λw. x)) g q›
    using Exi_E by blast
  then show ‹semantics e f g q›
    using ‹news c (p # q # z)› by simp
next
  case (Uni_I c p z)
  then have ‹∀x. list_all (semantics e (f(c := λw. x)) g) z›
    by simp
  then have ‹∀x. semantics e (f(c := λw. x)) g (sub 0 (Fun c []) p)›
    using Uni_I by blast
  then have ‹∀x. semantics (put e 0 x) (f(c := λw. x)) g p›
    by simp
  then have ‹∀x. semantics (put e 0 x) f g p›
    using ‹news c (p # z)› by simp
  then show ‹semantics e f g (Uni p)›
    by simp
qed (auto simp: list_all_iff)

theorem soundness: ‹OK p [] ⟹ semantics e f g p›
  by (simp add: soundness')

corollary ‹∃p. OK p []› ‹∃p. ¬ OK p []›
proof -
  have ‹OK (Imp p p) []› for p
    by (rule Imp_I, rule Assume, simp)
  then show ‹∃p. OK p []›
    by iprover
```

```
next
  have ‹¬ semantics (e :: _ ⇒ unit) f g Falsity› for e f g
    by simp
  then show ‹∃p. ¬ OK p []›
    using soundness by iprover
qed

section ‹Utilities›

lemma set_inter_compl_diff: ‹- A ∩ B = B - A› unfolding Diff_eq using inf_commute .

abbreviation Neg :: ‹fm ⇒ fm› where ‹Neg p ≡ Imp p Falsity›

abbreviation Truth :: ‹fm› where ‹Truth ≡ Neg Falsity›

primrec size_formulas :: ‹fm ⇒ nat› where
  ‹size_formulas Falsity = 0› |
  ‹size_formulas (Pre _ _) = 0› |
  ‹size_formulas (Con p q) = size_formulas p + size_formulas q + 1› |
  ‹size_formulas (Dis p q) = size_formulas p + size_formulas q + 1› |
  ‹size_formulas (Imp p q) = size_formulas p + size_formulas q + 1› |
  ‹size_formulas (Uni p) = size_formulas p + 1› |
  ‹size_formulas (Exi p) = size_formulas p + 1›

lemma sub_size_formulas [simp]: ‹size_formulas (sub i t p) = size_formulas p›
  by (induct p arbitrary: i t) simp_all
```

```
subsection ‹Extra Rules›

lemma explosion: ‹OK (Imp Falsity p) z›
  apply (rule Imp_I) apply (rule Boole) apply (rule Assume) by simp

lemma cut: ‹OK p z ⟹ OK q (p # z) ⟹ OK q z›
  apply (rule Imp_E) apply (rule Imp_I) .

lemma Falsity_E: ‹OK Falsity z ⟹ OK p z›
  apply (rule Imp_E) apply (rule explosion) .

lemma Boole': ‹OK p (Neg p # z) ⟹ OK p z›
  apply (rule Boole) apply (rule Imp_E) apply (rule Assume) by simp iprover

subsection ‹Closed Formulas›

primrec
  closed_term :: ‹nat ⇒ tm ⇒ bool› and
  closed_list :: ‹nat ⇒ tm list ⇒ bool› where
  ‹closed_term m (Var n) = (n < m)› |
  ‹closed_term m (Fun a ts) = closed_list m ts› |
  ‹closed_list m [] = True› |
  ‹closed_list m (t # ts) = (closed_term m t ∧ closed_list m ts)›

primrec closed :: ‹nat ⇒ fm ⇒ bool› where
  ‹closed m Falsity = True› |
  ‹closed m (Pre b ts) = closed_list m ts› |
```

```
  ‹closed m (Con p q) = (closed m p ∧ closed m q)› |
  ‹closed m (Dis p q) = (closed m p ∧ closed m q)› |
  ‹closed m (Imp p q) = (closed m p ∧ closed m q)› |
  ‹closed m (Uni p) = closed (Suc m) p› |
  ‹closed m (Exi p) = closed (Suc m) p›

lemma closed_mono':
  assumes ‹i ≤ j›
  shows ‹closed_term i t ⟹ closed_term j t›
    and ‹closed_list i l ⟹ closed_list j l›
  using assms by (induct t and l rule: closed_term.induct closed_list.induct) simp_all

lemma closed_mono: ‹i ≤ j ⟹ closed i p ⟹ closed j p›
proof (induct p arbitrary: i j)
  case (Pre i l)
  then show ?case
    using closed_mono' by simp
next
  case (Exi p)
  then have ‹closed (Suc i) p›
    by simp
  then have ‹closed (Suc j) p›
    using Exi by blast
  then show ?case
    by simp
next
  case (Uni p)
```

```
  then have ‹closed (Suc i) p›
    by simp
  then have ‹closed (Suc j) p›
    using Uni by blast
  then show ?case
    by simp
qed simp_all

lemma inc_closed [simp]:
  ‹closed_term 0 t ⟹ closed_term 0 (inc_term t)›
  ‹closed_list 0 l ⟹ closed_list 0 (inc_list l)›
  by (induct t and l rule: closed_term.induct closed_list.induct) simp_all

lemma sub_closed' [simp]:
  assumes ‹closed_term 0 u›
  shows ‹closed_term (Suc i) t ⟹ closed_term i (sub_term i u t)›
    and ‹closed_list (Suc i) l ⟹ closed_list i (sub_list i u l)›
  using assms
proof (induct t and l rule: closed_term.induct closed_list.induct)
  case (Var x)
  then show ?case
    using closed_mono'(1) by auto
qed simp_all

lemma sub_closed [simp]: ‹closed_term 0 t ⟹ closed (Suc i) p ⟹ closed i (sub i t p)›
  by (induct p arbitrary: i t) simp_all
```

```
subsection ‹Parameters›

primrec
  params_term :: ‹tm ⇒ id set› and
  params_list :: ‹tm list ⇒ id set› where
  ‹params_term (Var n) = {}› |
  ‹params_term (Fun a ts) = {a} ∪ params_list ts› |
  ‹params_list [] = {}› |
  ‹params_list (t # ts) = (params_term t ∪ params_list ts)›

primrec params :: ‹fm ⇒ id set› where
  ‹params Falsity = {}› |
  ‹params (Pre b ts) = params_list ts› |
  ‹params (Con p q) = params p ∪ params q› |
  ‹params (Dis p q) = params p ∪ params q› |
  ‹params (Imp p q) = params p ∪ params q› |
  ‹params (Uni p) = params p› |
  ‹params (Exi p) = params p›

lemma new_params' [simp]:
  ‹new_term c t = (c ∉ params_term t)›
  ‹new_list c l = (c ∉ params_list l)›
  by (induct t and l rule: new_term.induct new_list.induct) simp_all

lemma new_params [simp]: ‹new x p = (x ∉ params p)›
  by (induct p) simp_all
```

```
lemma news_params [simp]: ‹news c z = list_all (λp. c ∉ params p) z›
  by (induct z) simp_all

lemma params_inc [simp]:
  ‹params_term (inc_term t) = params_term t›
  ‹params_list (inc_list l) = params_list l›
  by (induct t and l rule: sub_term.induct sub_list.induct) simp_all

primrec
  psubst_term :: ‹(id ⇒ id) ⇒ tm ⇒ tm› and
  psubst_list :: ‹(id ⇒ id) ⇒ tm list ⇒ tm list› where
  ‹psubst_term f (Var i) = Var i› |
  ‹psubst_term f (Fun x ts) = Fun (f x) (psubst_list f ts)› |
  ‹psubst_list f [] = []› |
  ‹psubst_list f (t # ts) = psubst_term f t # psubst_list f ts›

primrec psubst :: ‹(id ⇒ id) ⇒ fm ⇒ fm› where
  ‹psubst f Falsity = Falsity› |
  ‹psubst f (Pre b ts) = Pre b (psubst_list f ts)› |
  ‹psubst f (Con p q) = Con (psubst f p) (psubst f q)› |
  ‹psubst f (Dis p q) = Dis (psubst f p) (psubst f q)› |
  ‹psubst f (Imp p q) = Imp (psubst f p) (psubst f q)› |
  ‹psubst f (Uni p) = Uni (psubst f p)› |
  ‹psubst f (Exi p) = Exi (psubst f p)›

lemma psubst_closed' [simp]:
  ‹closed_term i (psubst_term f t) = closed_term i t›
```

```
  ‹closed_list i (psubst_list f l) = closed_list i l›
  by (induct t and l rule: closed_term.induct closed_list.induct) simp_all

lemma psubst_closed [simp]: ‹closed i (psubst f p) = closed i p›
  by (induct p arbitrary: i) simp_all

lemma psubst_inc [simp]:
  ‹psubst_term f (inc_term t) = inc_term (psubst_term f t)›
  ‹psubst_list f (inc_list l) = inc_list (psubst_list f l)›
  by (induct t and l rule: psubst_term.induct psubst_list.induct) simp_all

lemma psubst_sub' [simp]:
  ‹psubst_term f (sub_term i u t) = sub_term i (psubst_term f u) (psubst_term f t)›
  ‹psubst_list f (sub_list i u l) = sub_list i (psubst_term f u) (psubst_list f l)›
  by (induct t and l rule: psubst_term.induct psubst_list.induct) simp_all

lemma psubst_sub [simp]: ‹psubst f (sub i t P) = sub i (psubst_term f t) (psubst f P)›
  by (induct P arbitrary: i t) simp_all

lemma psubst_upd' [simp]:
  ‹x ∉ params_term t ⟹ psubst_term (f(x := y)) t = psubst_term f t›
  ‹x ∉ params_list l ⟹ psubst_list (f(x := y)) l = psubst_list f l›
  by (induct t and l rule: psubst_term.induct psubst_list.induct) auto

lemma psubst_upd [simp]: ‹x ∉ params P ⟹ psubst (f(x := y)) P = psubst f P›
  by (induct P) simp_all
```

```
lemma psubst_id': ‹psubst_term id t = t› ‹psubst_list (λx. x) l = l›
  by (induct t and l rule: psubst_term.induct psubst_list.induct) simp_all

lemma psubst_id [simp]: ‹psubst id = id›
proof
  fix p
  show ‹psubst id p = id p›
    by (induct p) (simp_all add: psubst_id')
qed

lemma psubst_image' [simp]:
  ‹params_term (psubst_term f t) = f ` params_term t›
  ‹params_list (psubst_list f l) = f ` params_list l›
  by (induct t and l rule: params_term.induct params_list.induct) (simp_all add: image_Un)

lemma psubst_image [simp]: ‹params (psubst f p) = f ` params p›
  by (induct p) (simp_all add: image_Un)

lemma psubst_semantics' [simp]:
  ‹semantics_term e f (psubst_term h t) = semantics_term e (λx. f (h x)) t›
  ‹semantics_list e f (psubst_list h l) = semantics_list e (λx. f (h x)) l›
  by (induct t and l rule: semantics_term.induct semantics_list.induct) simp_all

lemma psubst_semantics: ‹semantics e f g (psubst h p) = semantics e (λx. f (h x)) g p›
  by (induct p arbitrary: e) simp_all

section ‹Completeness for Closed Formulas›
```

## subsection ‹Consistent Sets›

```
definition consistency :: ‹fm set set ⇒ bool› where
 ‹consistency C = (∀S. S ∈ C ⟶
   (∀p ts. ¬ (Pre p ts ∈ S ∧ Neg (Pre p ts) ∈ S)) ∧
   Falsity ∉ S ∧
   (∀p q. Con p q ∈ S ⟶ S ∪ {p, q} ∈ C) ∧
   (∀p q. Neg (Dis p q) ∈ S ⟶ S ∪ {Neg p, Neg q} ∈ C) ∧
   (∀p q. Dis p q ∈ S ⟶ S ∪ {p} ∈ C ∨ S ∪ {q} ∈ C) ∧
   (∀p q. Neg (Con p q) ∈ S ⟶ S ∪ {Neg p} ∈ C ∨ S ∪ {Neg q} ∈ C) ∧
   (∀p q. Imp p q ∈ S ⟶ S ∪ {Neg p} ∈ C ∨ S ∪ {q} ∈ C) ∧
   (∀p q. Neg (Imp p q) ∈ S ⟶ S ∪ {p, Neg q} ∈ C) ∧
   (∀P t. closed_term 0 t ⟶ Uni P ∈ S ⟶ S ∪ {sub 0 t P} ∈ C) ∧
   (∀P t. closed_term 0 t ⟶ Neg (Exi P) ∈ S ⟶ S ∪ {Neg (sub 0 t P)} ∈ C) ∧
   (∀P. Exi P ∈ S ⟶ (∃x. S ∪ {sub 0 (Fun x []) P} ∈ C)) ∧
   (∀P. Neg (Uni P) ∈ S ⟶ (∃x. S ∪ {Neg (sub 0 (Fun x []) P)} ∈ C)))›

definition alt_consistency :: ‹fm set set ⇒ bool› where
 ‹alt_consistency C = (∀S. S ∈ C ⟶
    (∀p ts. ¬ (Pre p ts ∈ S ∧ Neg (Pre p ts) ∈ S)) ∧
    Falsity ∉ S ∧
    (∀p q. Con p q ∈ S ⟶ S ∪ {p, q} ∈ C) ∧
    (∀p q. Neg (Dis p q) ∈ S ⟶ S ∪ {Neg p, Neg q} ∈ C) ∧
    (∀p q. Dis p q ∈ S ⟶ S ∪ {p} ∈ C ∨ S ∪ {q} ∈ C) ∧
    (∀p q. Neg (Con p q) ∈ S ⟶ S ∪ {Neg p} ∈ C ∨ S ∪ {Neg q} ∈ C) ∧
```

```
    (∀p q. Imp p q ∈ S ⟶ S ∪ {Neg p} ∈ C ∨ S ∪ {q} ∈ C) ∧
    (∀p q. Neg (Imp p q) ∈ S ⟶ S ∪ {p, Neg q} ∈ C) ∧
    (∀P t. closed_term 0 t ⟶ Uni P ∈ S ⟶ S ∪ {sub 0 t P} ∈ C) ∧
    (∀P t. closed_term 0 t ⟶ Neg (Exi P) ∈ S ⟶ S ∪ {Neg (sub 0 t P)} ∈ C) ∧
    (∀P x. (∀a ∈ S. x ∉ params a) ⟶ Exi P ∈ S ⟶ S ∪ {sub 0 (Fun x []) P} ∈ C) ∧
    (∀P x. (∀a ∈ S. x ∉ params a) ⟶ Neg (Uni P) ∈ S ⟶ S ∪ {Neg (sub 0 (Fun x []) P)} ∈ C))›

definition mk_alt_consistency :: ‹fm set set ⇒ fm set set› where
 ‹mk_alt_consistency C = {S. ∃f. psubst f ` S ∈ C}›

theorem alt_consistency:
 assumes conc: ‹consistency C›
 shows ‹alt_consistency (mk_alt_consistency C)› (is ‹alt_consistency ?C'›)
 unfolding alt_consistency_def
proof (intro allI impI conjI)
 fix S'

 assume ‹S' ∈ ?C'›
 then obtain f where sc: ‹psubst f ` S' ∈ C› (is ‹?S ∈ C›)
  unfolding mk_alt_consistency_def by blast

 fix p ts
 show ‹¬ (Pre p ts ∈ S' ∧ Neg (Pre p ts) ∈ S')›
 proof
  assume *: ‹Pre p ts ∈ S' ∧ Neg (Pre p ts) ∈ S'›
  then have ‹psubst f (Pre p ts) ∈ ?S›
   by blast
```

```
  then have ‹Pre p (psubst_list f ts) ∈ ?S›
    by simp
  then have ‹Neg (Pre p (psubst_list f ts)) ∉ ?S›
    using conc sc by (simp add: consistency_def)
  then have ‹Neg (Pre p ts) ∉ S'›
    by force
  then show False
    using * by blast
qed

have ‹Falsity ∉ ?S›
  using conc sc unfolding consistency_def by simp
then show ‹Falsity ∉ S'›
  by force

{ fix p q
  assume ‹Con p q ∈ S'›
  then have ‹psubst f (Con p q) ∈ ?S›
    by blast
  then have ‹?S ∪ {psubst f p, psubst f q} ∈ C›
    using conc sc by (simp add: consistency_def)
  then show ‹S' ∪ {p, q} ∈ ?C'›
    unfolding mk_alt_consistency_def by auto }

{ fix p q
  assume ‹Neg (Dis p q) ∈ S'›
  then have ‹psubst f (Neg (Dis p q)) ∈ ?S›
```

```
    by blast
  then have ‹?S ∪ {Neg (psubst f p), Neg (psubst f q)} ∈ C›
    using conc sc by (simp add: consistency_def)
  then show ‹S' ∪ {Neg p, Neg q} ∈ ?C'›
    unfolding mk_alt_consistency_def by auto }

{ fix p q
  assume ‹Neg (Imp p q) ∈ S'›
  then have ‹psubst f (Neg (Imp p q)) ∈ ?S›
    by blast
  then have ‹?S ∪ {psubst f p, Neg (psubst f q)} ∈ C›
    using conc sc by (simp add: consistency_def)
  then show ‹S' ∪ {p, Neg q} ∈ ?C'›
    unfolding mk_alt_consistency_def by auto }

{ fix p q
  assume ‹Dis p q ∈ S'›
  then have ‹psubst f (Dis p q) ∈ ?S›
    by blast
  then have ‹?S ∪ {psubst f p} ∈ C ∨ ?S ∪ {psubst f q} ∈ C›
    using conc sc by (simp add: consistency_def)
  then show ‹S' ∪ {p} ∈ ?C' ∨ S' ∪ {q} ∈ ?C'›
    unfolding mk_alt_consistency_def by auto }

{ fix p q
  assume ‹Neg (Con p q) ∈ S'›
  then have ‹psubst f (Neg (Con p q)) ∈ ?S›
```

```
  by blast
 then have ‹?S ∪ {Neg (psubst f p)} ∈ C ∨ ?S ∪ {Neg (psubst f q)} ∈ C›
  using conc sc by (simp add: consistency_def)
 then show ‹S' ∪ {Neg p} ∈ ?C' ∨ S' ∪ {Neg q} ∈ ?C'›
  unfolding mk_alt_consistency_def by auto }

{ fix p q
 assume ‹Imp p q ∈ S'›
 then have ‹psubst f (Imp p q) ∈ ?S›
  by blast
 then have ‹?S ∪ {Neg (psubst f p)} ∈ C ∨ ?S ∪ {psubst f q} ∈ C›
  using conc sc by (simp add: consistency_def)
 then show ‹S' ∪ {Neg p} ∈ ?C' ∨ S' ∪ {q} ∈ ?C'›
  unfolding mk_alt_consistency_def by auto }

{ fix P t
 assume ‹closed_term 0 t› and ‹Uni P ∈ S'›
 then have ‹psubst f (Uni P) ∈ ?S›
  by blast
 then have ‹?S ∪ {sub 0 (psubst_term f t) (psubst f P)} ∈ C›
  using ‹closed_term 0 t› conc sc by (simp add: consistency_def)
 then show ‹S' ∪ {sub 0 t P} ∈ ?C'›
  unfolding mk_alt_consistency_def by auto }

{ fix P t
 assume ‹closed_term 0 t› and ‹Neg (Exi P) ∈ S'›
 then have ‹psubst f (Neg (Exi P)) ∈ ?S›
```

```
    by blast
  then have ‹?S ∪ {Neg (sub 0 (psubst_term f t) (psubst f P))} ∈ C›
    using ‹closed_term 0 t› conc sc by (simp add: consistency_def)
  then show ‹S' ∪ {Neg (sub 0 t P)} ∈ ?C'›
    unfolding mk_alt_consistency_def by auto }

{ fix P x
  assume ‹∀a ∈ S'. x ∉ params a› and ‹Exi P ∈ S'›
  moreover have ‹psubst f (Exi P) ∈ ?S›
    using calculation by blast
  then have ‹∃y. ?S ∪ {sub 0 (Fun y []) (psubst f P)} ∈ C›
    using conc sc by (simp add: consistency_def)
  then obtain y where ‹?S ∪ {sub 0 (Fun y []) (psubst f P)} ∈ C›
    by blast

  moreover have ‹psubst (f(x := y)) ` S' = ?S›
    using calculation by (simp cong add: image_cong)
  then have ‹psubst (f(x := y)) `
      S' ∪ {sub 0 (Fun ((f(x := y)) x) []) (psubst (f(x := y)) P)} ∈ C›
    using calculation by auto
  then have ‹∃f. psubst f `
      S' ∪ {sub 0 (Fun (f x) []) (psubst f P)} ∈ C›
    by blast
  then show ‹S' ∪ {sub 0 (Fun x []) P} ∈ ?C'›
    unfolding mk_alt_consistency_def by simp }

{ fix P x
```

```
    assume ‹∀a ∈ S'. x ∉ params a› and ‹Neg (Uni P) ∈ S'›
    moreover have ‹psubst f (Neg (Uni P)) ∈ ?S›
      using calculation by blast
    then have ‹∃y. ?S ∪ {Neg (sub 0 (Fun y []) (psubst f P))} ∈ C›
      using conc sc by (simp add: consistency_def)
    then obtain y where ‹?S ∪ {Neg (sub 0 (Fun y []) (psubst f P))} ∈ C›
      by blast

    moreover have ‹psubst (f(x := y)) ` S' = ?S›
      using calculation by (simp cong add: image_cong)
    moreover have ‹psubst (f(x := y)) `
    S' ∪ {Neg (sub 0 (Fun ((f(x := y)) x) []) (psubst (f(x := y)) P))} ∈ C›
      using calculation by auto
    ultimately have ‹∃f. psubst f ` S' ∪ {Neg (sub 0 (Fun (f x) []) (psubst f P))} ∈ C›
      by blast
    then show ‹S' ∪ {Neg (sub 0 (Fun x []) P)} ∈ ?C'›
      unfolding mk_alt_consistency_def by simp }
qed

theorem mk_alt_consistency_subset: ‹C ⊆ mk_alt_consistency C›
  unfolding mk_alt_consistency_def
proof
  fix S
  assume ‹S ∈ C›
  then have ‹psubst id ` S ∈ C›
    by simp
  then have ‹∃f. psubst f ` S ∈ C›
```

```
    by blast
  then show ‹S ∈ {S. ∃f. psubst f ` S ∈ C}›
    by simp
qed

subsection ‹Closure under Subsets›

definition close :: ‹fm set set ⇒ fm set set› where
  ‹close C = {S. ∃S' ∈ C. S ⊆ S'}›

definition subset_closed :: ‹'a set set ⇒ bool› where
  ‹subset_closed C = (∀S' ∈ C. ∀S. S ⊆ S' ⟶ S ∈ C)›

lemma subset_in_close:
  assumes ‹S' ⊆ S› and ‹S ∪ x ∈ C›
  shows ‹S' ∪ x ∈ close C›
proof -
  have ‹S ∪ x ∈ close C›
    unfolding close_def using ‹S ∪ x ∈ C› by blast
  then show ?thesis
    unfolding close_def using ‹S' ⊆ S› by blast
qed

theorem close_consistency:
  assumes conc: ‹consistency C›
  shows ‹consistency (close C)›
  unfolding consistency_def
```

```
proof (intro allI impI conjI)
  fix S'
  assume ‹S' ∈ close C›
  then obtain S where ‹S ∈ C› and ‹S' ⊆ S›
    unfolding close_def by blast

  { fix p ts
    have ‹¬ (Pre p ts ∈ S ∧ Neg (Pre p ts) ∈ S)›
      using ‹S ∈ C› conc unfolding consistency_def by simp
    then show ‹¬ (Pre p ts ∈ S' ∧ Neg (Pre p ts) ∈ S')›
      using ‹S' ⊆ S› by blast }

  { have ‹Falsity ∉ S›
      using ‹S ∈ C› conc unfolding consistency_def by blast
    then show ‹Falsity ∉ S'›
      using ‹S' ⊆ S› by blast }

  { fix p q
    assume ‹Con p q ∈ S'›
    then have ‹Con p q ∈ S›
      using ‹S' ⊆ S› by blast
    then have ‹S ∪ {p, q} ∈ C›
      using ‹S ∈ C› conc unfolding consistency_def by simp
    then show ‹S' ∪ {p, q} ∈ close C›
      using ‹S' ⊆ S› subset_in_close by blast }

  { fix p q
```

```
  assume ‹Neg (Dis p q) ∈ S'›
  then have ‹Neg (Dis p q) ∈ S›
   using ‹S' ⊆ S› by blast
  then have ‹S ∪ {Neg p, Neg q} ∈ C›
   using ‹S ∈ C› conc unfolding consistency_def by simp
  then show ‹S' ∪ {Neg p, Neg q} ∈ close C›
   using ‹S' ⊆ S› subset_in_close by blast }

{ fix p q
  assume ‹Neg (Imp p q) ∈ S'›
  then have ‹Neg (Imp p q) ∈ S›
   using ‹S' ⊆ S› by blast
  then have ‹S ∪ {p, Neg q} ∈ C›
   using ‹S ∈ C› conc unfolding consistency_def by blast
  then show ‹S' ∪ {p, Neg q} ∈ close C›
   using ‹S' ⊆ S› subset_in_close by blast }

{ fix p q
  assume ‹Dis p q ∈ S'›
  then have ‹Dis p q ∈ S›
   using ‹S' ⊆ S› by blast
  then have ‹S ∪ {p} ∈ C ∨ S ∪ {q} ∈ C›
   using ‹S ∈ C› conc unfolding consistency_def by simp
  then show ‹S' ∪ {p} ∈ close C ∨ S' ∪ {q} ∈ close C›
   using ‹S' ⊆ S› subset_in_close by blast }

{ fix p q
```

```
  assume ‹Neg (Con p q) ∈ S'›
  then have ‹Neg (Con p q) ∈ S›
    using ‹S' ⊆ S› by blast
  then have ‹S ∪ {Neg p} ∈ C ∨ S ∪ {Neg q} ∈ C›
    using ‹S ∈ C› conc unfolding consistency_def by simp
  then show ‹S' ∪ {Neg p} ∈ close C ∨ S' ∪ {Neg q} ∈ close C›
    using ‹S' ⊆ S› subset_in_close by blast }

{ fix p q
  assume ‹Imp p q ∈ S'›
  then have ‹Imp p q ∈ S›
    using ‹S' ⊆ S› by blast
  then have ‹S ∪ {Neg p} ∈ C ∨ S ∪ {q} ∈ C›
    using ‹S ∈ C› conc unfolding consistency_def by simp
  then show ‹S' ∪ {Neg p} ∈ close C ∨ S' ∪ {q} ∈ close C›
    using ‹S' ⊆ S› subset_in_close by blast }

{ fix P t
  assume ‹closed_term 0 t› and ‹Uni P ∈ S'›
  then have ‹Uni P ∈ S›
    using ‹S' ⊆ S› by blast
  then have ‹S ∪ {sub 0 t P} ∈ C›
    using ‹closed_term 0 t› ‹S ∈ C› conc
    unfolding consistency_def by blast
  then show ‹S' ∪ {sub 0 t P} ∈ close C›
    using ‹S' ⊆ S› subset_in_close by blast }
```

```
{ fix P t
  assume ‹closed_term 0 t› and ‹Neg (Exi P) ∈ S'›
  then have ‹Neg (Exi P) ∈ S›
    using ‹S' ⊆ S› by blast
  then have ‹S ∪ {Neg (sub 0 t P)} ∈ C›
    using ‹closed_term 0 t› ‹S ∈ C› conc
    unfolding consistency_def by blast
  then show ‹S' ∪ {Neg (sub 0 t P)} ∈ close C›
    using ‹S' ⊆ S› subset_in_close by blast }

{ fix P
  assume ‹Exi P ∈ S'›
  then have ‹Exi P ∈ S›
    using ‹S' ⊆ S› by blast
  then have ‹∃c. S ∪ {sub 0 (Fun c []) P} ∈ C›
    using ‹S ∈ C› conc unfolding consistency_def by blast
  then show ‹∃c. S' ∪ {sub 0 (Fun c []) P} ∈ close C›
    using ‹S' ⊆ S› subset_in_close by blast }

{ fix P
  assume ‹Neg (Uni P) ∈ S'›
  then have ‹Neg (Uni P) ∈ S›
    using ‹S' ⊆ S› by blast
  then have ‹∃c. S ∪ {Neg (sub 0 (Fun c []) P)} ∈ C›
    using ‹S ∈ C› conc unfolding consistency_def by simp
  then show ‹∃c. S' ∪ {Neg (sub 0 (Fun c []) P)} ∈ close C›
    using ‹S' ⊆ S› subset_in_close by blast }
```

```
qed

theorem close_closed: ‹subset_closed (close C)›
  unfolding close_def subset_closed_def by blast

theorem close_subset: ‹C ⊆ close C›
  unfolding close_def by blast

theorem mk_alt_consistency_closed:
  assumes ‹subset_closed C›
  shows ‹subset_closed (mk_alt_consistency C)›
  unfolding subset_closed_def
proof (intro ballI allI impI)
  fix S S'
  assume ‹S ∈ mk_alt_consistency C› and ‹S' ⊆ S›
  then obtain f where *: ‹psubst f ` S ∈ C›
    unfolding mk_alt_consistency_def by blast
  moreover have ‹psubst f ` S' ⊆ psubst f ` S›
    using ‹S' ⊆ S› by blast
  ultimately have ‹psubst f ` S' ∈ C›
    using ‹subset_closed C› unfolding subset_closed_def by blast
  then show ‹S' ∈ mk_alt_consistency C›
    unfolding mk_alt_consistency_def by blast
qed

subsection ‹Finite Character›
```

```
definition finite_char :: ‹'a set set ⇒ bool› where
  ‹finite_char C = (∀S. S ∈ C = (∀S'. finite S' ⟶ S' ⊆ S ⟶ S' ∈ C))›

definition mk_finite_char :: ‹'a set set ⇒ 'a set set› where
  ‹mk_finite_char C = {S. ∀S'. S' ⊆ S ⟶ finite S' ⟶ S' ∈ C}›

theorem finite_char: ‹finite_char (mk_finite_char C)›
  unfolding finite_char_def mk_finite_char_def by blast

theorem finite_alt_consistency:
  assumes altconc: ‹alt_consistency C›
    and ‹subset_closed C›
  shows ‹alt_consistency (mk_finite_char C)›
  unfolding alt_consistency_def
proof (intro allI impI conjI)
  fix S
  assume ‹S ∈ mk_finite_char C›
  then have finc: ‹∀S' ⊆ S. finite S' ⟶ S' ∈ C›
    unfolding mk_finite_char_def by blast

  have ‹∀S' ∈ C. ∀S ⊆ S'. S ∈ C›
    using ‹subset_closed C› unfolding subset_closed_def by blast
  then have sc: ‹∀S' x. S' ∪ x ∈ C ⟶ (∀S ⊆ S' ∪ x. S ∈ C)›
    by blast

  { fix p ts
    show ‹¬ (Pre p ts ∈ S ∧ Neg (Pre p ts) ∈ S)›
```

```
  proof
    assume ‹Pre p ts ∈ S ∧ Neg (Pre p ts) ∈ S›
    then have ‹{Pre p ts, Neg (Pre p ts)} ∈ C›
      using finc by simp
    then show False
      using altconc unfolding alt_consistency_def by fast
  qed }

show ‹Falsity ∉ S›
proof
  assume ‹Falsity ∈ S›
  then have ‹{Falsity} ∈ C›
    using finc by simp
  then show False
    using altconc unfolding alt_consistency_def by fast
qed

{ fix p q
  assume *: ‹Con p q ∈ S›
  show ‹S ∪ {p, q} ∈ mk_finite_char C›
    unfolding mk_finite_char_def
  proof (intro allI impI CollectI)
    fix S'
    let ?S' = ‹S' - {p, q} ∪ {Con p q}›

    assume ‹S' ⊆ S ∪ {p, q}› and ‹finite S'›
    then have ‹?S' ⊆ S›
```

```
      using * by blast
    moreover have ‹finite ?S'›
      using ‹finite S'› by blast
    ultimately have ‹?S' ∈ C›
      using finc by blast
    then have ‹?S' ∪ {p, q} ∈ C›
      using altconc unfolding alt_consistency_def by simp
    then show ‹S' ∈ C›
      using sc by blast
  qed }

{ fix p q
  assume *: ‹Neg (Dis p q) ∈ S›
  show ‹S ∪ {Neg p, Neg q} ∈ mk_finite_char C›
    unfolding mk_finite_char_def
  proof (intro allI impI CollectI)
    fix S'
    let ?S' = ‹S' - {Neg p, Neg q} ∪ {Neg (Dis p q)}›

    assume ‹S' ⊆ S ∪ {Neg p, Neg q}› and ‹finite S'›
    then have ‹?S' ⊆ S›
      using * by blast
    moreover have ‹finite ?S'›
      using ‹finite S'› by blast
    ultimately have ‹?S' ∈ C›
      using finc by blast
    then have ‹?S' ∪ {Neg p, Neg q} ∈ C›
```

```
      using altconc unfolding alt_consistency_def by simp
    then show ‹S' ∈ C›
      using sc by blast
  qed }

{ fix p q
  assume *: ‹Neg (Imp p q) ∈ S›
  show ‹S ∪ {p, Neg q} ∈ mk_finite_char C›
    unfolding mk_finite_char_def
  proof (intro allI impI CollectI)
    fix S'
    let ?S' = ‹S' - {p, Neg q} ∪ {Neg (Imp p q)}›

    assume ‹S' ⊆ S ∪ {p, Neg q}› and ‹finite S'›
    then have ‹?S' ⊆ S›
      using * by blast
    moreover have ‹finite ?S'›
      using ‹finite S'› by blast
    ultimately have ‹?S' ∈ C›
      using finc by blast
    then have ‹?S' ∪ {p, Neg q} ∈ C›
      using altconc unfolding alt_consistency_def by simp
    then show ‹S' ∈ C›
      using sc by blast
  qed }

{ fix p q
```

```
  assume *: ‹Dis p q ∈ S›
  show ‹S ∪ {p} ∈ mk_finite_char C ∨ S ∪ {q} ∈ mk_finite_char C›
  proof (rule ccontr)
    assume ‹¬ ?thesis›
    then obtain Sa and Sb
      where ‹Sa ⊆ S ∪ {p}› and ‹finite Sa› and ‹Sa ∉ C›
        and ‹Sb ⊆ S ∪ {q}› and ‹finite Sb› and ‹Sb ∉ C›
      unfolding mk_finite_char_def by blast

    let ?S' = ‹(Sa - {p}) ∪ (Sb - {q}) ∪ {Dis p q}›

    have ‹?S' ⊆ S›
      using ‹Sa ⊆ S ∪ {p}› ‹Sb ⊆ S ∪ {q}› * by blast
    moreover have ‹finite ?S'›
      using ‹finite Sa› ‹finite Sb› by blast
    ultimately have ‹?S' ∈ C›
      using finc by blast
    then have ‹?S' ∪ {p} ∈ C ∨ ?S' ∪ {q} ∈ C›
      using altconc unfolding alt_consistency_def by simp
    then have ‹Sa ∈ C ∨ Sb ∈ C›
      using sc by blast
    then show False
      using ‹Sa ∉ C› ‹Sb ∉ C› by blast
  qed }

{ fix p q
  assume *: ‹Neg (Con p q) ∈ S›
```

```
  show ‹S ∪ {Neg p} ∈ mk_finite_char C ∨ S ∪ {Neg q} ∈ mk_finite_char C›
  proof (rule ccontr)
    assume ‹¬ ?thesis›
    then obtain Sa and Sb
      where ‹Sa ⊆ S ∪ {Neg p}› and ‹finite Sa› and ‹Sa ∉ C›
        and ‹Sb ⊆ S ∪ {Neg q}› and ‹finite Sb› and ‹Sb ∉ C›
      unfolding mk_finite_char_def by blast

    let ?S' = ‹(Sa - {Neg p}) ∪ (Sb - {Neg q}) ∪ {Neg (Con p q)}›

    have ‹?S' ⊆ S›
      using ‹Sa ⊆ S ∪ {Neg p}› ‹Sb ⊆ S ∪ {Neg q}› * by blast
    moreover have ‹finite ?S'›
      using ‹finite Sa› ‹finite Sb› by blast
    ultimately have ‹?S' ∈ C›
      using finc by blast
    then have ‹?S' ∪ {Neg p} ∈ C ∨ ?S' ∪ {Neg q} ∈ C›
      using altconc unfolding alt_consistency_def by simp
    then have ‹Sa ∈ C ∨ Sb ∈ C›
      using sc by blast
    then show False
      using ‹Sa ∉ C› ‹Sb ∉ C› by blast
  qed }

{ fix p q
  assume *: ‹Imp p q ∈ S›
  show ‹S ∪ {Neg p} ∈ mk_finite_char C ∨ S ∪ {q} ∈ mk_finite_char C›
```

```
  proof (rule ccontr)
    assume ‹¬ ?thesis›
    then obtain Sa and Sb
      where ‹Sa ⊆ S ∪ {Neg p}› and ‹finite Sa› and ‹Sa ∉ C›
        and ‹Sb ⊆ S ∪ {q}› and ‹finite Sb› and ‹Sb ∉ C›
      unfolding mk_finite_char_def by blast

    let ?S' = ‹(Sa - {Neg p}) ∪ (Sb - {q}) ∪ {Imp p q}›

    have ‹?S' ⊆ S›
      using ‹Sa ⊆ S ∪ {Neg p}› ‹Sb ⊆ S ∪ {q}› * by blast
    moreover have ‹finite ?S'›
      using ‹finite Sa› ‹finite Sb› by blast
    ultimately have ‹?S' ∈ C›
      using finc by blast
    then have ‹?S' ∪ {Neg p} ∈ C ∨ ?S' ∪ {q} ∈ C›
      using altconc unfolding alt_consistency_def by simp
    then have ‹Sa ∈ C ∨ Sb ∈ C›
      using sc by blast
    then show False
      using ‹Sa ∉ C› ‹Sb ∉ C› by blast
  qed }

{ fix P t
  assume *: ‹Uni P ∈ S› and ‹closed_term 0 t›
  show ‹S ∪ {sub 0 t P} ∈ mk_finite_char C›
    unfolding mk_finite_char_def
```

```
  proof (intro allI impI CollectI)
    fix S'
    let ?S' = ‹S' - {sub 0 t P} ∪ {Uni P}›

    assume ‹S' ⊆ S ∪ {sub 0 t P}› and ‹finite S'›
    then have ‹?S' ⊆ S›
      using * by blast
    moreover have ‹finite ?S'›
      using ‹finite S'› by blast
    ultimately have ‹?S' ∈ C›
      using finc by blast
    then have ‹?S' ∪ {sub 0 t P} ∈ C›
      using altconc ‹closed_term 0 t›
      unfolding alt_consistency_def by simp
    then show ‹S' ∈ C›
      using sc by blast
  qed }

{ fix P t
  assume *: ‹Neg (Exi P) ∈ S› and ‹closed_term 0 t›
  show ‹S ∪ {Neg (sub 0 t P)} ∈ mk_finite_char C›
    unfolding mk_finite_char_def
  proof (intro allI impI CollectI)
    fix S'
    let ?S' = ‹S' - {Neg (sub 0 t P)} ∪ {Neg (Exi P)}›

    assume ‹S' ⊆ S ∪ {Neg (sub 0 t P)}› and ‹finite S'›
```

```
    then have ‹?S' ⊆ S›
      using * by blast
    moreover have ‹finite ?S'›
      using ‹finite S'› by blast
    ultimately have ‹?S' ∈ C›
      using finc by blast
    then have ‹?S' ∪ {Neg (sub 0 t P)} ∈ C›
      using altconc ‹closed_term 0 t›
      unfolding alt_consistency_def by simp
    then show ‹S' ∈ C›
      using sc by blast
  qed }

{ fix P x
  assume *: ‹Exi P ∈ S› and ‹∀a ∈ S. x ∉ params a›
  show ‹S ∪ {sub 0 (Fun x []) P} ∈ mk_finite_char C›
    unfolding mk_finite_char_def
  proof (intro allI impI CollectI)
    fix S'
    let ?S' = ‹(S' - {sub 0 (Fun x []) P}) ∪ {Exi P}›

    assume ‹S' ⊆ S ∪ {sub 0 (Fun x []) P}› and ‹finite S'›
    then have ‹?S' ⊆ S›
      using * by blast
    moreover have ‹finite ?S'›
      using ‹finite S'› by blast
    ultimately have ‹?S' ∈ C›
```

```
    using finc by blast
   moreover have ‹∀a ∈ ?S'. x ∉ params a›
    using ‹∀a ∈ S. x ∉ params a› ‹?S' ⊆ S› by blast
   ultimately have ‹?S' ∪ {sub 0 (Fun x []) P} ∈ C›
    using altconc ‹∀a ∈ S. x ∉ params a›
    unfolding alt_consistency_def by blast
   then show ‹S' ∈ C›
    using sc by blast
  qed }

 { fix P x
  assume *: ‹Neg (Uni P) ∈ S› and ‹∀a ∈ S. x ∉ params a›
  show ‹S ∪ {Neg (sub 0 (Fun x []) P)} ∈ mk_finite_char C›
   unfolding mk_finite_char_def
  proof (intro allI impI CollectI)
   fix S'
   let ?S' = ‹S' - {Neg (sub 0 (Fun x []) P)} ∪ {Neg (Uni P)}›

   assume ‹S' ⊆ S ∪ {Neg (sub 0 (Fun x []) P)}› and ‹finite S'›
   then have ‹?S' ⊆ S›
    using * by blast
   moreover have ‹finite ?S'›
    using ‹finite S'› by blast
   ultimately have ‹?S' ∈ C›
    using finc by blast
   moreover have ‹∀a ∈ ?S'. x ∉ params a›
    using ‹∀a ∈ S. x ∉ params a› ‹?S' ⊆ S› by blast
```

```
    ultimately have ‹?S' ∪ {Neg (sub 0 (Fun x []) P)} ∈ C›
      using altconc ‹∀a ∈ S. x ∉ params a›
      unfolding alt_consistency_def by simp
    then show ‹S' ∈ C›
      using sc by blast
  qed }
qed

theorem finite_char_closed: ‹finite_char C ⟹ subset_closed C›
  unfolding finite_char_def subset_closed_def
proof (intro ballI allI impI)
  fix S S'
  assume *: ‹∀S. (S ∈ C) = (∀S'. finite S' ⟶ S' ⊆ S ⟶ S' ∈ C)›
    and ‹S' ∈ C› and ‹S ⊆ S'›
  then have ‹∀S'. finite S' ⟶ S' ⊆ S ⟶ S' ∈ C› by blast
  then show ‹S ∈ C› using * by blast
qed

theorem finite_char_subset: ‹subset_closed C ⟹ C ⊆ mk_finite_char C›
  unfolding mk_finite_char_def subset_closed_def by blast

subsection ‹Enumerating Datatypes›

primrec diag :: ‹nat ⇒ (nat × nat)› where
  ‹diag 0 = (0, 0)›
| ‹diag (Suc n) =
     (let (x, y) = diag n
```

```
    in case y of
      0 ⇒ (0, Suc x)
     | Suc y ⇒ (Suc x, y))›

theorem diag_le1: ‹fst (diag (Suc n)) < Suc n›
 by (induct n) (simp_all add: Let_def split_def split: nat.split)

theorem diag_le2: ‹snd (diag (Suc (Suc n))) < Suc (Suc n)›
proof (induct n)
 case 0
 then show ?case by simp
next
 case (Suc n')
 then show ?case
 proof (induct n')
  case 0
  then show ?case by simp
 next
  case (Suc _)
  then show ?case
   using diag_le1 by (simp add: Let_def split_def split: nat.split)
 qed
qed

theorem diag_le3: ‹fst (diag n) = Suc x ⟹ snd (diag n) < n›
proof (induct n)
 case 0
```

```
  then show ?case by simp
next
  case (Suc n')
  then show ?case
  proof (induct n')
    case 0
    then show ?case by simp
  next
    case (Suc n'')
    then show ?case using diag_le2 by simp
  qed
qed

theorem diag_le4: ‹fst (diag n) = Suc x ⟹ x < n›
proof (induct n)
  case 0
  then show ?case by simp
next
  case (Suc n')
  then have ‹fst (diag (Suc n')) < Suc n'›
    using diag_le1 by blast
  then show ?case using Suc by simp
qed

function undiag :: ‹nat × nat ⇒ nat› where
  ‹undiag (0, 0) = 0›
| ‹undiag (0, Suc y) = Suc (undiag (y, 0))›
```

```
| ‹undiag (Suc x, y) = Suc (undiag (x, Suc y))›
  by pat_completeness auto
termination
  by (relation ‹measure (λ(x, y). ((x + y) * (x + y + 1)) div 2 + x)›) auto

theorem diag_undiag [simp]: ‹diag (undiag (x, y)) = (x, y)›
  by (induct rule: undiag.induct) simp_all

datatype btree = Leaf nat | Branch btree btree

function diag_btree :: ‹nat ⇒ btree› where
  ‹diag_btree n = (case fst (diag n) of
      0 ⇒ Leaf (snd (diag n))
    | Suc x ⇒ Branch (diag_btree x) (diag_btree (snd (diag n))))›
  by auto
termination
  by (relation ‹measure id›) (auto intro: diag_le3 diag_le4)

primrec undiag_btree :: ‹btree ⇒ nat› where
  ‹undiag_btree (Leaf n) = undiag (0, n)›
| ‹undiag_btree (Branch t1 t2) =
    undiag (Suc (undiag_btree t1), undiag_btree t2)›

theorem diag_undiag_btree [simp]: ‹diag_btree (undiag_btree t) = t›
  by (induct t) simp_all

declare diag_btree.simps [simp del] undiag_btree.simps [simp del]
```

```
fun list_of_btree :: ‹(nat ⇒ 'a) ⇒ btree ⇒ 'a list› where
  ‹list_of_btree f (Leaf x) = []›
| ‹list_of_btree f (Branch (Leaf n) t) = f n # list_of_btree f t›
| ‹list_of_btree f _ = undefined›

primrec btree_of_list :: ‹('a ⇒ nat) ⇒ 'a list ⇒ btree› where
  ‹btree_of_list f [] = Leaf 0›
| ‹btree_of_list f (x # xs) = Branch (Leaf (f x)) (btree_of_list f xs)›

definition diag_list :: ‹(nat ⇒ 'a) ⇒ nat ⇒ 'a list› where
  ‹diag_list f n = list_of_btree f (diag_btree n)›

definition undiag_list :: ‹('a ⇒ nat) ⇒ 'a list ⇒ nat› where
  ‹undiag_list f xs = undiag_btree (btree_of_list f xs)›

theorem diag_undiag_list [simp]: ‹(⋀x. d (u x) = x) ⟹ diag_list d (undiag_list u xs) = xs›
  by (induct xs) (simp_all add: diag_list_def undiag_list_def)

fun string_of_btree :: ‹btree ⇒ string› where
  ‹string_of_btree (Leaf x) = []›
| ‹string_of_btree (Branch (Leaf n) t) = char_of n # string_of_btree t›
| ‹string_of_btree _ = undefined›

primrec btree_of_string :: ‹string ⇒ btree› where
  ‹btree_of_string [] = Leaf 0›
| ‹btree_of_string (x # xs) = Branch (Leaf (of_char x)) (btree_of_string xs)›
```

```
definition diag_string :: ‹nat ⇒ string› where
  ‹diag_string n = string_of_btree (diag_btree n)›

definition undiag_string :: ‹string ⇒ nat› where
  ‹undiag_string xs = undiag_btree (btree_of_string xs)›

theorem diag_undiag_string [simp]: ‹diag_string (undiag_string xs) = xs›
  by (induct xs) (simp_all add: diag_string_def undiag_string_def)

lemma inj_undiag_string: ‹inj undiag_string›
  by (metis diag_undiag_string inj_onI)

fun
  term_of_btree :: ‹btree ⇒ tm› and
  term_list_of_btree :: ‹btree ⇒ tm list› where
  ‹term_of_btree (Leaf m) = Var m›
| ‹term_of_btree (Branch (Leaf m) t) =
    Fun (diag_string m) (term_list_of_btree t)›
| ‹term_list_of_btree (Leaf m) = []›
| ‹term_list_of_btree (Branch t1 t2) =
    term_of_btree t1 # term_list_of_btree t2›
| ‹term_of_btree (Branch (Branch _ _) _) = undefined›

primrec
  btree_of_term :: ‹tm ⇒ btree› and
  btree_of_term_list :: ‹tm list ⇒ btree› where
```

```
  ‹btree_of_term (Var m) = Leaf m›
| ‹btree_of_term (Fun m ts) = Branch (Leaf (undiag_string m)) (btree_of_term_list ts)›
| ‹btree_of_term_list [] = Leaf 0›
| ‹btree_of_term_list (t # ts) = Branch (btree_of_term t) (btree_of_term_list ts)›

theorem term_btree:
  shows ‹term_of_btree (btree_of_term t) = t›
    and ‹term_list_of_btree (btree_of_term_list ts) = ts›
  by (induct t and ts rule: btree_of_term.induct btree_of_term_list.induct) simp_all

definition diag_term :: ‹nat ⇒ tm› where
  ‹diag_term n = term_of_btree (diag_btree n)›

definition undiag_term :: ‹tm ⇒ nat› where
  ‹undiag_term t = undiag_btree (btree_of_term t)›

theorem diag_undiag_term [simp]: ‹diag_term (undiag_term t) = t›
  unfolding diag_term_def undiag_term_def using term_btree by simp

fun form_of_btree :: ‹btree ⇒ fm› where
  ‹form_of_btree (Leaf 0) = Falsity›
| ‹form_of_btree (Branch (Leaf 0) (Branch (Leaf m) (Leaf n))) =
    Pre (diag_string m) (diag_list diag_term n)›
| ‹form_of_btree (Branch (Leaf (Suc 0)) (Branch t1 t2)) =
    Con (form_of_btree t1) (form_of_btree t2)›
| ‹form_of_btree (Branch (Leaf (Suc (Suc 0))) (Branch t1 t2)) =
    Dis (form_of_btree t1) (form_of_btree t2)›
```

```
| ‹form_of_btree (Branch (Leaf (Suc (Suc (Suc 0)))) (Branch t1 t2)) =
   Imp (form_of_btree t1) (form_of_btree t2)›
| ‹form_of_btree (Branch (Leaf (Suc (Suc (Suc (Suc 0))))) t) =
   Uni (form_of_btree t)›
| ‹form_of_btree (Branch (Leaf (Suc (Suc (Suc (Suc (Suc 0)))))) t) =
   Exi (form_of_btree t)›
| ‹form_of_btree (Leaf (Suc _)) = undefined›
| ‹form_of_btree (Branch (Leaf (Suc (Suc (Suc (Suc (Suc (Suc _))))))) _) = undefined›
| ‹form_of_btree (Branch (Leaf (Suc (Suc (Suc 0)))) (Leaf _)) = undefined›
| ‹form_of_btree (Branch (Leaf (Suc (Suc 0))) (Leaf _)) = undefined›
| ‹form_of_btree (Branch (Leaf (Suc 0)) (Leaf _)) = undefined›
| ‹form_of_btree (Branch (Branch _ _) _) = undefined›
| ‹form_of_btree (Branch (Leaf 0) (Leaf _)) = undefined›
| ‹form_of_btree (Branch (Leaf 0) (Branch (Branch _ _) _)) = undefined›
| ‹form_of_btree (Branch (Leaf 0) (Branch (Leaf _) (Branch _ _))) = undefined›

primrec btree_of_form :: ‹fm ⇒ btree› where
 ‹btree_of_form Falsity = Leaf 0›
| ‹btree_of_form (Pre b ts) = Branch (Leaf 0)
   (Branch (Leaf (undiag_string b)) (Leaf (undiag_list undiag_term ts)))›
| ‹btree_of_form (Con a b) = Branch (Leaf (Suc 0))
   (Branch (btree_of_form a) (btree_of_form b))›
| ‹btree_of_form (Dis a b) = Branch (Leaf (Suc (Suc 0)))
   (Branch (btree_of_form a) (btree_of_form b))›
| ‹btree_of_form (Imp a b) = Branch (Leaf (Suc (Suc (Suc 0))))
   (Branch (btree_of_form a) (btree_of_form b))›
| ‹btree_of_form (Uni a) = Branch (Leaf (Suc (Suc (Suc (Suc 0)))))
```

```
  (btree_of_form a)›
| ‹btree_of_form (Exi a) = Branch (Leaf (Suc (Suc (Suc (Suc (Suc 0))))))
  (btree_of_form a)›

definition diag_form :: ‹nat ⇒ fm› where
 ‹diag_form n = form_of_btree (diag_btree n)›

definition undiag_form :: ‹fm ⇒ nat› where
 ‹undiag_form x = undiag_btree (btree_of_form x)›

theorem diag_undiag_form [simp]: ‹diag_form (undiag_form f) = f›
 unfolding diag_form_def undiag_form_def by (induct f) simp_all

definition diag_form' :: ‹nat ⇒ fm› where
 ‹diag_form' = diag_form›

definition undiag_form' :: ‹fm ⇒ nat› where
 ‹undiag_form' = undiag_form›

theorem diag_undiag_form' [simp]: ‹diag_form' (undiag_form' f) = f›
 by (simp add: diag_form'_def undiag_form'_def)

abbreviation ‹from_nat ≡ diag_form'›
abbreviation ‹to_nat ≡ undiag_form'›

subsection ‹Extension to Maximal Consistent Sets›
```

```
definition is_chain :: ‹(nat ⇒ 'a set) ⇒ bool› where
  ‹is_chain f = (∀n. f n ⊆ f (Suc n))›

lemma is_chainD: ‹is_chain f ⟹ x ∈ f m ⟹ x ∈ f (m + n)›
  unfolding is_chain_def by (induct n) auto

lemma is_chainD':
  assumes ‹is_chain f› and ‹x ∈ f m› and ‹m ≤ k›
  shows ‹x ∈ f k›
proof -
  have ‹∃n. k = m + n›
    using ‹m ≤ k› by (simp add: le_iff_add)
  then obtain n where ‹k = m + n›
    by blast
  then show ‹x ∈ f k›
    using ‹is_chain f› ‹x ∈ f m›
    by (simp add: is_chainD)
qed

lemma chain_index:
  assumes ch: ‹is_chain f› and fin: ‹finite F›
  shows ‹F ⊆ (⋃n. f n) ⟹ ∃n. F ⊆ f n›
  using fin
proof (induct rule: finite_induct)
  case empty
  then show ?case by blast
next
```

```
  case (insert x F)
  then have ‹∃n. F ⊆ f n› and ‹∃m. x ∈ f m› and ‹F ⊆ (⋃x. f x)›
    using ch by simp_all
  then obtain n and m where ‹F ⊆ f n› and ‹x ∈ f m›
    by blast
  have ‹m ≤ max n m› and ‹n ≤ max n m›
    by simp_all
  have ‹x ∈ f (max n m)›
    using is_chainD' ch ‹x ∈ f m› ‹m ≤ max n m› by fast
  moreover have ‹F ⊆ f (max n m)›
    using is_chainD' ch ‹F ⊆ f n› ‹n ≤ max n m› by fast
  ultimately show ?case by blast
qed

lemma chain_union_closed':
  assumes ‹is_chain f› and ‹∀n. f n ∈ C› and ‹∀S' ∈ C. ∀S ⊆ S'. S ∈ C›
    and ‹finite S'› and ‹S' ⊆ (⋃n. f n)›
  shows ‹S' ∈ C›
proof -
  note ‹finite S'› and ‹S' ⊆ (⋃n. f n)›
  then obtain n where ‹S' ⊆ f n›
    using chain_index ‹is_chain f› by blast
  moreover have ‹f n ∈ C›
    using ‹∀n. f n ∈ C› by blast
  ultimately show ‹S' ∈ C›
    using ‹∀S' ∈ C. ∀S ⊆ S'. S ∈ C› by blast
qed
```

```
theorem chain_union_closed:
  assumes ‹finite_char C› and ‹is_chain f› and ‹∀n. f n ∈ C›
  shows ‹(⋃n. f n) ∈ C›
proof -
  have ‹subset_closed C›
    using finite_char_closed ‹finite_char C› by blast
  then have ‹∀S' ∈ C. ∀S ⊆ S'. S ∈ C›
    using subset_closed_def by blast
  then have ‹∀S'. finite S' ⟶ S' ⊆ (⋃n. f n) ⟶ S' ∈ C›
    using chain_union_closed' assms by blast
  moreover have ‹((⋃n. f n) ∈ C) = (∀S'. finite S' ⟶ S' ⊆ (⋃n. f n) ⟶ S' ∈ C)›
    using ‹finite_char C› unfolding finite_char_def by blast
  ultimately show ?thesis by blast
qed

abbreviation dest_Neg :: ‹fm ⇒ fm› where
  ‹dest_Neg p ≡ (case p of (Imp p' Falsity) ⇒ p' | p' ⇒ p')›

abbreviation dest_Uni :: ‹fm ⇒ fm› where
  ‹dest_Uni p ≡ (case p of (Uni p') ⇒ p' | p' ⇒ p')›

abbreviation dest_Exi :: ‹fm ⇒ fm› where
  ‹dest_Exi p ≡ (case p of (Exi p') ⇒ p' | p' ⇒ p')›

primrec extend :: ‹fm set ⇒ fm set set ⇒ (nat ⇒ fm) ⇒ nat ⇒ fm set› where
  ‹extend S C f 0 = S› |
```

```
‹extend S C f (Suc n) = (if extend S C f n ∪ {f n} ∈ C
  then (if (∃p. f n = Exi p)
    then extend S C f n ∪ {f n} ∪ {sub 0
      (Fun (SOME k. k ∉ (⋃p ∈ extend S C f n ∪ {f n}. params p)) [])
      (dest_Exi (f n))}
    else if (∃p. f n = Neg (Uni p))
    then extend S C f n ∪ {f n} ∪ {Neg (sub 0
      (Fun (SOME k. k ∉ (⋃p ∈ extend S C f n ∪ {f n}. params p)) [])
      (dest_Uni (dest_Neg (f n))))}
    else extend S C f n ∪ {f n})
  else extend S C f n)›

definition Extend :: ‹fm set ⇒ fm set set ⇒ (nat ⇒ fm) ⇒ fm set› where
 ‹Extend S C f = (⋃n. extend S C f n)›

theorem is_chain_extend: ‹is_chain (extend S C f)›
 by (simp add: is_chain_def) blast

lemma finite_params' [simp]: ‹finite (params_term t)› ‹finite (params_list l)›
 by (induct t and l rule: params_term.induct params_list.induct) simp_all

lemma finite_params [simp]: ‹finite (params p)›
 by (induct p) simp_all

lemma finite_params_extend [simp]:
 ‹infinite (⋂p ∈ S. - params p) ⟹ infinite (⋂p ∈ extend S C f n. - params p)›
 by (induct n) (simp_all add: set_inter_compl_diff)
```

```
lemma infinite_params_available:
  assumes ‹infinite (- (⋃p ∈ S. params p))›
  shows ‹∃x. x ∉ (⋃p ∈ extend S C f n ∪ {f n}. params p)›
    (is ‹_ (⋃_ ∈ ?S'. _)›)
proof -
  have ‹infinite (- (⋃p ∈ ?S'. params p))›
    using assms by (simp add: set_inter_compl_diff)
  then obtain x where ‹x ∈ - (⋃p ∈ ?S'. params p)›
    using infinite_imp_nonempty by blast
  then show ‹∃x. x ∉ (⋃p ∈ ?S'. params p)›
    by blast
qed

lemma extend_in_C_Exi:
  assumes ‹alt_consistency C›
    and ‹infinite (- (⋃p ∈ S. params p))›
    and ‹extend S C f n ∪ {f n} ∈ C› (is ‹?S' ∈ C›)
    and ‹∃p. f n = Exi p›
  shows ‹extend S C f (Suc n) ∈ C›
proof -
  obtain p where *: ‹f n = Exi p›
    using ‹∃p. f n = Exi p› by blast

  let ?x = ‹(SOME k. k ∉ (⋃p ∈ ?S'. params p))›

  from ‹infinite (- (⋃p ∈ S. params p))›
```

```
  have ‹∃x. x ∉ (⋃p ∈ ?S'. params p)›
    using infinite_params_available by blast
  then have ‹?x ∉ (⋃p ∈ ?S'. params p)›
    using someI_ex by metis
  then have ‹(?S' ∪ {sub 0 (Fun ?x []) p}) ∈ C›
    using * ‹?S' ∈ C› ‹alt_consistency C›
    unfolding alt_consistency_def by simp
  then show ?thesis
    using assms * by simp
qed

lemma extend_in_C_Neg_Uni:
  assumes ‹alt_consistency C›
    and ‹infinite (- (⋃p ∈ S. params p))›
    and ‹extend S C f n ∪ {f n} ∈ C› (is ‹?S' ∈ C›)
    and ‹∀p. f n ≠ Exi p›
    and ‹∃p. f n = Neg (Uni p)›
  shows ‹extend S C f (Suc n) ∈ C›
proof -
  obtain p where *: ‹f n = Neg (Uni p)›
    using ‹∃p. f n = Neg (Uni p)› by blast

  let ?x = ‹(SOME k. k ∉ (⋃p ∈ ?S'. params p))›

  have ‹∃x. x ∉ (⋃p ∈ ?S'. params p)›
    using ‹infinite (- (⋃p ∈ S. params p))› infinite_params_available by blast
  then have ‹?x ∉ (⋃p ∈ ?S'. params p)›
```

```
    using someI_ex by metis
  moreover have ‹Neg (Uni p) ∈ ?S'›
    using * by simp
  ultimately have ‹?S' ∪ {Neg (sub 0 (Fun ?x []) p)} ∈ C›
    using ‹?S' ∈ C› ‹alt_consistency C› unfolding alt_consistency_def by simp
  then show ?thesis
    using assms * by simp
qed

lemma extend_in_C_no_delta:
  assumes ‹extend S C f n ∪ {f n} ∈ C›
    and ‹∀p. f n ≠ Exi p›
    and ‹∀p. f n ≠ Neg (Uni p)›
  shows ‹extend S C f (Suc n) ∈ C›
  using assms by simp

lemma extend_in_C_stop:
  assumes ‹extend S C f n ∈ C›
    and ‹extend S C f n ∪ {f n} ∉ C›
  shows ‹extend S C f (Suc n) ∈ C›
  using assms by simp

theorem extend_in_C:
  ‹alt_consistency C ⟹ S ∈ C ⟹ infinite (- (⋃p ∈ S. params p)) ⟹ extend S C f n ∈ C›
proof (induct n)
  case 0
  then show ?case by simp
```

```
next
  case (Suc n)
  then show ?case
    using extend_in_C_Exi extend_in_C_Neg_Uni
      extend_in_C_no_delta extend_in_C_stop by metis
qed

theorem Extend_in_C: ‹alt_consistency C ⟹ finite_char C ⟹
    S ∈ C ⟹ infinite (- (⋃p ∈ S. params p)) ⟹ Extend S C f ∈ C›
  using chain_union_closed is_chain_extend extend_in_C
  unfolding Extend_def by blast

theorem Extend_subset: ‹S ⊆ Extend S C f›
  unfolding Extend_def using Union_upper extend.simps(1) range_eqI by metis

definition maximal :: ‹'a set ⇒ 'a set set ⇒ bool› where
  ‹maximal S C = (∀S' ∈ C. S ⊆ S' ⟶ S = S')›

theorem Extend_maximal:
  assumes ‹∀y :: fm. ∃n. y = f n› and ‹finite_char C›
  shows ‹maximal (Extend S C f) C›
  unfolding maximal_def Extend_def
proof (intro ballI impI)
  fix S'
  assume ‹S' ∈ C› and ‹(⋃x. extend S C f x) ⊆ S'›
  moreover have ‹S' ⊆ (⋃x. extend S C f x)›
  proof (rule ccontr)
```

```
    assume ‹¬ S' ⊆ (⋃x. extend S C f x)›
    then obtain z where ‹z ∈ S'› and *: ‹z ∉ (⋃x. extend S C f x)›
      by blast
    then obtain n where ‹z = f n›
      using ‹∀y. ∃n. y = f n› by blast

    from ‹(⋃x. extend S C f x) ⊆ S'› ‹z = f n› ‹z ∈ S'›
    have ‹extend S C f n ∪ {f n} ⊆ S'› by blast

    from ‹finite_char C›
    have ‹subset_closed C›
      using finite_char_closed by blast
    then have ‹∀S' ∈ C. ∀S ⊆ S'. S ∈ C›
      unfolding subset_closed_def by simp
    then have ‹∀S ⊆ S'. S ∈ C›
      using ‹S' ∈ C› by blast
    then have ‹extend S C f n ∪ {f n} ∈ C›
      using ‹extend S C f n ∪ {f n} ⊆ S'› by blast
    then have ‹z ∈ extend S C f (Suc n)›
      using ‹z ∉ (⋃x. extend S C f x)› ‹z = f n› by simp
    then show False
      using * by blast
  qed
  ultimately show ‹(⋃x. extend S C f x) = S'›
    by simp
qed
```

```
subsection ‹Hintikka Sets and Herbrand Models›

definition hintikka :: ‹fm set ⇒ bool› where
  ‹hintikka H =
    ((∀p ts. ¬ (Pre p ts ∈ H ∧ Neg (Pre p ts) ∈ H)) ∧
    Falsity ∉ H ∧
    (∀p q. Con p q ∈ H ⟶ p ∈ H ∧ q ∈ H) ∧
    (∀p q. Neg (Dis p q) ∈ H ⟶ Neg p ∈ H ∧ Neg q ∈ H) ∧
    (∀p q. Dis p q ∈ H ⟶ p ∈ H ∨ q ∈ H) ∧
    (∀p q. Neg (Con p q) ∈ H ⟶ Neg p ∈ H ∨ Neg q ∈ H) ∧
    (∀p q. Imp p q ∈ H ⟶ Neg p ∈ H ∨ q ∈ H) ∧
    (∀p q. Neg (Imp p q) ∈ H ⟶ p ∈ H ∧ Neg q ∈ H) ∧
    (∀P t. closed_term 0 t ⟶ Uni P ∈ H ⟶ sub 0 t P ∈ H) ∧
    (∀P t. closed_term 0 t ⟶ Neg (Exi P) ∈ H ⟶ Neg (sub 0 t P) ∈ H) ∧
    (∀P. Exi P ∈ H ⟶ (∃t. closed_term 0 t ∧ sub 0 t P ∈ H)) ∧
    (∀P. Neg (Uni P) ∈ H ⟶ (∃t. closed_term 0 t ∧ Neg (sub 0 t P) ∈ H)))›

datatype htm = HFun id ‹htm list›

primrec
  tm_of_htm :: ‹htm ⇒ tm› and
  tms_of_htms :: ‹htm list ⇒ tm list› where
  ‹tm_of_htm (HFun a hts) = Fun a (tms_of_htms hts)› |
  ‹tms_of_htms [] = []› |
  ‹tms_of_htms (ht # hts) = tm_of_htm ht # tms_of_htms hts›

lemma herbrand_semantics [simp]:
```

```
  ‹closed_term 0 t ⟹ tm_of_htm (semantics_term e HFun t) = t›
  ‹closed_list 0 l ⟹ tms_of_htms (semantics_list e HFun l) = l›
  by (induct t and l rule: closed_term.induct closed_list.induct) simp_all

lemma herbrand_semantics' [simp]:
  ‹semantics_term e HFun (tm_of_htm ht) = ht›
  ‹semantics_list e HFun (tms_of_htms hts) = hts›
  by (induct ht and hts rule: tm_of_htm.induct tms_of_htms.induct) simp_all

theorem closed_htm [simp]:
  ‹closed_term 0 (tm_of_htm ht)›
  ‹closed_list 0 (tms_of_htms hts)›
  by (induct ht and hts rule: tm_of_htm.induct tms_of_htms.induct) simp_all

theorem hintikka_model:
  assumes hin: ‹hintikka H›
  shows ‹(p ∈ H ⟶ closed 0 p ⟶ semantics e HFun (λi l. Pre i (tms_of_htms l) ∈ H) p) ∧
    (Neg p ∈ H ⟶ closed 0 p ⟶ semantics e HFun (λi l. Pre i (tms_of_htms l) ∈ H) (Neg p))›
proof (induct p rule: wf_induct)
  show ‹wf (measure size_formulas)›
    by blast
next
  let ?semantics = ‹semantics e HFun (λi l. Pre i (tms_of_htms l) ∈ H)›

  fix x
  assume wf: ‹∀y. (y, x) ∈ measure size_formulas ⟶
     (y ∈ H ⟶ closed 0 y ⟶ ?semantics y) ∧
```

```
(Neg y ∈ H ⟶ closed 0 y ⟶ ?semantics (Neg y))›

show ‹(x ∈ H ⟶ closed 0 x ⟶ ?semantics x) ∧
  (Neg x ∈ H ⟶ closed 0 x ⟶ ?semantics (Neg x))›
proof (cases x)
  case Falsity
  show ?thesis
  proof (intro conjI impI)
    assume ‹x ∈ H›
    then show ‹?semantics x›
      using Falsity hin by (simp add: hintikka_def)
  next
    assume ‹Neg x ∈ H›
    then show ‹?semantics (Neg x)›
      using Falsity by simp
  qed
next
  case (Pre p ts)
  show ?thesis
  proof (intro conjI impI)
    assume ‹x ∈ H› and ‹closed 0 x›
    then show ‹?semantics x›
      using Pre by simp
  next
    assume ‹Neg x ∈ H› and ‹closed 0 x›
    then have ‹Neg (Pre p ts) ∈ H›
      using Pre by simp
```

```
    then have ‹Pre p ts ∉ H›
      using hin unfolding hintikka_def by meson
    then show ‹?semantics (Neg x)›
      using Pre ‹closed 0 x› by simp
  qed
next
  case (Con p q)
  then show ?thesis
  proof (intro conjI impI)
    assume ‹x ∈ H› and ‹closed 0 x›
    then have ‹Con p q ∈ H› and ‹closed 0 (Con p q)›
      using Con by simp_all
    then have ‹p ∈ H ∧ q ∈ H›
      using Con hin unfolding hintikka_def by blast
    then show ‹?semantics x›
      using Con wf ‹closed 0 (Con p q)› by simp
  next
    assume ‹Neg x ∈ H› and ‹closed 0 x›
    then have ‹Neg (Con p q) ∈ H› and ‹closed 0 (Con p q)›
      using Con by simp_all
    then have ‹Neg p ∈ H ∨ Neg q ∈ H›
      using hin unfolding hintikka_def by meson
    then show ‹?semantics (Neg x)›
      using Con wf ‹closed 0 (Con p q)› by force
  qed
next
  case (Dis p q)
```

```
  then show ?thesis
  proof (intro conjI impI)
    assume ‹x ∈ H› and ‹closed 0 x›
    then have ‹Dis p q ∈ H› and ‹closed 0 (Dis p q)›
      using Dis by simp_all
    then have ‹p ∈ H ∨ q ∈ H›
      using hin unfolding hintikka_def by meson
    then show ‹?semantics x›
      using Dis wf ‹closed 0 (Dis p q)› by fastforce
  next
    assume ‹Neg x ∈ H› and ‹closed 0 x›
    then have ‹Neg (Dis p q) ∈ H› and ‹closed 0 (Dis p q)›
      using Dis by simp_all
    then have ‹Neg p ∈ H ∧ Neg q ∈ H›
      using hin unfolding hintikka_def by meson
    then show ‹?semantics (Neg x)›
      using Dis wf ‹closed 0 (Dis p q)› by force
  qed
next
  case (Imp p q)
  then show ?thesis
  proof (intro conjI impI)
    assume ‹x ∈ H› and ‹closed 0 x›
    then have ‹Imp p q ∈ H› and ‹closed 0 (Imp p q)›
      using Imp by simp_all
    then have ‹Neg p ∈ H ∨ q ∈ H›
      using hin unfolding hintikka_def by meson
```

```
    then show ‹?semantics x›
      using Imp wf ‹closed 0 (Imp p q)› by force
  next
    assume ‹Neg x ∈ H› and ‹closed 0 x›
    then have ‹Neg (Imp p q) ∈ H› and ‹closed 0 (Imp p q)›
      using Imp by simp_all
    then have ‹p ∈ H ∧ Neg q ∈ H›
      using hin unfolding hintikka_def by meson
    then show ‹?semantics (Neg x)›
      using Imp wf ‹closed 0 (Imp p q)› by force
  qed
next
  case (Uni P)
  then show ?thesis
  proof (intro conjI impI)
    assume ‹x ∈ H› and ‹closed 0 x›
    have ‹∀z. semantics (put e 0 z) HFun (λa ts. Pre a (tms_of_htms ts) ∈ H) P›
    proof (rule allI)
      fix z
      from ‹x ∈ H› and ‹closed 0 x›
      have ‹Uni P ∈ H› and ‹closed 0 (Uni P)›
        using Uni by simp_all
      then have *: ‹∀P t. closed_term 0 t ⟶ Uni P ∈ H ⟶ sub 0 t P ∈ H›
        using hin unfolding hintikka_def by meson
      from ‹closed 0 (Uni P)›
      have ‹closed (Suc 0) P› by simp
```

```
    have ‹(sub 0 (tm_of_htm z) P, Uni P) ∈ measure size_formulas ⟶
          (sub 0 (tm_of_htm z) P ∈ H ⟶ closed 0 (sub 0 (tm_of_htm z) P) ⟶
          ?semantics (sub 0 (tm_of_htm z) P))›
     using Uni wf by blast
    then show ‹semantics (put e 0 z) HFun (λa ts. Pre a (tms_of_htms ts) ∈ H) P›
     using * ‹Uni P ∈ H› ‹closed (Suc 0) P› by simp
  qed
  then show ‹?semantics x›
    using Uni by simp
next
  assume ‹Neg x ∈ H› and ‹closed 0 x›
  then have ‹Neg (Uni P) ∈ H›
    using Uni by simp
  then have ‹∃t. closed_term 0 t ∧ Neg (sub 0 t P) ∈ H›
    using Uni hin unfolding hintikka_def by blast
  then obtain t where *: ‹closed_term 0 t ∧ Neg (sub 0 t P) ∈ H›
    by blast
  then have ‹closed 0 (sub 0 t P)›
    using Uni ‹closed 0 x› by simp

  have ‹(sub 0 t P, Uni P) ∈ measure size_formulas ⟶
        (Neg (sub 0 t P) ∈ H ⟶ closed 0 (sub 0 t P) ⟶
        ?semantics (Neg (sub 0 t P)))›
    using Uni wf by blast
  then have ‹?semantics (Neg (sub 0 t P))›
    using Uni * ‹closed 0 (sub 0 t P)› by simp
  then have ‹∃z. ¬ semantics (put e 0 z) HFun (λa ts. Pre a (tms_of_htms ts) ∈ H) P›
```

```
    by (meson semantics.simps(1,3) substitute)
   then show ‹?semantics (Neg x)›
    using Uni by simp
  qed
next
  case (Exi P)
  then show ?thesis
  proof (intro conjI impI allI)
   assume ‹x ∈ H› and ‹closed 0 x›
   then have ‹∃t. closed_term 0 t ∧ (sub 0 t P) ∈ H›
    using Exi hin unfolding hintikka_def by blast
   then obtain t where *: ‹closed_term 0 t ∧ (sub 0 t P) ∈ H›
    by blast
   then have ‹closed 0 (sub 0 t P)›
    using Exi ‹closed 0 x› by simp

   have ‹(sub 0 t P, Exi P) ∈ measure size_formulas ⟶
        (sub 0 t P ∈ H ⟶ closed 0 (sub 0 t P) ⟶
        ?semantics (sub 0 t P))›
    using Exi wf by blast
   then have ‹?semantics (sub 0 t P)›
    using Exi * ‹closed 0 (sub 0 t P)› by simp
   then have ‹∃z. semantics (put e 0 z) HFun (λa ts. Pre a (tms_of_htms ts) ∈ H) P›
    by auto
   then show ‹?semantics x›
    using Exi by simp
  next
```

```
      assume ‹Neg x ∈ H› and ‹closed 0 x›
      have ‹∀z. ¬ semantics (put e 0 z) HFun (λa ts. Pre a (tms_of_htms ts) ∈ H) P›
      proof (rule allI)
        fix z
        from ‹Neg x ∈ H› and ‹closed 0 x›
        have ‹Neg (Exi P) ∈ H› and ‹closed 0 (Neg (Exi P))›
          using Exi by simp_all
        then have *: ‹∀P t. closed_term 0 t ⟶ Neg (Exi P) ∈ H ⟶ Neg (sub 0 t P) ∈ H›
          using hin unfolding hintikka_def by meson
        from ‹closed 0 (Neg (Exi P))›
        have ‹closed (Suc 0) P› by simp

        have ‹(sub 0 (tm_of_htm z) P, Exi P) ∈ measure size_formulas ⟶
            (Neg (sub 0 (tm_of_htm z) P) ∈ H ⟶ closed 0 (sub 0 (tm_of_htm z) P) ⟶
            ?semantics (Neg (sub 0 (tm_of_htm z) P)))›
          using Exi wf by blast
        then show ‹¬ semantics (put e 0 z) HFun (λa ts. Pre a (tms_of_htms ts) ∈ H) P›
          using * ‹Neg (Exi P) ∈ H› ‹closed (Suc 0) P› by auto
      qed
      then show ‹?semantics (Neg x)›
        using Exi by simp
    qed
  qed
qed

lemma Exi_in_extend:
  assumes ‹extend S C f n ∪ {f n} ∈ C› (is ‹?S' ∈ C›)
```

```
    and ‹Exi P = f n›
  shows ‹sub 0 (Fun (SOME k. k ∉ (⋃p ∈ ?S'. params p)) []) P ∈ extend S C f (Suc n)›
    (is ‹sub 0 ?t P ∈ _›)
proof -
  have ‹∃p. f n = Exi p›
    using ‹Exi P = f n› by metis
  then have ‹extend S C f (Suc n) = (?S' ∪ {sub 0 ?t (dest_Exi (f n))})›
    using ‹?S' ∈ C› by simp
  also have ‹… = (?S' ∪ {sub 0 ?t (dest_Exi (Exi P))})›
    using ‹Exi P = f n› by simp
  also have ‹… = (?S' ∪ {sub 0 ?t P})›
    by simp
  finally show ?thesis
    by blast
qed

lemma Neg_Uni_in_extend:
  assumes ‹extend S C f n ∪ {f n} ∈ C› (is ‹?S' ∈ C›)
    and ‹Neg (Uni P) = f n›
  shows ‹Neg (sub 0 (Fun (SOME k. k ∉ (⋃p ∈ ?S'. params p)) []) P) ∈ extend S C f (Suc n)›
    (is ‹Neg (sub 0 ?t P) ∈ _›)
proof -
  have ‹f n ≠ Exi P›
    using ‹Neg (Uni P) = f n› by auto

  have ‹∃p. f n = Neg (Uni p)›
    using ‹Neg (Uni P) = f n› by metis
```

```
  then have ‹extend S C f (Suc n) = (?S' ∪ {Neg (sub 0 ?t (dest_Uni (dest_Neg (f n))))})›
    using ‹?S' ∈ C› ‹f n ≠ Exi P› by auto
  also have ‹… = (?S' ∪ {Neg (sub 0 ?t (dest_Uni (dest_Neg (Neg (Uni P)))))})›
    using ‹Neg (Uni P) = f n› by simp
  also have ‹… = (?S' ∪ {Neg (sub 0 ?t P)})›
    by simp
  finally show ?thesis
    by blast
qed

theorem extend_hintikka:
  assumes ‹S ∈ C›
    and fin_ch: ‹finite_char C›
    and infin_p: ‹infinite (- (⋃p ∈ S. params p))›
    and surj: ‹∀y. ∃n. y = f n›
    and altc: ‹alt_consistency C›
  shows ‹hintikka (Extend S C f)› (is ‹hintikka ?H›)
  unfolding hintikka_def
proof (intro allI impI conjI)
  have ‹maximal ?H C› and ‹?H ∈ C›
    using Extend_maximal Extend_in_C assms by blast+

  { fix p ts
    show ‹¬ (Pre p ts ∈ ?H ∧ Neg (Pre p ts) ∈ ?H)›
      using ‹?H ∈ C› altc unfolding alt_consistency_def by fast }

  show ‹Falsity ∉ ?H›
```

```
  using ‹?H ∈ C› altc unfolding alt_consistency_def by blast

{ fix p q
  assume ‹Con p q ∈ ?H›
  then have ‹?H ∪ {p, q} ∈ C›
   using ‹?H ∈ C› altc unfolding alt_consistency_def by fast
  then show ‹p ∈ ?H› and ‹q ∈ ?H›
   using ‹maximal ?H C› unfolding maximal_def by fast+ }

{ fix p q
  assume ‹Neg (Dis p q) ∈ ?H›
  then have ‹?H ∪ {Neg p, Neg q} ∈ C›
   using ‹?H ∈ C› altc unfolding alt_consistency_def by fast
  then show ‹Neg p ∈ ?H› and ‹Neg q ∈ ?H›
   using ‹maximal ?H C› unfolding maximal_def by fast+ }

{ fix p q
  assume ‹Neg (Imp p q) ∈ ?H›
  then have ‹?H ∪ {p, Neg q} ∈ C›
   using ‹?H ∈ C› altc unfolding alt_consistency_def by blast
  then show ‹p ∈ ?H› and ‹Neg q ∈ ?H›
   using ‹maximal ?H C› unfolding maximal_def by fast+ }

{ fix p q
  assume ‹Dis p q ∈ ?H›
  then have ‹?H ∪ {p} ∈ C ∨ ?H ∪ {q} ∈ C›
   using ‹?H ∈ C› altc unfolding alt_consistency_def by fast
```

```
  then show ‹p ∈ ?H ∨ q ∈ ?H›
    using ‹maximal ?H C› unfolding maximal_def by fast }

{ fix p q
  assume ‹Neg (Con p q) ∈ ?H›
  then have ‹?H ∪ {Neg p} ∈ C ∨ ?H ∪ {Neg q} ∈ C›
    using ‹?H ∈ C› altc unfolding alt_consistency_def by simp
  then show ‹Neg p ∈ ?H ∨ Neg q ∈ ?H›
    using ‹maximal ?H C› unfolding maximal_def by fast }

{ fix p q
  assume ‹Imp p q ∈ ?H›
  then have ‹?H ∪ {Neg p} ∈ C ∨ ?H ∪ {q} ∈ C›
    using ‹?H ∈ C› altc unfolding alt_consistency_def by simp
  then show ‹Neg p ∈ ?H ∨ q ∈ ?H›
    using ‹maximal ?H C› unfolding maximal_def by fast }

{ fix P t
  assume ‹Uni P ∈ ?H› and ‹closed_term 0 t›
  then have ‹?H ∪ {sub 0 t P} ∈ C›
    using ‹?H ∈ C› altc unfolding alt_consistency_def by blast
  then show ‹sub 0 t P ∈ ?H›
    using ‹maximal ?H C› unfolding maximal_def by fast }

{ fix P t
  assume ‹Neg (Exi P) ∈ ?H› and ‹closed_term 0 t›
  then have ‹?H ∪ {Neg (sub 0 t P)} ∈ C›
```

```
    using ‹?H ∈ C› altc unfolding alt_consistency_def by blast
  then show ‹Neg (sub 0 t P) ∈ ?H›
    using ‹maximal ?H C› unfolding maximal_def by fast }

{ fix P
  assume ‹Exi P ∈ ?H›
  obtain n where *: ‹Exi P = f n›
    using surj by blast

  let ?t = ‹Fun (SOME k.
    k ∉ (⋃p ∈ extend S C f n ∪ {f n}. params p)) []›

  have ‹closed_term 0 ?t›
    by simp

  moreover have ‹extend S C f n ∪ {f n} ⊆ ?H›
    using ‹Exi P ∈ ?H› * Extend_def by (simp add: UN_upper)
  then have ‹extend S C f n ∪ {f n} ∈ C›
    using ‹?H ∈ C› fin_ch finite_char_closed subset_closed_def by metis
  then have ‹sub 0 ?t P ∈ ?H›
    using * Exi_in_extend Extend_def by fast
  ultimately show ‹∃t. closed_term 0 t ∧ sub 0 t P ∈ ?H›
    by blast }

{ fix P
  assume ‹Neg (Uni P) ∈ ?H›
  obtain n where *: ‹Neg (Uni P) = f n›
```

```
    using surj by blast

  let ?t = ‹Fun (SOME k. k ∉ (⋃p ∈ extend S C f n ∪ {f n}. params p)) []›

  have ‹closed_term 0 ?t›
    by simp

  moreover have ‹extend S C f n ∪ {f n} ⊆ ?H›
    using ‹Neg (Uni P) ∈ ?H› * Extend_def by (simp add: UN_upper)
  then have ‹extend S C f n ∪ {f n} ∈ C›
    using ‹?H ∈ C› fin_ch finite_char_closed subset_closed_def by metis
  then have ‹Neg (sub 0 ?t P) ∈ ?H›
    using * Neg_Uni_in_extend Extend_def by fast
  ultimately show ‹∃t. closed_term 0 t ∧ Neg (sub 0 t P) ∈ ?H›
    by blast }
qed

subsection ‹Model Existence›

lemma hintikka_Extend_S:
 assumes ‹consistency C› and ‹S ∈ C›
  and ‹infinite (- (⋃p ∈ S. params p))›
 defines ‹C' ≡ mk_finite_char (mk_alt_consistency (close C))›
 shows ‹hintikka (Extend S C' from_nat)›
proof -
 have ‹finite_char C'›
  using C'_def finite_char by blast
```

```
  moreover have ‹∀y. y = from_nat (to_nat y)›
    by simp
  then have ‹∀y. ∃n. y = from_nat n›
    by blast
  moreover have ‹alt_consistency C'›
    using C'_def ‹consistency C› finite_alt_consistency alt_consistency
      close_closed close_consistency mk_alt_consistency_closed
    by blast
  moreover have ‹S ∈ close C›
    using close_subset ‹S ∈ C› by blast
  then have ‹S ∈ mk_alt_consistency (close C)›
    using mk_alt_consistency_subset by blast
  then have ‹S ∈ C'›
    using C'_def close_closed finite_char_subset mk_alt_consistency_closed by blast
  ultimately show ?thesis
    using extend_hintikka ‹infinite (- (⋃p ∈ S. params p))› by metis
qed

theorem model_existence:
  assumes ‹infinite (- (⋃p ∈ S. params p))›
    and ‹p ∈ S› ‹closed 0 p›
    and ‹S ∈ C› ‹consistency C›
  defines ‹C' ≡ mk_finite_char (mk_alt_consistency (close C))›
  defines ‹H ≡ Extend S C' from_nat›
  shows ‹semantics e HFun (λa ts. Pre a (tms_of_htms ts) ∈ H) p›
  using assms hintikka_model hintikka_Extend_S Extend_subset by blast
```

```
subsection ‹Completeness Using Herbrand Terms›

theorem OK_consistency: ‹consistency {set z | z. ¬ (OK Falsity z)}›
  unfolding consistency_def
proof (intro conjI allI impI notI)
  fix S
  assume ‹S ∈ {set z | z. ¬ (OK Falsity z)}› (is ‹S ∈ ?C›)
  then obtain z where *: ‹S = set z› and ‹¬ (OK Falsity z)›
    by blast

  { fix i l
    assume ‹Pre i l ∈ S ∧ Neg (Pre i l) ∈ S›
    then have ‹OK (Pre i l) z› and ‹OK (Neg (Pre i l)) z›
      using Assume * by auto
    then have ‹OK Falsity z›
      using Imp_E by blast
    then show False
      using ‹¬ (OK Falsity z)› by blast }

  { assume ‹Falsity ∈ S›
    then have ‹OK Falsity z›
      using Assume * by simp
    then show False
      using ‹¬ (OK Falsity z)› by blast }

  { fix p q
    assume ‹Con p q ∈ S›
```

```
  then have ‹OK (Con p q) z›
    using Assume * by simp
  then have ‹OK p z› and ‹OK q z›
    using Con_E1 Con_E2 by blast+

  { assume ‹OK Falsity (p # q # z)›
    then have ‹OK (Neg p) (q # z)›
      using Imp_I by blast
    then have ‹OK (Neg p) z›
      using cut ‹OK q z› by blast
    then have ‹OK Falsity z›
      using Imp_E ‹OK p z› by blast
    then have False
      using ‹¬ (OK Falsity z)› by blast }
  then have ‹¬ (OK Falsity (p # q # z))›
    by blast
  moreover have ‹S ∪ {p, q} = set (p # q # z)›
    using * by simp
  ultimately show ‹S ∪ {p, q} ∈ ?C›
    by blast }

{ fix p q
  assume ‹Neg (Dis p q) ∈ S›
  then have ‹OK (Neg (Dis p q)) z›
    using Assume * by simp

  have ‹OK p (p # Neg q # z)›
```

```
  using Assume by simp
then have ‹OK (Dis p q) (p # Neg q # z)›
  using Dis_I1 by blast
moreover have ‹OK (Neg (Dis p q)) (p # Neg q # z)›
  using * ‹Neg (Dis p q) ∈ S› Assume by simp
ultimately have ‹OK Falsity (p # Neg q # z)›
  using Imp_E ‹OK (Neg (Dis p q)) (p # Neg q # z)› by blast
then have ‹OK (Neg p) (Neg q # z)›
  using Imp_I by blast

have ‹OK q (q # z)›
  using Assume by simp
then have ‹OK (Dis p q) (q # z)›
  using Dis_I2 by blast
moreover have ‹OK (Neg (Dis p q)) (q # z)›
  using * ‹Neg (Dis p q) ∈ S› Assume by simp
ultimately have ‹OK Falsity (q # z)›
  using Imp_E ‹OK (Neg (Dis p q)) (q # z)› by blast
then have ‹OK (Neg q) z›
  using Imp_I by blast

{ assume ‹OK Falsity (Neg p # Neg q # z)›
  then have ‹OK (Neg (Neg p)) (Neg q # z)›
    using Imp_I by blast
  then have ‹OK Falsity (Neg q # z)›
    using Imp_E ‹OK (Neg p) (Neg q # z)› by blast
  then have ‹OK Falsity z›
```

```
    using cut ‹OK (Neg q) z› by blast
   then have False
    using ‹¬ (OK Falsity z)› by blast }
  then have ‹¬ (OK Falsity (Neg p # Neg q # z))›
   by blast
  moreover have ‹S ∪ {Neg p, Neg q} = set (Neg p # Neg q # z)›
   using * by simp
  ultimately show ‹S ∪ {Neg p, Neg q} ∈ ?C›
   by blast }

{ fix p q
  assume ‹Neg (Imp p q) ∈ S›

  have ‹OK p (p # Neg p # Neg q # z)›
   using Assume by simp
  moreover have ‹OK (Neg p) (p # Neg p # Neg q # z)›
   using Assume by simp
  ultimately have ‹OK Falsity (p # Neg p # Neg q # z)›
   using Imp_E by blast
  then have ‹OK q (p # Neg p # Neg q # z)›
   using Falsity_E by blast
  then have ‹OK (Imp p q) (Neg p # Neg q # z)›
   using Imp_I by blast
  moreover have ‹OK (Neg (Imp p q)) (Neg p # Neg q # z)›
   using * ‹Neg (Imp p q) ∈ S› Assume by simp
  ultimately have ‹OK Falsity (Neg p # Neg q # z)›
   using Imp_E by blast
```

```
then have ‹OK p (Neg q # z)›
  using Boole by blast

have ‹OK q (p # q # z)›
  using Assume by simp
then have ‹OK (Imp p q) (q # z)›
  using Imp_I by blast
moreover have ‹OK (Neg (Imp p q)) (q # z)›
  using * ‹Neg (Imp p q) ∈ S› Assume by simp
ultimately have ‹OK Falsity (q # z)›
  using Imp_E by blast
then have ‹OK (Neg q) z›
  using Imp_I by blast

{ assume ‹OK Falsity (p # Neg q # z)›
  then have ‹OK (Neg p) (Neg q # z)›
    using Imp_I by blast
  then have ‹OK Falsity (Neg q # z)›
    using Imp_E ‹OK p (Neg q # z)› by blast
  then have ‹OK Falsity z›
    using cut ‹OK (Neg q) z› by blast
  then have False
    using ‹¬ (OK Falsity z)› by blast }
then have ‹¬ (OK Falsity (p # Neg q # z))›
  by blast
moreover have ‹{p, Neg q} ∪ S = set (p # Neg q # z)›
  using * by simp
```

```
  ultimately show ‹S ∪ {p, Neg q} ∈ ?C›
    by blast }

{ fix p q
  assume ‹Dis p q ∈ S›
  then have ‹OK (Dis p q) z›
    using * Assume by simp

  { assume ‹(∀G'. set G' = S ∪ {p} ⟶ OK Falsity G')›
      and ‹(∀G'. set G' = S ∪ {q} ⟶ OK Falsity G')›
    then have ‹OK Falsity (p # z)› and ‹OK Falsity (q # z)›
      using * by simp_all
    then have ‹OK Falsity z›
      using Dis_E ‹OK (Dis p q) z› by blast
    then have False
      using ‹¬ (OK Falsity z)› by blast }
  then show ‹S ∪ {p} ∈ ?C ∨ S ∪ {q} ∈ ?C›
    by blast }

{ fix p q
  assume ‹Neg (Con p q) ∈ S›

  let ?x = ‹Dis (Neg p) (Neg q)›

  have ‹OK p (q # p # Neg ?x # z)› and ‹OK q (q # p # Neg ?x # z)›
    using Assume by simp_all
  then have ‹OK (Con p q) (q # p # Neg ?x # z)›
```

```
  using Con_I by blast
moreover have ‹OK (Neg (Con p q)) (q # p # Neg ?x # z)›
  using * ‹Neg (Con p q) ∈ S› Assume by simp
ultimately have ‹OK Falsity (q # p # Neg ?x # z)›
  using Imp_E by blast
then have ‹OK (Neg q) (p # Neg ?x # z)›
  using Imp_I by blast
then have ‹OK ?x (p # Neg ?x # z)›
  using Dis_I2 by blast
moreover have ‹OK (Neg ?x) (p # Neg ?x # z)›
  using Assume by simp
ultimately have ‹OK Falsity (p # Neg ?x # z)›
  using Imp_E by blast
then have ‹OK (Neg p) (Neg ?x # z)›
  using Imp_I by blast
then have ‹OK ?x (Neg ?x # z)›
  using Dis_I1 by blast
then have ‹OK (Dis (Neg p) (Neg q)) z›
  using Boole' by blast

{ assume ‹(∀G'. set G' = S ∪ {Neg p} ⟶ OK Falsity G')›
    and ‹(∀G'. set G' = S ∪ {Neg q} ⟶ OK Falsity G')›
  then have ‹OK Falsity (Neg p # z )› and ‹OK Falsity (Neg q # z)›
    using * by simp_all
  then have ‹OK Falsity z›
    using Dis_E ‹OK (Dis (Neg p) (Neg q)) z› by blast
  then have False
```

```
      using ‹¬ (OK Falsity z)› by blast }
  then show ‹S ∪ {Neg p} ∈ ?C ∨ S ∪ {Neg q} ∈ ?C›
    by blast }

{ fix p q
  assume ‹Imp p q ∈ S›

  let ?x = ‹Dis (Neg p) q›

  have ‹OK p (p # Neg ?x # z)›
    using Assume by simp
  moreover have ‹OK (Imp p q) (p # Neg ?x # z)›
    using * ‹Imp p q ∈ S› Assume by simp
  ultimately have ‹OK q (p # Neg ?x # z)›
    using Imp_E by blast
  then have ‹OK ?x (p # Neg ?x # z)›
    using Dis_I2 by blast
  moreover have ‹OK (Neg ?x) (p # Neg ?x # z)›
    using Assume by simp
  ultimately have ‹OK Falsity (p # Neg ?x # z)›
    using Imp_E by blast
  then have ‹OK (Neg p) (Neg ?x # z)›
    using Imp_I by blast
  then have ‹OK ?x (Neg ?x # z)›
    using Dis_I1 by blast
  then have ‹OK (Dis (Neg p) q) z›
    using Boole' by blast
```

```
  { assume ‹(∀G'. set G' = S ∪ {Neg p} ⟶ OK Falsity G')›
      and ‹(∀G'. set G' = S ∪ {q} ⟶ OK Falsity G')›
    then have ‹OK Falsity (Neg p # z)› and ‹OK Falsity (q # z)›
      using * by simp_all
    then have ‹OK Falsity z›
      using Dis_E ‹OK (Dis (Neg p) q) z› by blast
    then have False
      using ‹¬ (OK Falsity z)› by blast }
  then show ‹S ∪ {Neg p} ∈ ?C ∨ S ∪ {q} ∈ ?C›
    by blast }

{ fix P t
  assume ‹closed_term 0 t› and ‹Uni P ∈ S›
  then have ‹OK (Uni P) z›
    using Assume * by simp
  then have ‹OK (sub 0 t P) z›
    using Uni_E by blast

  { assume ‹OK Falsity (sub 0 t P # z)›
    then have ‹OK Falsity z›
      using cut ‹OK (sub 0 t P) z› by blast
    then have False
      using ‹¬ (OK Falsity z)› by blast }
  then have ‹¬ (OK Falsity (sub 0 t P # z))›
    by blast
  moreover have ‹S ∪ {sub 0 t P} = set (sub 0 t P # z)›
```

```
    using * by simp
  ultimately show ‹S ∪ {sub 0 t P} ∈ ?C›
    by blast }

{ fix P t
  assume ‹closed_term 0 t› and ‹Neg (Exi P) ∈ S›
  then have ‹OK (Neg (Exi P)) z›
    using Assume * by simp
  then have ‹OK (sub 0 t P) (sub 0 t P # z)›
    using Assume by simp
  then have ‹OK (Exi P) (sub 0 t P # z)›
    using Exi_I by blast
  moreover have ‹OK (Neg (Exi P)) (sub 0 t P # z)›
    using * ‹Neg (Exi P) ∈ S› Assume by simp
  ultimately have ‹OK Falsity (sub 0 t P # z)›
    using Imp_E by blast
  then have ‹OK (Neg (sub 0 t P)) z›
    using Imp_I by blast

  { assume ‹OK Falsity (Neg (sub 0 t P) # z)›
    then have ‹OK Falsity z›
      using cut ‹OK (Neg (sub 0 t P)) z› by blast
    then have False
      using ‹¬ (OK Falsity z)› by blast }
  then have ‹¬ (OK Falsity (Neg (sub 0 t P) # z))›
    by blast
  moreover have ‹S ∪ {Neg (sub 0 t P)} = set (Neg (sub 0 t P) # z)›
```

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

{ fix P
  assume ‹Exi P ∈ S›
  then have ‹OK (Exi P) z›
    using * Assume by simp

  have ‹finite ((⋃p ∈ set z. params p) ∪ params P)›
    by simp
  then have ‹infinite (- ((⋃p ∈ set z. params p) ∪ params P))›
    using infinite_UNIV_listI Diff_infinite_finite finite_compl by blast
  then have ‹infinite (- ((⋃p ∈ set z. params p) ∪ params P))›
    by (simp add: Compl_eq_Diff_UNIV)
  then obtain x where **: ‹x ∈ - ((⋃p ∈ set z. params p) ∪ params P)›
    using infinite_imp_nonempty by blast

  { assume ‹OK Falsity (sub 0 (Fun x []) P # z)›
    moreover have ‹news x (P # Falsity # z)›
      using ** by (simp add: list_all_iff)
    ultimately have ‹OK Falsity z›
      using Exi_E ‹OK (Exi P) z› by fast
    then have False
      using ‹¬ (OK Falsity z)› by blast}
  then have ‹¬ (OK Falsity (sub 0 (Fun x []) P # z))›
    by blast
```

```
  moreover have ‹S ∪ {sub 0 (Fun x []) P} = set (sub 0 (Fun x []) P # z)›
    using * by simp
  ultimately show ‹∃x. S ∪ {sub 0 (Fun x []) P} ∈ ?C›
    by blast }

{ fix P
  assume ‹Neg (Uni P) ∈ S›
  then have ‹OK (Neg (Uni P)) z›
    using * Assume by simp

  have ‹finite ((⋃p ∈ set z. params p) ∪ params P)›
    by simp
  then have ‹infinite (- ((⋃p ∈ set z. params p) ∪ params P))›
    using infinite_UNIV_listI Diff_infinite_finite finite_compl by blast
  then have ‹infinite (- ((⋃p ∈ set z. params p) ∪ params P))›
    by (simp add: Compl_eq_Diff_UNIV)
  then obtain x where **: ‹x ∈ - ((⋃p ∈ set z. params p) ∪ params P)›
    using infinite_imp_nonempty by blast

  let ?x = ‹Neg (Exi (Neg P))›

  have ‹OK (Neg (sub 0 (Fun x []) P)) (Neg (sub 0 (Fun x []) P) # ?x # z)›
    using Assume by simp
  then have ‹OK (Exi (Neg P)) (Neg (sub 0 (Fun x []) P) # ?x # z)›
    using Exi_I by simp
  moreover have ‹OK ?x (Neg (sub 0 (Fun x []) P) # ?x # z)›
    using Assume by simp
```

```
ultimately have ‹OK Falsity (Neg (sub 0 (Fun x []) P) # ?x # z)›
  using Imp_E by blast
then have ‹OK (sub 0 (Fun x []) P) (?x # z)›
  using Boole by blast
moreover have ‹news x (P # ?x # z)›
  using ** by (simp add: list_all_iff)
ultimately have ‹OK (Uni P) (?x # z)›
  using Uni_I by fast
moreover have ‹OK (Neg (Uni P)) (?x # z)›
  using * ‹Neg (Uni P) ∈ S› Assume by simp
ultimately have ‹OK Falsity (?x # z)›
  using Imp_E by blast
then have ‹OK (Exi (Neg P)) z›
  using Boole by blast

{ assume ‹OK Falsity (Neg (sub 0 (Fun x []) P) # z)›
  then have ‹OK (sub 0 (Fun x []) P) z›
    using Boole by blast
  moreover have ‹news x (P # z)›
    using ** by (simp add: list_all_iff)
  ultimately have ‹OK (Uni P) z›
    using Uni_I by blast
  then have ‹OK Falsity z›
    using Imp_E ‹OK (Neg (Uni P)) z› by blast
  then have False
    using ‹¬ (OK Falsity z)› by blast }
then have ‹¬ (OK Falsity (Neg (sub 0 (Fun x []) P) # z))›
```

```
    by blast
  moreover have ‹S ∪ {Neg (sub 0 (Fun x []) P)} = set (Neg (sub 0 (Fun x []) P) # z)›
    using * by simp
  ultimately show ‹∃x. S ∪ {Neg (sub 0 (Fun x []) P)} ∈ ?C›
    by blast }
qed

theorem natded_complete:
  assumes ‹closed 0 p› and ‹list_all (closed 0) z›
    and mod: ‹∀(e :: _ ⇒ htm) f g. list_all (semantics e f g) z ⟶ semantics e f g p›
  shows ‹OK p z›
proof (rule Boole, rule ccontr)
  fix e
  assume ‹¬ (OK Falsity (Neg p # z))›

  let ?S = ‹set (Neg p # z)›
  let ?C = ‹{set z | z. ¬ (OK Falsity z)}›
  let ?C' = ‹mk_finite_char (mk_alt_consistency (close ?C))›
  let ?H = ‹Extend ?S ?C' from_nat›
  let ?f = HFun
  let ?g = ‹λi l. Pre i (tms_of_htms l) ∈ ?H›

  { fix x
    assume ‹x ∈ ?S›
    moreover have ‹closed 0 x›
      using ‹closed 0 p› ‹list_all (closed 0) z› ‹x ∈ ?S›
      by (auto simp: list_all_iff)
```

```
    moreover have ‹?S ∈ ?C›
      using ‹¬ (OK Falsity (Neg p # z))› by blast
    moreover have ‹consistency ?C›
      using OK_consistency by blast
    moreover have ‹infinite (- (⋃p ∈ ?S. params p))›
      by (simp add: Compl_eq_Diff_UNIV infinite_UNIV_listI)
    ultimately have ‹semantics e ?f ?g x›
      using model_existence by simp }
  then have ‹semantics e ?f ?g (Neg p)›
    and ‹list_all (semantics e ?f ?g) z›
    unfolding list_all_def by fastforce+
  then have ‹semantics e ?f ?g p›
    using mod by blast
  then show False
    using ‹semantics e ?f ?g (Neg p)› by simp
qed

subsection ‹Löwenheim-Skolem›

theorem sat_consistency:
  ‹consistency {S. infinite (- (⋃p ∈ S. params p)) ∧ (∃f. ∀p ∈ S. semantics e f g p)}›
  (is ‹consistency ?C›)
  unfolding consistency_def
proof (intro allI impI conjI)
  fix S :: ‹fm set›
  assume ‹S ∈ ?C›
  then have inf_params: ‹infinite (- (⋃p ∈ S. params p))›
```

```
  and ‹∃f. ∀p ∈ S. semantics e f g p›
  by blast+
then obtain f where *: ‹∀x ∈ S. semantics e f g x› by blast

{ fix p ts
  show ‹¬ (Pre p ts ∈ S ∧ Neg (Pre p ts) ∈ S)›
  proof
    assume ‹Pre p ts ∈ S ∧ Neg (Pre p ts) ∈ S›
    then have ‹semantics e f g (Pre p ts) ∧ semantics e f g (Neg (Pre p ts))›
      using * by blast
    then show False by auto
  qed }

show ‹Falsity ∉ S›
  using * by fastforce

{ fix p q
  assume ‹Con p q ∈ S›
  then have ‹∀x ∈ S ∪ {p, q}. semantics e f g x›
    using * by auto
  moreover have ‹infinite (- (⋃p ∈ S ∪ {p, q}. params p))›
    using inf_params by (simp add: set_inter_compl_diff)
  ultimately show ‹S ∪ {p, q} ∈ ?C›
    by blast }

{ fix p q
  assume ‹Neg (Dis p q) ∈ S›
```

```
  then have ‹∀x ∈ S ∪ {Neg p, Neg q}. semantics e f g x›
   using * by auto
  moreover have ‹infinite (- (⋃p ∈ S ∪ {Neg p, Neg q}. params p))›
   using inf_params by (simp add: set_inter_compl_diff)
  ultimately show ‹S ∪ {Neg p, Neg q} ∈ ?C›
   by blast }

{ fix p q
  assume ‹Neg (Imp p q) ∈ S›
  then have ‹∀x ∈ S ∪ {p, Neg q}. semantics e f g x›
   using * by auto
  moreover have ‹infinite (- (⋃p ∈ S ∪ {p, Neg q}. params p))›
   using inf_params by (simp add: set_inter_compl_diff)
  ultimately show ‹S ∪ {p, Neg q} ∈ ?C›
   by blast }

{ fix p q
  assume ‹Dis p q ∈ S›
  then have ‹(∀x ∈ S ∪ {p}. semantics e f g x) ∨
           (∀x ∈ S ∪ {q}. semantics e f g x)›
   using * by auto
  moreover have ‹infinite (- (⋃p ∈ S ∪ {p}. params p))›
   and ‹infinite (- (⋃p ∈ S ∪ {q}. params p))›
   using inf_params by (simp_all add: set_inter_compl_diff)
  ultimately show ‹S ∪ {p} ∈ ?C ∨ S ∪ {q} ∈ ?C›
   by blast }
```

```
{ fix p q
  assume ‹Neg (Con p q) ∈ S›
  then have ‹(∀x ∈ S ∪ {Neg p}. semantics e f g x) ∨
           (∀x ∈ S ∪ {Neg q}. semantics e f g x)›
   using * by auto
  moreover have ‹infinite (- (⋃p ∈ S ∪ {Neg p}. params p))›
   and ‹infinite (- (⋃p ∈ S ∪ {Neg q}. params p))›
   using inf_params by (simp_all add: set_inter_compl_diff)
  ultimately show ‹S ∪ {Neg p} ∈ ?C ∨ S ∪ {Neg q} ∈ ?C›
   by blast }

{ fix p q
  assume ‹Imp p q ∈ S›
  then have ‹(∀x ∈ S ∪ {Neg p}. semantics e f g x) ∨
           (∀x ∈ S ∪ {q}. semantics e f g x)›
   using * by auto
  moreover have ‹infinite (- (⋃p ∈ S ∪ {Neg p}. params p))›
   and ‹infinite (- (⋃p ∈ S ∪ {q}. params p))›
   using inf_params by (simp_all add: set_inter_compl_diff)
  ultimately show ‹S ∪ {Neg p} ∈ ?C ∨ S ∪ {q} ∈ ?C›
   by blast }

{ fix P t
  assume ‹Uni P ∈ S›
  then have ‹∀x ∈ S ∪ {sub 0 t P}. semantics e f g x›
   using * by auto
  moreover have ‹infinite (- (⋃p ∈ S ∪ {sub 0 t P}. params p))›
```

```
    using inf_params by (simp add: set_inter_compl_diff)
  ultimately show ‹S ∪ {sub 0 t P} ∈ ?C›
    by blast }

{ fix P t
  assume ‹Neg (Exi P) ∈ S›
  then have ‹∀x ∈ S ∪ {Neg (sub 0 t P)}. semantics e f g x›
    using * by auto
  moreover have ‹infinite (- (⋃p ∈ S ∪ {Neg (sub 0 t P)}. params p))›
    using inf_params by (simp add: set_inter_compl_diff)
  ultimately show ‹S ∪ {Neg (sub 0 t P)} ∈ ?C›
    by blast }

{ fix P
  assume ‹Exi P ∈ S›
  then obtain y where ‹semantics (put e 0 y) f g P›
    using * by fastforce
  moreover obtain x where **: ‹x ∈ - (⋃p ∈ S. params p)›
    using inf_params infinite_imp_nonempty by blast
  then have ‹x ∉ params P›
    using ‹Exi P ∈ S› by auto
  moreover have ‹∀p ∈ S. semantics e (f(x := λ_. y)) g p›
    using * ** by simp
  ultimately have ‹∀p ∈ S ∪ {sub 0 (Fun x []) P}.
      semantics e (f(x := λ_. y)) g p›
    by simp
  moreover have
```

```
    ‹infinite (- (⋃p ∈ S ∪ {sub 0 (Fun x []) P}. params p))›
    using inf_params by (simp add: set_inter_compl_diff)
  ultimately show ‹∃x. S ∪ {sub 0 (Fun x []) P} ∈ ?C›
    by blast }

 { fix P
  assume ‹Neg (Uni P) ∈ S›
  then obtain y where ‹¬ semantics (put e 0 y) f g P›
    using * by fastforce
  moreover obtain x where **: ‹x ∈ - (⋃p ∈ S. params p)›
    using inf_params infinite_imp_nonempty by blast
  then have ‹x ∉ params P›
    using ‹Neg (Uni P) ∈ S› by auto
  moreover have ‹∀p ∈ S. semantics e (f(x := λ_. y)) g p›
    using * ** by simp
  ultimately have ‹∀p ∈ S ∪ {Neg (sub 0 (Fun x []) P)}. semantics e (f(x := λ_. y)) g p›
    by simp
  moreover have ‹infinite (- (⋃p ∈ S ∪ {Neg (sub 0 (Fun x []) P)}. params p))›
    using inf_params by (simp add: set_inter_compl_diff)
  ultimately show ‹∃x. S ∪ {Neg (sub 0 (Fun x []) P)} ∈ ?C›
    by blast }
qed

primrec double :: ‹'a list ⇒ 'a list› where
 ‹double [] = []› |
 ‹double (x#xs) = x # x # double xs›
```

```
fun undouble :: ‹'a list ⇒ 'a list› where
  ‹undouble [] = []› |
  ‹undouble [x] = [x]› |
  ‹undouble (x#_#xs) = x # undouble xs›

lemma undouble_double_id [simp]: ‹undouble (double xs) = xs›
  by (induct xs) simp_all

lemma infinite_double_Cons: ‹infinite (range (λxs. a # double xs))›
  using undouble_double_id infinite_UNIV_listI
  by (metis (mono_tags, lifting) finite_imageD inj_onI list.inject)

lemma double_Cons_neq: ‹a # (double xs) ≠ double ys›
proof -
  have ‹odd (length (a # double xs))›
    by (induct xs) simp_all
  moreover have ‹even (length (double ys))›
    by (induct ys) simp_all
  ultimately show ?thesis
    by fastforce
qed

lemma doublep_infinite_params: ‹infinite (- (⋃p ∈ psubst double ` S. params p))›
proof (rule infinite_super)
  fix a
  show ‹infinite (range (λxs :: id. a # double xs))›
    using infinite_double_Cons by metis
```

```
next
  fix a
  show ‹range (λxs. a # double xs) ⊆ - (⋃p ∈ psubst double ` S. params p)›
    using double_Cons_neq by fastforce
qed

theorem loewenheim_skolem:
  assumes ‹∀p ∈ S. semantics e f g p› ‹∀p ∈ S. closed 0 p›
  defines ‹C ≡ {S. infinite (- (⋃p ∈ S. params p)) ∧ (∃f. ∀p ∈ S. semantics e f g p)}›
  defines ‹C' ≡ mk_finite_char (mk_alt_consistency (close C))›
  defines ‹H ≡ Extend (psubst double ` S) C' from_nat›
  shows ‹∀p ∈ S. semantics e' (λxs. HFun (double xs)) (λi l. Pre i (tms_of_htms l) ∈ H) p›
proof (intro ballI impI)
  fix p
  assume ‹p ∈ S›

  let ?g = ‹λi l. Pre i (tms_of_htms l) ∈ H›

  have ‹∀p ∈ psubst double ` S. semantics e (λxs. f (undouble xs)) g p›
    using ‹∀p ∈ S. semantics e f g p› by (simp add: psubst_semantics)
  then have ‹psubst double ` S ∈ C›
    using C_def doublep_infinite_params by blast
  moreover have ‹psubst double p ∈ psubst double ` S›
    using ‹p ∈ S› by blast
  moreover have ‹closed 0 (psubst double p)›
    using ‹∀p ∈ S. closed 0 p› ‹p ∈ S› by simp
  moreover have ‹consistency C›
```

```
    using C_def sat_consistency by blast
  ultimately have ‹semantics e' HFun ?g (psubst double p)›
    using C_def C'_def H_def model_existence by simp
  then show ‹semantics e' (λxs. HFun (double xs)) ?g p›
    using psubst_semantics by blast
qed

subsection ‹Countable Variants›

lemma infinity:
  assumes inj: ‹∀n :: nat. undiago (diago n) = n›
  assumes all_tree: ‹∀n :: nat. (diago n) ∈ tree›
  shows ‹infinite tree›
proof -
  from inj all_tree have ‹∀n. n = undiago (diago n) ∧ (diago n) ∈ tree›
    by simp
  then have ‹undiago ` tree = (UNIV :: nat set)›
    by auto
  then have ‹infinite tree›
    by (metis finite_imageI infinite_UNIV_nat)
  then show ?thesis
    by simp
qed

definition nat_of_string :: ‹string ⇒ nat› where
  ‹nat_of_string ≡ (SOME f. bij f)›
```

```
definition string_of_nat :: ‹nat ⇒ string› where
  ‹string_of_nat ≡ inv nat_of_string›

lemma nat_of_string_string_of_nat [simp]: ‹nat_of_string (string_of_nat n) = n›
  using Schroeder_Bernstein bij_is_surj infinite_UNIV_listI infinite_iff_countable_subset
    nat_of_string_def someI_ex string_of_nat_def surj_f_inv_f top_greatest inj_undiag_string
  by (metis (mono_tags, lifting))

lemma string_of_nat_nat_of_string [simp]: ‹string_of_nat (nat_of_string n) = n›
  using Schroeder_Bernstein UNIV_I bij_is_inj infinite_UNIV_listI infinite_iff_countable_subset
    inv_into_f_f nat_of_string_def someI_ex string_of_nat_def top_greatest inj_undiag_string
  by (metis (mono_tags, lifting))

lemma infinite_htms: ‹infinite (UNIV :: htm set)›
proof -
  let ?diago = ‹λn. HFun (string_of_nat n) []›
  let ?undiago = ‹λa. nat_of_string (case a of HFun f ts ⇒ f)›
  show ?thesis
    using infinity[of ?undiago ?diago UNIV] by simp
qed

definition e_conv :: ‹('a ⇒ 'b) ⇒ (nat ⇒ 'a) ⇒ (nat ⇒ 'b)› where
  ‹e_conv b_of_a e ≡ (λn. b_of_a (e n))›

definition f_conv ::
  ‹('a ⇒ 'b) ⇒ (id ⇒ 'a list ⇒ 'a) ⇒ (id ⇒ 'b list ⇒ 'b)› where
  ‹f_conv b_of_a f ≡ (λa ts. b_of_a (f a (map (inv b_of_a) ts)))›
```

```
definition g_conv ::
  ‹('a ⇒ 'b) ⇒ (id ⇒ 'a list ⇒ bool) ⇒ (id ⇒ 'b list ⇒ bool)› where
  ‹g_conv b_of_a g ≡ (λa ts. g a (map (inv b_of_a) ts))›

lemma semantics_bij':
  assumes ‹bij (b_of_a :: 'a ⇒ 'b)›
  shows
    ‹semantics_term (e_conv b_of_a e) (f_conv b_of_a f) p = b_of_a (semantics_term e f p)›
    ‹semantics_list (e_conv b_of_a e) (f_conv b_of_a f) l = map b_of_a (semantics_list e f l)›
  unfolding e_conv_def f_conv_def using assms
  by (induct p and l rule: semantics_term.induct semantics_list.induct) (simp_all add: bij_is_inj)

lemma put_e_conv: ‹(put (e_conv b_of_a e) m (b_of_a x)) = e_conv b_of_a (put e m x)›
  unfolding e_conv_def by auto

lemma semantics_bij:
  assumes ‹bij (b_of_a :: 'a ⇒ 'b)›
  shows ‹semantics e f g p = semantics (e_conv b_of_a e) (f_conv b_of_a f) (g_conv b_of_a g) p›
proof (induct p arbitrary: e f g)
  case (Pre a l)
  then show ?case
    unfolding g_conv_def using assms
    by (simp add: semantics_bij' bij_is_inj)
next
  case (Exi p)
  let ?e = ‹e_conv b_of_a e›
```

```
   and ?f = ‹f_conv b_of_a f›
   and ?g = ‹g_conv b_of_a g›

  have ‹(∃x'. semantics (put ?e 0 x') ?f ?g p) = (∃x. semantics (put ?e 0 (b_of_a x)) ?f ?g p)›
   using assms by (metis bij_pointE)
  also have ‹… = (∃x. semantics (e_conv b_of_a (put e 0 x)) ?f ?g p)›
   using put_e_conv by metis
  finally show ?case
   using Exi by simp
next
  case (Uni p)
  have ‹(∀x. semantics (put (e_conv b_of_a e) 0 x) (f_conv b_of_a f) (g_conv b_of_a g) p) =
      (∀x. semantics (put (e_conv b_of_a e) 0 (b_of_a x)) (f_conv b_of_a f) (g_conv b_of_a g) p)›
   using assms by (metis bij_pointE)
  also have ‹… = (∀x. semantics (e_conv b_of_a (put e 0 x)) (f_conv b_of_a f) (g_conv b_of_a g) p)›
   using put_e_conv by metis
  finally show ?case
   using Uni by simp
qed simp_all

fun
  hterm_of_btree :: ‹btree ⇒ htm› and
  hterm_list_of_btree :: ‹btree ⇒ htm list› where
  ‹hterm_of_btree (Leaf _) = undefined›
| ‹hterm_of_btree (Branch (Leaf m) t) =
    HFun (diag_string m) (hterm_list_of_btree t)›
| ‹hterm_list_of_btree (Leaf m) = []›
```

```
| ‹hterm_list_of_btree (Branch t1 t2) =
   hterm_of_btree t1 # hterm_list_of_btree t2›
| ‹hterm_of_btree (Branch (Branch _ _) _) = undefined›

primrec
 btree_of_hterm :: ‹htm ⇒ btree› and
 btree_of_hterm_list :: ‹htm list ⇒ btree› where
 ‹btree_of_hterm (HFun m ts) = Branch (Leaf (undiag_string m)) (btree_of_hterm_list ts)›
| ‹btree_of_hterm_list [] = Leaf 0›
| ‹btree_of_hterm_list (t # ts) = Branch (btree_of_hterm t) (btree_of_hterm_list ts)›

theorem hterm_btree:
 shows ‹hterm_of_btree (btree_of_hterm t) = t›
  and ‹hterm_list_of_btree (btree_of_hterm_list ts) = ts›
 by (induct t and ts rule: btree_of_hterm.induct btree_of_hterm_list.induct) simp_all

definition diag_hterm :: ‹nat ⇒ htm› where
 ‹diag_hterm n = hterm_of_btree (diag_btree n)›

definition undiag_hterm :: ‹htm ⇒ nat› where
 ‹undiag_hterm t = undiag_btree (btree_of_hterm t)›

theorem diag_undiag_hterm [simp]: ‹diag_hterm (undiag_hterm t) = t›
 by (simp add: diag_hterm_def undiag_hterm_def hterm_btree)

lemma htm: ‹∃f :: htm ⇒ nat. inj f›
 unfolding inj_def using diag_undiag_hterm by metis
```

```
definition denumerable :: ‹'a set ⇒ bool›
  where ‹denumerable S ≡ (∃f :: 'a ⇒ nat. inj_on f S) ∧ (∃f :: nat ⇒ 'a. range f ⊆ S ∧ inj f)›

lemma denumerable_bij: ‹denumerable S ⟷ (∃f. bij_betw f (UNIV :: nat set) S)›
  unfolding denumerable_def
  using Schroeder_Bernstein UNIV_I bij_betw_def bij_betw_inv subsetI by metis

hide_fact denumerable_def

lemma denumerable_htm: ‹denumerable (UNIV :: htm set)›
  using infinite_htms htm denumerable_bij Schroeder_Bernstein infinite_iff_countable_subset
    top_greatest by metis

abbreviation ‹sentence ≡ closed 0›

lemma sentence_completeness':
  assumes ‹∀(e :: _ ⇒ 'a) f g. list_all (semantics e f g) z ⟶ semantics e f g p›
    and ‹sentence p›
    and ‹list_all sentence z›
    and ‹denumerable (UNIV :: 'a set)›
  shows ‹OK p z›
proof -
  have ‹∀(e :: _ ⇒ htm) f g. list_all (semantics e f g) z ⟶ semantics e f g p›
  proof (intro allI)
    fix e :: ‹nat ⇒ htm›
      and f :: ‹id ⇒ htm list ⇒ htm›
```

```
    and g :: ‹id ⇒ htm list ⇒ bool›

  obtain a_of_htm :: ‹htm ⇒ 'a› where p_a_of_hterm: ‹bij a_of_htm›
    using assms(4) infinite_htms htm denumerable_bij
      Schroeder_Bernstein bij_comp infinite_iff_countable_subset top_greatest by metis

  let ?e = ‹e_conv a_of_htm e›
  let ?f = ‹f_conv a_of_htm f›
  let ?g = ‹g_conv a_of_htm g›

  have ‹list_all (semantics ?e ?f ?g) z ⟶ semantics ?e ?f ?g p›
    using assms(1) by blast
  then show ‹list_all (semantics e f g) z ⟶ semantics e f g p›
    using p_a_of_hterm semantics_bij by (metis list.pred_cong)
 qed
 then show ?thesis
  using assms(2) assms(3) natded_complete by blast
qed

theorem sentence_completeness:
 assumes ‹∀(e :: _ ⇒ 'a) f g. semantics e f g p›
  and ‹sentence p›
  and ‹denumerable (UNIV :: 'a set)›
 shows ‹OK p []›
 using assms by (simp add: sentence_completeness')

corollary ‹∀(e :: _ ⇒ nat) f g. semantics e f g p ⟹ sentence p ⟹ OK p []›
```

```
  using sentence_completeness denumerable_bij by blast

section ‹Open Formulas›

subsection ‹Renaming›

lemma new_psubst_image':
  ‹new_term c t ⟹ d ∉ image f (params_term t) ⟹ new_term d (psubst_term (f(c := d)) t)›
  ‹new_list c l ⟹ d ∉ image f (params_list l) ⟹ new_list d (psubst_list (f(c := d)) l)›
  by (induct t and l rule: new_term.induct new_list.induct) auto

lemma new_psubst_image: ‹new c p ⟹ d ∉ image f (params p) ⟹ new d (psubst (f(c := d)) p)›
  using new_psubst_image' by (induct p) auto

lemma news_psubst: ‹news c z ⟹ d ∉ image f (⋃p ∈ set z. params p) ⟹
    news d (map (psubst (f(c := d))) z)›
  using new_psubst_image by (induct z) auto

lemma member_psubst: ‹member p z ⟹ member (psubst f p) (map (psubst f) z)›
  by (induct z) auto

lemma OK_psubst: ‹OK p z ⟹ OK (psubst f p) (map (psubst f) z)›
proof (induct p z arbitrary: f rule: OK.induct)
  case (Assume p z)
  then show ?case
    using OK.Assume member_psubst by blast
next
```

```
case (Exi_E p z q c)
let ?params = ‹params p ∪ params q ∪ (⋃p ∈ set z. params p)›

have ‹finite ?params›
  by simp
then obtain fresh where *: ‹fresh ∉ ?params ∪ {c} ∪ image f ?params›
  using ex_new_if_finite
  by (metis finite.emptyI finite.insertI finite_UnI finite_imageI infinite_UNIV_listI)

let ?f = ‹f(c := fresh)›

have ‹news c (p # q # z)›
  using Exi_E by blast
then have ‹new fresh (psubst ?f p)› ‹new fresh (psubst ?f q)› ‹news fresh (map (psubst ?f) z)›
  using * new_psubst_image news_psubst by (fastforce simp add: image_Un)+
have ‹OK (psubst ?f (Exi p)) (map (psubst ?f) z)›
  using Exi_E by blast
then have ‹OK (Exi (psubst ?f p)) (map (psubst ?f) z)›
  by simp
moreover have ‹OK (psubst ?f q) (map (psubst ?f) (sub 0 (Fun c []) p # z))›
  using Exi_E by blast
then have ‹OK (psubst ?f q) (sub 0 (Fun fresh []) (psubst ?f p) # map (psubst ?f) z)›
  by simp
moreover have ‹news fresh (map (psubst ?f) (p # q # z))›
  using ‹new fresh (psubst ?f p)› ‹new fresh (psubst ?f q)› ‹news fresh (map (psubst ?f) z)›
  by simp
then have ‹news fresh (psubst ?f p # psubst ?f q # map (psubst ?f) z)›
```

```
    by simp
  ultimately have ‹OK (psubst ?f q) (map (psubst ?f) z)›
    using OK.Exi_E by blast
  moreover have ‹list_all (new c) z›
    using Exi_E by simp
  then have ‹map (psubst ?f) z = map (psubst f) z›
    unfolding list_all_iff by simp
  ultimately show ?case
    using Exi_E by simp
next
  case (Uni_I c p z)
  let ?params = ‹params p ∪(⋃p ∈ set z. params p)›

  have ‹finite ?params›
    by simp
  then obtain fresh where *: ‹fresh ∉ ?params ∪ {c} ∪ image f ?params›
    using ex_new_if_finite
    by (metis finite.emptyI finite.insertI finite_UnI finite_imageI infinite_UNIV_listI)

  let ?f = ‹f(c := fresh)›

  have ‹news c (p # z)›
    using Uni_I by blast
  then have ‹new fresh (psubst ?f p)› ‹news fresh (map (psubst ?f) z)›
    using * new_psubst_image news_psubst by (fastforce simp add: image_Un)+
  then have ‹map (psubst ?f) z = map (psubst f) z›
    using Uni_I allnew new_params
```

```
    by (metis (mono_tags, lifting) Ball_set map_eq_conv news.simps(2) psubst_upd)

  have ‹OK (psubst ?f (sub 0 (Fun c []) p)) (map (psubst ?f) z)›
    using Uni_I by blast
  then have ‹OK (sub 0 (Fun fresh []) (psubst ?f p)) (map (psubst ?f) z)›
    by simp
  moreover have ‹news fresh (map (psubst ?f) (p # z))›
    using ‹new fresh (psubst ?f p)› ‹news fresh (map (psubst ?f) z)›
    by simp
  then have ‹news fresh (psubst ?f p # map (psubst ?f) z)›
    by simp
  ultimately have ‹OK (Uni (psubst ?f p)) (map (psubst ?f) z)›
    using OK.Uni_I by blast
  then show ?case
    using Uni_I ‹map (psubst ?f) z = map (psubst f) z› by simp
qed (auto intro: OK.intros)

subsection ‹Substitution for Constants›

primrec
  subc_term :: ‹id ⇒ tm ⇒ tm ⇒ tm› and
  subc_list :: ‹id ⇒ tm ⇒ tm list ⇒ tm list› where
  ‹subc_term c s (Var n) = Var n› |
  ‹subc_term c s (Fun i l) = (if i = c then s else Fun i (subc_list c s l))› |
  ‹subc_list c s [] = []› |
  ‹subc_list c s (t # l) = subc_term c s t # subc_list c s l›
```

```
primrec subc :: ‹id ⇒ tm ⇒ fm ⇒ fm› where
  ‹subc c s Falsity = Falsity› |
  ‹subc c s (Pre i l) = Pre i (subc_list c s l)› |
  ‹subc c s (Imp p q) = Imp (subc c s p) (subc c s q)› |
  ‹subc c s (Dis p q) = Dis (subc c s p) (subc c s q)› |
  ‹subc c s (Con p q) = Con (subc c s p) (subc c s q)› |
  ‹subc c s (Exi p) = Exi (subc c (inc_term s) p)› |
  ‹subc c s (Uni p) = Uni (subc c (inc_term s) p)›

primrec subcs :: ‹id ⇒ tm ⇒ fm list ⇒ fm list› where
  ‹subcs c s [] = []› |
  ‹subcs c s (p # z) = subc c s p # subcs c s z›

lemma sub_0_inc:
  ‹sub_term 0 s (inc_term t) = t›
  ‹sub_list 0 s (inc_list l) = l›
  by (induct t and l rule: sub_term.induct sub_list.induct) simp_all

lemma sub_new':
  ‹new_term c s ⟹ new_term c t ⟹ new_term c (sub_term m s t)›
  ‹new_term c s ⟹ new_list c l ⟹ new_list c (sub_list m s l)›
  by (induct t and l rule: sub_term.induct sub_list.induct) simp_all

lemma sub_new: ‹new_term c s ⟹ new c p ⟹ new c (sub m s p)›
  using sub_new' by (induct p arbitrary: m s) simp_all

lemma sub_new_all:
```

```
  assumes ‹a ∉ set cs› ‹list_all (λc. new c p) cs›
  shows ‹∀c ∈ set cs. new c (sub m (Fun a []) p)›
  using assms sub_new by (induct cs) auto

lemma subc_new' [simp]:
  ‹new_term c t ⟹ subc_term c s t = t›
  ‹new_list c l ⟹ subc_list c s l = l›
  by (induct t and l rule: new_term.induct new_list.induct) auto

lemma subc_new [simp]: ‹new c p ⟹ subc c s p = p›
  by (induct p arbitrary: s) simp_all

lemma subcs_news [simp]: ‹news c z ⟹ subcs c s z = z›
  by (induct z) simp_all

lemma subc_psubst' [simp]:
  ‹(∀x ∈ params_term t. x ≠ c ⟶ f x ≠ f c) ⟹
    psubst_term f (subc_term c s t) = subc_term (f c) (psubst_term f s) (psubst_term f t)›
  ‹(∀x ∈ params_list l. x ≠ c ⟶ f x ≠ f c) ⟹
    psubst_list f (subc_list c s l) = subc_list (f c) (psubst_term f s) (psubst_list f l)›
  by (induct t and l rule: psubst_term.induct psubst_list.induct) simp_all

lemma subc_psubst [simp]: ‹(∀x ∈ params p. x ≠ c ⟶ f x ≠ f c) ⟹
    psubst f (subc c s p) = subc (f c) (psubst_term f s) (psubst f p)›
  by (induct p arbitrary: s) simp_all

lemma subcs_psubst [simp]: ‹(∀x ∈ (⋃p ∈ set z. params p). x ≠ c ⟶ f x ≠ f c) ⟹
```

```
  map (psubst f) (subcs c s z) = subcs (f c) (psubst_term f s) (map (psubst f) z)›
 by (induct z) simp_all

lemma new_inc':
 ‹new_term c t ⟹ new_term c (inc_term t)›
 ‹new_list c l ⟹ new_list c (inc_list l)›
 by (induct t and l rule: new_term.induct new_list.induct) simp_all

lemma new_subc':
 ‹new_term d s ⟹ new_term d t ⟹ new_term d (subc_term c s t)›
 ‹new_term d s ⟹ new_list d l ⟹ new_list d (subc_list c s l)›
 by (induct t and l rule: sub_term.induct sub_list.induct) simp_all

lemma new_subc: ‹new_term d s ⟹ new d p ⟹ new d (subc c s p)›
 using new_subc' by (induct p arbitrary: s) simp_all

lemma news_subcs: ‹new_term d s ⟹ news d z ⟹ news d (subcs c s z)›
 using new_subc by (induct z) simp_all

lemma psubst_new_free':
 ‹c ≠ n ⟹ new_term n (psubst_term (id(n := c)) t)›
 ‹c ≠ n ⟹ new_list n (psubst_list (id(n := c)) l)›
 by (induct t and l rule: params_term.induct params_list.induct) simp_all

lemma psubst_new_free: ‹c ≠ n ⟹ new n (psubst (id(n := c)) p)›
 using psubst_new_free' by (induct p) simp_all
```

```
lemma map_psubst_new_free: ‹c ≠ n ⟹ news n (map (psubst (id(n := c))) z)›
  using psubst_new_free by (induct z) simp_all

lemma psubst_new_away' [simp]:
  ‹new_term fresh t ⟹ psubst_term (id(fresh := c)) (psubst_term (id(c := fresh)) t) = t›
  ‹new_list fresh l ⟹ psubst_list (id(fresh := c)) (psubst_list (id(c := fresh)) l) = l›
  by (induct t and l rule: psubst_term.induct psubst_list.induct) auto

lemma psubst_new_away [simp]: ‹new fresh p ⟹
    psubst (id(fresh := c)) (psubst (id(c := fresh)) p) = p›
  by (induct p) simp_all

lemma map_psubst_new_away:
  ‹news fresh z ⟹ map (psubst (id(fresh := c))) (map (psubst (id(c := fresh))) z) = z›
  by (induct z) simp_all

lemma psubst_new':
  ‹new_term c t ⟹ psubst_term (id(c := x)) t = t›
  ‹new_list c l ⟹ psubst_list (id(c := x)) l = l›
  by (induct t and l rule: psubst_term.induct psubst_list.induct) auto

lemma psubst_new: ‹new c p ⟹ psubst (id(c := x)) p = p›
  using psubst_new' by (induct p) simp_all

lemma map_psubst_new: ‹news c z ⟹ map (psubst (id(c := x))) z = z›
  using psubst_new by (induct z) simp_all
```

```
lemma inc_sub':
  ‹inc_term (sub_term m u t) = sub_term (m + 1) (inc_term u) (inc_term t)›
  ‹inc_list (sub_list m u l) = sub_list (m + 1) (inc_term u) (inc_list l)›
  by (induct t and l rule: sub_term.induct sub_list.induct) simp_all

lemma new_subc_same':
  ‹new_term c s ⟹ new_term c (subc_term c s t)›
  ‹new_term c s ⟹ new_list c (subc_list c s l)›
  by (induct t and l rule: subc_term.induct subc_list.induct) simp_all

lemma new_subc_same: ‹new_term c s ⟹ new c (subc c s p)›
  using new_subc_same' by (induct p arbitrary: s) simp_all

lemma inc_subc:
  ‹inc_term (subc_term c s t) = subc_term c (inc_term s) (inc_term t)›
  ‹inc_list (subc_list c s l) = subc_list c (inc_term s) (inc_list l)›
  by (induct t and l rule: inc_term.induct inc_list.induct) simp_all

lemma new_subc_put':
  ‹new_term c s ⟹ subc_term c s (sub_term m u t) = subc_term c s (sub_term m (subc_term c s u) t)›
  ‹new_term c s ⟹ subc_list c s (sub_list m u l) = subc_list c s (sub_list m (subc_term c s u) l)›
  using new_subc_same'
  by (induct t and l rule: subc_term.induct subc_list.induct) simp_all

lemma new_subc_put: ‹new_term c s ⟹ subc c s (sub m t p) = subc c s (sub m (subc_term c s t) p)›
proof (induct p arbitrary: s m t)
  case Falsity
```

```
  show ?case
    by simp
next
  case (Pre i l)
  have ‹subc_list c s (sub_list m t l) = subc_list c s (sub_list m (subc_term c s t) l)›
    using Pre.prems new_subc_put'(2) by blast
  then show ?case
    by simp
next
  case (Imp p q)
  have ‹subc c s (sub m t p) = subc c s (sub m (subc_term c s t) p)›
    using Imp.hyps(1) Imp.prems by blast
  moreover have ‹subc c s (sub m t q) = subc c s (sub m (subc_term c s t) q)›
    using Imp.hyps(2) Imp.prems by blast
  ultimately show ?case
    by simp
next
  case (Dis p q)
  have ‹subc c s (sub m t p) = subc c s (sub m (subc_term c s t) p)›
    using Dis.hyps(1) Dis.prems by blast
  moreover have ‹subc c s (sub m t q) = subc c s (sub m (subc_term c s t) q)›
    using Dis.hyps(2) Dis.prems by blast
  ultimately show ?case
    by simp
next
  case (Con p q)
  have ‹subc c s (sub m t p) = subc c s (sub m (subc_term c s t) p)›
```

```
    using Con.hyps(1) Con.prems by blast
  moreover have ‹subc c s (sub m t q) = subc c s (sub m (subc_term c s t) q)›
    using Con.hyps(2) Con.prems by blast
  ultimately show ?case
    by simp
next
  case (Exi p)
  have ‹subc c s (sub m (subc_term c s t) (Exi p)) =
      Exi (subc c (inc_term s) (sub (Suc m) (subc_term c (inc_term s) (inc_term t)) p))›
    using inc_subc by simp
  also have ‹… = Exi (subc c (inc_term s) (sub (Suc m) (inc_term t) p))›
    using Exi new_inc' by metis
  finally show ?case
    by simp
next
  case (Uni p)
  have ‹subc c s (sub m (subc_term c s t) (Uni p)) =
      Uni (subc c (inc_term s) (sub (Suc m) (subc_term c (inc_term s) (inc_term t)) p))›
    using inc_subc by simp
  also have ‹… = Uni (subc c (inc_term s) (sub (Suc m) (inc_term t) p))›
    using Uni new_inc' by metis
  finally show ?case
    by simp
qed

lemma subc_sub_new':
  ‹new_term c u ⟹ subc_term c (sub_term m u s) (sub_term m u t) = sub_term m u (subc_term c s t)›
```

```
  ‹new_term c u ⟹ subc_list c (sub_term m u s) (sub_list m u l) = sub_list m u (subc_list c s l)›
  by (induct t and l rule: subc_term.induct subc_list.induct) simp_all

lemma subc_sub_new:
  ‹new_term c t ⟹ subc c (sub_term m t s) (sub m t p) = sub m t (subc c s p)›
  using subc_sub_new' inc_sub' by (induct p arbitrary: m t s) simp_all

lemma subc_sub_0_new [simp]:
  ‹new_term c t ⟹ subc c s (sub 0 t p) = sub 0 t (subc c (inc_term s) p)›
  using subc_sub_new sub_0_inc by metis

lemma member_subc: ‹member p z ⟹ member (subc c s p) (subcs c s z)›
  by (induct z) auto

lemma OK_subc: ‹OK p z ⟹ OK (subc c s p) (subcs c s z)›
proof (induct p z arbitrary: c s rule: OK.induct)
  case (Assume p z)
  then show ?case
    using member_subc OK.Assume by blast
next
  case (Imp_E p q z)
  then have
    ‹OK (Imp (subc c s p) (subc c s q)) (subcs c s z)›
    ‹OK (subc c s p) (subcs c s z)›
    by simp_all
  then show ?case
    using OK.Imp_E by blast
```

```
next
  case (Dis_E p q z r)
  then have
    ‹OK (Dis (subc c s p) (subc c s q)) (subcs c s z)›
    ‹OK (subc c s r) (subc c s p # subcs c s z)›
    ‹OK (subc c s r) (subc c s q # subcs c s z)›
    by simp_all
  then show ?case
    using OK.Dis_E by blast
next
  case (Exi_E p z q d)
  then show ?case
  proof (cases ‹c = d›)
    case True
    then have ‹OK q z›
      using Exi_E OK.Exi_E by blast
    moreover have ‹new c q› and ‹news c z›
      using Exi_E True by simp_all
    ultimately show ?thesis
      by simp
  next
    case False
    let ?params = ‹params p ∪ params q ∪ (⋃p ∈ set z. params p) ∪ params_term s ∪ {c} ∪ {d}›

    have ‹finite ?params›
      by simp
    then obtain fresh where fresh: ‹fresh ∉ ?params›
```

```
  by (meson ex_new_if_finite infinite_UNIV_listI)

let ?s = ‹psubst_term (id(d := fresh)) s›
let ?f = ‹id(d := fresh, fresh := d)›

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

have ‹new_term d ?s›
  using fresh psubst_new_free'(1) by simp
then have ‹psubst_term ?f ?s = psubst_term (id(fresh := d)) ?s›
  using new_params' fun_upd_twist(1) psubst_upd'(1) by metis
then have psubst_s: ‹psubst_term ?f ?s = s›
  using fresh psubst_new_away' by simp

have ‹?f c = c› and ‹new_term (?f c) (Fun fresh [])›
  using False fresh by auto

have ‹OK (subc c (psubst_term ?f ?s) (Exi p)) (subcs c (psubst_term ?f ?s) z)›
  using Exi_E by blast
then have exi_p:
  ‹OK (Exi (subc c (inc_term (psubst_term ?f ?s)) p)) (subcs c s z)›
  using psubst_s by simp

have ‹news d z›
  using Exi_E by simp
moreover have ‹news fresh z›
```

```
  using fresh by (induct z) simp_all
ultimately have ‹map (psubst ?f) z = z›
  by (induct z) simp_all
moreover have ‹∀x ∈ ⋃p ∈ set z. params p. x ≠ c ⟶ ?f x ≠ ?f c›
  by simp
ultimately have psubst_z: ‹map (psubst ?f) (subcs c ?s z) = subcs c s z›
  using ‹?f c = c› psubst_s by simp

have ‹psubst ?f (subc c ?s (sub 0 (Fun d []) p)) =
  subc (?f c) (psubst_term ?f ?s) (psubst ?f (sub 0 (Fun d []) p))›
  using subc_psubst fresh by simp
also have ‹… = subc c s (sub 0 (Fun fresh []) (psubst ?f p))›
  using psubst_sub psubst_s ‹?f c = c› by simp
also have ‹… = subc c s (sub 0 (Fun fresh []) p)›
  using Exi_E fresh by simp
finally have psubst_p: ‹psubst ?f (subc c ?s (sub 0 (Fun d []) p)) =
    sub 0 (Fun fresh []) (subc c (inc_term s) p)›
  using ‹new_term (?f c) (Fun fresh [])› ‹?f c = c› by (simp del: subc_psubst)

have ‹∀x ∈ params q. x ≠ c ⟶ ?f x ≠ ?f c›
  using f by blast
then have psubst_q: ‹psubst ?f (subc c ?s q) = subc c s q›
  using Exi_E fresh ‹?f c = c› psubst_s subc_psubst f by simp

have ‹OK (subc c ?s q) (subcs c ?s (sub 0 (Fun d []) p # z))›
  using Exi_E by blast
then have ‹OK (subc c ?s q) (subc c ?s (sub 0 (Fun d []) p) # subcs c ?s z)›
```

```
    by simp
   then have ‹OK (psubst ?f (subc c ?s q)) (psubst ?f (subc c ?s (sub 0 (Fun d []) p))
      # map (psubst ?f) (subcs c ?s z))›
    using OK_psubst by (fastforce simp del: subc_psubst subcs_psubst)
   then have q: ‹OK (subc c s q) (sub 0 (Fun fresh []) (subc c (inc_term s) p) # subcs c s z)›
    using psubst_q psubst_z psubst_p by simp

   have ‹new fresh (subc c (inc_term s) p)›
    using fresh new_subc by simp
   moreover have ‹new fresh (subc c s q)›
    using fresh new_subc by simp
   moreover have ‹news fresh (subcs c s z)›
    using fresh ‹news fresh z› news_subcs by simp
   ultimately have news_pqz: ‹news fresh (subc c (inc_term s) p # subc c s q # subcs c s z)›
    by simp

   show ‹OK (subc c s q) (subcs c s z)›
    using OK.Exi_E exi_p q news_pqz psubst_s by metis
  qed
next
  case (Exi_I t p z)
  let ?params = ‹params p ∪ (⋃p ∈ set z. params p) ∪ params_term s ∪ params_term t ∪ {c}›

  have ‹finite ?params›
   by simp
  then obtain fresh where fresh: ‹fresh ∉ ?params›
   by (meson ex_new_if_finite infinite_UNIV_listI)
```

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

have c: ‹?g c = c›
  using fresh by simp
have s: ‹psubst_term ?g ?s = s›
  using fresh psubst_new_away' by simp
have p: ‹psubst ?g (Exi p) = Exi p›
  using fresh psubst_new_away by simp

have ‹∀x ∈ (⋃p ∈ set z. params p). x ≠ c ⟶ ?g x ≠ ?g c›
  using fresh by auto
moreover have ‹map (psubst ?g) z = z›
  using fresh by (induct z) simp_all
ultimately have z: ‹map (psubst ?g) (subcs c ?s z) = subcs c s z›
  using s by simp

have ‹new_term c ?s›
  using fresh psubst_new_free' by simp
then have ‹OK (subc c ?s (sub 0 (subc_term c ?s t) p)) (subcs c ?s z)›
  using Exi_I new_subc_put by metis
moreover have ‹new_term c (subc_term c ?s t)›
  using ‹new_term c ?s› new_subc_same' by blast
ultimately have ‹OK (sub 0 (subc_term c ?s t) (subc c (inc_term ?s) p)) (subcs c ?s z)›
  by simp
```

```
  then have ‹OK (subc c ?s (Exi p)) (subcs c ?s z)›
    using OK.Exi_I by simp
  then have ‹OK (psubst ?g (subc c ?s (Exi p))) (map (psubst ?g) (subcs c ?s z))›
    using OK_psubst by blast
  moreover have ‹∀x ∈ params (Exi p). x ≠ c ⟶ ?g x ≠ ?g c›
    using fresh by auto
  ultimately show ‹OK (subc c s (Exi p)) (subcs c s z)›
    using subc_psubst c s p z by simp
next
  case (Uni_E p z t)
  let ?params = ‹params p ∪ (⋃p ∈ set z. params p) ∪ params_term s ∪ params_term t ∪ {c}›

  have ‹finite ?params›
    by simp
  then obtain fresh where fresh: ‹fresh ∉ ?params›
    by (meson ex_new_if_finite infinite_UNIV_listI)

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

  have c: ‹?g c = c›
    using fresh by simp
  have s: ‹psubst_term ?g ?s = s›
    using fresh psubst_new_away' by simp
  have p: ‹psubst ?g (sub 0 t p) = sub 0 t p›
```

```
  using fresh psubst_new psubst_sub sub_new psubst_new'(1) by auto

have ‹∀x ∈ (⋃p ∈ set z. params p). x ≠ c ⟶ ?g x ≠ ?g c›
  using fresh by auto
moreover have ‹map (psubst ?g) z = z›
  using fresh by (induct z) simp_all
ultimately have z: ‹map (psubst ?g) (subcs c ?s z) = subcs c s z›
  using s by simp

have ‹new_term c ?s›
  using fresh psubst_new_free' by simp

have ‹OK (Uni (subc c (inc_term ?s) p)) (subcs c ?s z)›
  using Uni_E by simp
then have ‹OK (sub 0 (subc_term c ?s t) (subc c (inc_term ?s) p)) (subcs c ?s z)›
  using OK.Uni_E by blast
moreover have ‹new_term c (subc_term c ?s t)›
  using ‹new_term c ?s› new_subc_same' by blast
ultimately have ‹OK (subc c ?s (sub 0 (subc_term c ?s t) p)) (subcs c ?s z)›
  by simp
then have ‹OK (subc c ?s (sub 0 t p)) (subcs c ?s z)›
  using new_subc_put ‹new_term c ?s› by metis
then have ‹OK (psubst ?g (subc c ?s (sub 0 t p))) (map (psubst ?g) (subcs c ?s z))›
  using OK_psubst by blast
moreover have ‹∀x ∈ params (sub 0 t p). x ≠ c ⟶ ?g x ≠ ?g c›
  using fresh p psubst_new_free new_params by (metis fun_upd_apply id_apply)
ultimately show ‹OK (subc c s (sub 0 t p)) (subcs c s z)›
```

```
    using subc_psubst c s p z by simp
  next
    case (Uni_I d p z)
    then show ?case
    proof (cases ‹c = d›)
      case True
      then have ‹OK (Uni p) z›
        using Uni_I OK.Uni_I by blast
      moreover have ‹new c p› and ‹news c z›
        using Uni_I True by simp_all
      ultimately show ?thesis
        by simp
    next
      case False
      let ?params = ‹params p ∪ (⋃p ∈ set z. params p) ∪ params_term s ∪ {c} ∪ {d}›

      have ‹finite ?params›
        by simp
      then obtain fresh where fresh: ‹fresh ∉ ?params›
        by (meson ex_new_if_finite infinite_UNIV_listI)

      let ?s = ‹psubst_term (id(d := fresh)) s›
      let ?f = ‹id(d := fresh, fresh := d)›

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

```
have ‹new_term d ?s›
  using fresh psubst_new_free' by simp
then have ‹psubst_term ?f ?s = psubst_term (id(fresh := d)) ?s›
  using new_params' fun_upd_twist(1) psubst_upd'(1) by metis
then have psubst_s: ‹psubst_term ?f ?s = s›
  using fresh psubst_new_away' by simp

have ‹?f c = c› and ‹new_term c (Fun fresh [])›
  using False fresh by auto

have ‹psubst ?f (subc c ?s (sub 0 (Fun d []) p)) =
  subc (?f c) (psubst_term ?f ?s) (psubst ?f (sub 0 (Fun d []) p))›
  using subc_psubst by simp
also have ‹… = subc c s (sub 0 (Fun fresh []) (psubst ?f p))›
  using ‹?f c = c› psubst_sub psubst_s by simp
also have ‹… = subc c s (sub 0 (Fun fresh []) p)›
  using Uni_I fresh by simp
finally have psubst_p: ‹psubst ?f (subc c ?s (sub 0 (Fun d []) p)) =
    sub 0 (Fun fresh []) (subc c (inc_term s) p)›
  using ‹new_term c (Fun fresh [])› by simp

have ‹news d z›
  using Uni_I by simp
moreover have ‹news fresh z›
  using fresh by (induct z) simp_all
ultimately have ‹map (psubst ?f) z = z›
  by (induct z) simp_all
```

```
    moreover have ‹∀x ∈ ⋃p ∈ set z. params p. x ≠ c ⟶ ?f x ≠ ?f c›
      by auto
    ultimately have psubst_z: ‹map (psubst ?f) (subcs c ?s z) = subcs c s z›
      using ‹?f c = c› psubst_s by simp

    have ‹OK (subc c ?s (sub 0 (Fun d []) p)) (subcs c ?s z)›
      using Uni_I by blast
    then have ‹OK (psubst ?f (subc c ?s (sub 0 (Fun d []) p))) (map (psubst ?f) (subcs c ?s z))›
      using OK_psubst by blast
    then have ‹OK (psubst ?f (subc c ?s (sub 0 (Fun d []) p))) (subcs c s z)›
      using psubst_z by simp
    then have sub_p: ‹OK (sub 0 (Fun fresh []) (subc c (inc_term s) p)) (subcs c s z)›
      using psubst_p by simp

    have ‹new_term fresh s›
      using fresh by simp
    then have ‹new_term fresh (inc_term s)›
      by simp
    then have ‹new fresh (subc c (inc_term s) p)›
      using fresh new_subc by simp
    moreover have ‹news fresh (subcs c s z)›
      using ‹news fresh z› ‹new_term fresh s› news_subcs by fast
    ultimately show ‹OK (subc c s (Uni p)) (subcs c s z)›
      using OK.Uni_I sub_p by simp
  qed
qed (auto intro: OK.intros)
```

```
subsection ‹Weakening Assumptions›

lemma psubst_new_subset:
  assumes ‹set z ⊆ set z'› ‹c ∉ (⋃p ∈ set z. params p)›
  shows ‹set z ⊆ set (map (psubst (id(c := n))) z')›
  using assms by force

lemma subset_cons: ‹set z ⊆ set z' ⟹ set (p # z) ⊆ set (p # z')›
  by auto

lemma weaken_assumptions: ‹OK p z ⟹ set z ⊆ set z' ⟹ OK p z'›
proof (induct p z arbitrary: z' rule: OK.induct)
  case (Assume p z)
  then show ?case
    using OK.Assume by auto
next
  case (Boole p z)
  then have ‹OK Falsity (Neg p # z')›
    using subset_cons by metis
  then show ?case
    using OK.Boole by blast
next
  case (Imp_I q p z)
  then have ‹OK q (p # z')›
    using subset_cons by metis
  then show ?case
    using OK.Imp_I by blast
```

```
next
 case (Dis_E p q z r)
 then have ‹OK r (p # z')› ‹OK r (q # z')›
  using subset_cons by metis+
 then show ?case
  using OK.Dis_E Dis_E by blast
next
 case (Exi_E p z q c)
 let ?params = ‹params p ∪ params q ∪ (⋃p ∈ set z'. params p) ∪ {c}›

 have ‹finite ?params›
  by simp
 then obtain fresh where ‹fresh ∉ ?params›
  by (meson ex_new_if_finite List.finite_set infinite_UNIV_listI)
 then have fresh: ‹new fresh p ∧ new fresh q ∧ news fresh z' ∧ fresh ≠ c›
  using allnew new_params by (metis Ball_set UN_iff UnI1 UnI2 insertCI)

 let ?z' = ‹map (psubst (id(c := fresh))) z'›

 have ‹news c z›
  using Exi_E by simp
 then have ‹set z ⊆ set ?z'›
  using Exi_E psubst_new_subset by (simp add: Ball_set)
 then have ‹OK (Exi p) ?z'›
  using Exi_E by blast

 moreover have ‹set (sub 0 (Fun c []) p # z) ⊆ set (sub 0 (Fun c []) p # ?z')›
```

```
    using ‹set z ⊆ set ?z'› by auto
  then have ‹OK q (sub 0 (Fun c []) p # ?z')›
    using Exi_E by blast

  moreover have ‹news c ?z'›
    using fresh map_psubst_new_free by simp
  then have ‹news c (p # q # ?z')›
    using Exi_E by simp

  ultimately have ‹OK q ?z'›
    using Exi_E OK.Exi_E by blast

  then have ‹OK (psubst (id(fresh := c)) q) (map (psubst (id(fresh := c))) ?z')›
    using OK_psubst by blast
  moreover have ‹map (psubst (id(fresh := c))) ?z' = z'›
    using fresh map_psubst_new_away by blast
  moreover have ‹psubst (id(fresh := c)) q = q›
    using fresh by simp
  ultimately show ‹OK q z'›
    by simp
next
  case (Uni_I c p z)
  let ?params = ‹params p ∪ (⋃p ∈ set z'. params p) ∪ {c}›

  have ‹finite ?params›
    by simp
  then obtain fresh where ‹fresh ∉ ?params›
```

```
  by (meson ex_new_if_finite List.finite_set infinite_UNIV_listI)
then have fresh: ‹new fresh p ∧ news fresh z' ∧ fresh ≠ c›
  using allnew new_params by (metis Ball_set UN_iff UnI1 UnI2 insertCI)

let ?z' = ‹map (psubst (id(c := fresh))) z'›

have ‹news c z›
  using Uni_I by simp
then have ‹set z ⊆ set ?z'›
  using Uni_I psubst_new_subset allnew new_params map_psubst_new image_set subset_image_iff
  by (metis (no_types))
then have ‹OK (sub 0 (Fun c []) p) ?z'›
  using Uni_I by blast

moreover have ‹∀p ∈ set ?z'. c ∉ params p›
  using fresh psubst_new_free by simp
then have ‹list_all (λp. c ∉ params p) (p # ?z')›
  using Uni_I by (simp add: list_all_iff)
then have ‹news c (p # ?z')›
  by simp

ultimately have ‹OK (Uni p) ?z'›
  using Uni_I OK.Uni_I by blast

then have ‹OK (psubst (id(fresh := c)) (Uni p)) (map (psubst (id(fresh := c))) ?z')›
  using OK_psubst by blast
moreover have ‹map (psubst (id(fresh := c))) ?z' = z'›
```

```
    using fresh map_psubst_new_away by blast
  moreover have ‹psubst (id(fresh := c)) (Uni p) = Uni p›
    using fresh Uni_I by simp
  ultimately show ‹OK (Uni p) z'›
    by simp
qed (auto intro: OK.intros)

subsection ‹Implications and Assumptions›

primrec put_imps :: ‹fm ⇒ fm list ⇒ fm› where
  ‹put_imps p [] = p› |
  ‹put_imps p (q # z) = Imp q (put_imps p z)›

lemma semantics_put_imps:
  ‹(list_all (semantics e f g) z ⟶ semantics e f g p) = semantics e f g (put_imps p z)›
  by (induct z) auto

lemma shift_imp_assum:
  assumes ‹OK (Imp p q) z›
  shows ‹OK q (p # z)›
proof -
  have ‹set z ⊆ set (p # z)›
    by auto
  then have ‹OK (Imp p q) (p # z)›
    using assms weaken_assumptions by blast
  moreover have ‹OK p (p # z)›
    using Assume by simp
```

```
  ultimately show ‹OK q (p # z)›
    using Imp_E by blast
qed

lemma remove_imps: ‹OK (put_imps p z) z' ⟹ OK p (rev z @ z')›
  using shift_imp_assum by (induct z arbitrary: z') simp_all

subsection ‹Closure Elimination›

lemma subc_sub_closed_var' [simp]:
  ‹new_term c t ⟹ closed_term (Suc m) t ⟹ subc_term c (Var m) (sub_term m (Fun c []) t) = t›
  ‹new_list c l ⟹ closed_list (Suc m) l ⟹ subc_list c (Var m) (sub_list m (Fun c []) l) = l›
  by (induct t and l rule: sub_term.induct sub_list.induct) auto

lemma subc_sub_closed_var [simp]: ‹new c p ⟹ closed (Suc m) p ⟹
    subc c (Var m) (sub m (Fun c []) p) = p›
  by (induct p arbitrary: m) simp_all

primrec put_unis :: ‹nat ⇒ fm ⇒ fm› where
  ‹put_unis 0 p = p› |
  ‹put_unis (Suc m) p = Uni (put_unis m p)›

lemma sub_put_unis [simp]: ‹sub i (Fun c []) (put_unis k p) = put_unis k (sub (i + k) (Fun c []) p)›
  by (induct k arbitrary: i) simp_all

lemma closed_put_unis [simp]: ‹closed m (put_unis k p) = closed (m + k) p›
  by (induct k arbitrary: m) simp_all
```

```
lemma valid_put_unis: ‹∀(e :: _ ⇒ 'a) f g. semantics e f g p ⟹
    semantics (e :: _ ⇒ 'a) f g (put_unis m p)›
  by (induct m arbitrary: e) simp_all

lemma put_unis_collapse: ‹put_unis m (put_unis n p) = put_unis (m + n) p›
  by (induct m) simp_all

fun consts_for_unis :: ‹fm ⇒ id list ⇒ fm› where
  ‹consts_for_unis (Uni p) (c#cs) = consts_for_unis (sub 0 (Fun c []) p) cs› |
  ‹consts_for_unis p _ = p›

lemma consts_for_unis: ‹OK (put_unis (length cs) p) [] ⟹
  OK (consts_for_unis (put_unis (length cs) p) cs) []›
proof (induct cs arbitrary: p)
  case (Cons c cs)
  then have ‹OK (Uni (put_unis (length cs) p)) []›
    by simp
  then have ‹OK (sub 0 (Fun c []) (put_unis (length cs) p)) []›
    using Uni_E by blast
  then show ?case
    using Cons by simp
qed simp

primrec vars_for_consts :: ‹fm ⇒ id list ⇒ fm› where
  ‹vars_for_consts p [] = p› |
  ‹vars_for_consts p (c # cs) = subc c (Var (length cs)) (vars_for_consts p cs)›
```

```
lemma vars_for_consts: ‹OK p [] ⟹ OK (vars_for_consts p xs) []›
  using OK_subc by (induct xs arbitrary: p) fastforce+

lemma vars_for_consts_for_unis:
  ‹closed (length cs) p ⟹ list_all (λc. new c p) cs ⟹ distinct cs ⟹
   vars_for_consts (consts_for_unis (put_unis (length cs) p) cs) cs = p›
  using sub_new_all by (induct cs arbitrary: p) (auto simp: list_all_iff)

lemma fresh_constant: ‹∃c. c ∉ set cs ∧ new c p›
proof -
  have ‹finite (set cs ∪ params p)›
    by simp
  then show ?thesis
    using ex_new_if_finite UnI1 UnI2 infinite_UNIV_listI new_params by metis
qed

lemma fresh_constants: ‹∃cs. length cs = m ∧ list_all (λc. new c p) cs ∧ distinct cs›
proof (induct m)
  case (Suc m)
  then obtain cs where ‹length cs = m ∧ list_all (λc. new c p) cs ∧ distinct cs›
    by blast
  moreover obtain c where ‹c ∉ set cs ∧ new c p›
    using Suc fresh_constant by blast
  ultimately have ‹length (c # cs) = Suc m ∧ list_all (λc. new c p) (c # cs) ∧ distinct (c # cs)›
    by simp
  then show ?case
```

```
    by blast
qed simp

lemma closed_max:
  assumes ‹closed m p› ‹closed n q›
  shows ‹closed (max m n) p ∧ closed (max m n) q›
proof -
  have ‹m ≤ max m n› and ‹n ≤ max m n›
    by simp_all
  then show ?thesis
    using assms closed_mono by metis
qed

lemma ex_closed' [simp]: ‹∃m. closed_term m t› ‹∃n. closed_list n l›
proof (induct t and l rule: closed_term.induct closed_list.induct)
  case (Cons_tm t l)
  then obtain m and n where ‹closed_term m t› and ‹closed_list n l›
    by blast
  moreover have ‹m ≤ max m n› and ‹n ≤ max m n›
    by simp_all
  ultimately have ‹closed_term (max m n) t› and ‹closed_list (max m n) l›
    using closed_mono' by blast+
  then show ?case
    by auto
qed auto

lemma ex_closed [simp]: ‹∃m. closed m p›
```

```
proof (induct p)
  case (Imp p q)
  then show ?case
    using closed_max by fastforce
next
  case (Dis p q)
  then show ?case
    using closed_max by fastforce
next
  case (Con p q)
  then show ?case
    using closed_max by fastforce
next
  case (Exi p)
  then obtain m where ‹closed m p›
    by blast
  then have ‹closed (Suc m) p›
    using closed_mono Suc_n_not_le_n nat_le_linear by blast
  then show ?case
    by auto
next
  case (Uni p)
  then obtain m where ‹closed m p›
    by blast
  then have ‹closed (Suc m) p›
    using closed_mono Suc_n_not_le_n nat_le_linear by blast
  then show ?case
```

```
   by auto
qed simp_all

lemma ex_closure: ‹∃m. sentence (put_unis m p)›
  by simp

lemma remove_unis_sentence:
  assumes ‹sentence (put_unis m p)› ‹OK (put_unis m p) []›
  shows ‹OK p []›
proof -
  obtain cs :: ‹id list› where ‹length cs = m›
    and *: ‹distinct cs› and **: ‹list_all (λc. new c p) cs›
    using assms fresh_constants by blast
  then have ‹OK (consts_for_unis (put_unis (length cs) p) cs) []›
    using assms consts_for_unis by blast
  then have ‹OK (vars_for_consts (consts_for_unis (put_unis (length cs) p) cs) cs) []›
    using vars_for_consts by blast
  moreover have ‹closed (length cs) p›
    using assms ‹length cs = m› by simp
  ultimately show ‹OK p []›
    using vars_for_consts_for_unis * ** by simp
qed

section ‹Completeness›

theorem completeness':
  assumes ‹∀(e :: _ ⇒ 'a) f g. list_all (semantics e f g) z ⟶ semantics e f g p›
```

```
    and ‹denumerable (UNIV :: 'a set)›
  shows ‹OK p z›
proof -
  let ?p = ‹put_imps p (rev z)›

  have *: ‹∀(e :: _ ⇒ 'a) f g. semantics e f g ?p›
    using assms(1) semantics_put_imps by fastforce
  obtain m where **: ‹sentence (put_unis m ?p)›
    using ex_closure by blast
  moreover have ‹∀(e :: _ ⇒ 'a) f g. semantics e f g (put_unis m ?p)›
    using * valid_put_unis by blast
  ultimately have ‹OK (put_unis m ?p) []›
    using assms(2) sentence_completeness by blast
  then have ‹OK ?p []›
    using ** remove_unis_sentence by blast
  then show ‹OK p z›
    using remove_imps by fastforce
qed

lemma completeness'':
  assumes ‹∀(e :: _ ⇒ htm) f g. list_all (semantics e f g) z ⟶ semantics e f g p›
  shows ‹OK p z›
  using assms completeness' denumerable_htm by fast

theorem completeness:
  assumes ‹∀(e :: _ ⇒ 'a) f g. semantics e f g p›
    and ‹denumerable (UNIV :: 'a set)›
```

```
  shows ‹OK p []›
  using assms by (simp add: completeness')

corollary ‹∀(e :: _ ⇒ nat) f g. semantics e f g p ⟹ OK p []›
  using completeness denumerable_bij by blast

section ‹Main Result› — ‹NaDeA is sound and complete›

abbreviation ‹valid p ≡ ∀(e :: _ ⇒ nat) f g. semantics e f g p›

theorem main: ‹valid p ⟷ OK p []›
proof
  assume ‹valid p›
  with completeness show ‹OK p []›
    using denumerable_bij by blast
next
  assume ‹OK p []›
  with soundness show ‹valid p›
    by (intro allI)
qed

theorem valid_semantics: ‹valid p ⟶ semantics e f g p›
proof
  assume ‹valid p›
  then have ‹OK p []›
    unfolding main .
  with soundness show ‹semantics e f g p› .
```

```
qed

theorem any_unis: ‹OK (put_unis k p) [] ⟹ OK (put_unis m p) []›
  using main ex_closure put_unis_collapse remove_unis_sentence valid_put_unis by metis

corollary ‹OK p [] ⟹ OK (put_unis m p) []› ‹OK (put_unis m p) [] ⟹ OK p []›
  using any_unis put_unis.simps(1) by metis+

section ‹Tableau Calculus› — ‹NaDeA variant›

inductive TC :: ‹fm list ⇒ bool› (‹⊣ _› 0) where
  Dummy: ‹⊣ Falsity # z› |
  Basic: ‹⊣ Pre i l # Neg (Pre i l) # z› |
  AlImp: ‹⊣ p # Neg q # z ⟹ ⊣ Neg (Imp p q) # z› |
  AlDis: ‹⊣ Neg p # Neg q # z ⟹ ⊣ Neg (Dis p q) # z› |
  AlCon: ‹⊣ p # q # z ⟹ ⊣ Con p q # z› |
  BeImp: ‹⊣ Neg p # z ⟹ ⊣ q # z ⟹ ⊣ Imp p q # z› |
  BeDis: ‹⊣ p # z ⟹ ⊣ q # z ⟹ ⊣ Dis p q # z› |
  BeCon: ‹⊣ Neg p # z ⟹ ⊣ Neg q # z ⟹ ⊣ Neg (Con p q) # z› |
  GaExi: ‹⊣ Neg (sub 0 t p) # z ⟹ ⊣ Neg (Exi p) # z› |
  GaUni: ‹⊣ sub 0 t p # z ⟹ ⊣ Uni p # z› |
  DeExi: ‹⊣ sub 0 (Fun c []) p # z ⟹ news c (p # z) ⟹ ⊣ Exi p # z› |
  DeUni: ‹⊣ Neg (sub 0 (Fun c []) p) # z ⟹ news c (p # z) ⟹ ⊣ Neg (Uni p) # z› |
  Extra: ‹⊣ p # z ⟹ member p z ⟹ ⊣ z›

fun compl :: ‹fm ⇒ fm› where
  ‹compl (Neg p) = p› |
```

```
 ‹compl p = Neg p›

definition tableauproof :: ‹fm list ⇒ fm ⇒ bool› where
 ‹tableauproof z p ≡ (⊣ compl p # z)›

lemma compl: ‹compl p = Neg p ∨ (∃q. compl p = q ∧ p = Neg q)›
 by (induct p rule: compl.induct) simp_all

lemma compl_compl: ‹semantics e f g p ⟷ semantics e f g (compl (compl p))›
 using compl by (metis semantics.simps(1) semantics.simps(3))

lemma new_compl: ‹new n p ⟹ new n (compl p)›
 by (cases p rule: compl.cases) simp_all

lemma news_compl: ‹news c z ⟹ news c (map compl z)›
 using new_compl by (induct z) simp_all

lemma closed_compl: ‹closed m p ⟹ closed m (compl p)›
proof (induct p arbitrary: m)
 case (Imp p1 p2)
 then show ?case
  by (metis closed.simps(5) compl)
qed simp_all

lemma semantics_compl: ‹¬ (semantics e f g (compl p)) ⟷ semantics e f g p›
proof (induct p)
 case (Imp p1 p2)
```

```
  then show ?case
    using compl_compl by (metis compl.simps(1) semantics.simps(1) semantics.simps(3))
qed simp_all

subsection ‹Soundness›

theorem TC_soundness:
  ‹⊣ z ⟹ ∃p ∈ set z. ¬ semantics e f g p›
proof (induct arbitrary: f rule: TC.induct)
  case (Extra p z)
  then show ?case
    by fastforce
next
  case (DeExi n p z)
  show ?case
  proof (rule ccontr)
    assume ‹¬ (∃p ∈ set (Exi p # z). ¬ semantics e f g p)›
    then have *: ‹∀p ∈ set (Exi p # z). semantics e f g p›
      by simp

    then obtain x where ‹semantics (put e 0 x) (f(n := λw. x)) g p›
      using DeExi.hyps(3) by auto
    then have **: ‹semantics e (f(n := λw. x)) g (sub 0 (Fun n []) p)›
      by simp

    have ‹∃p ∈ set (sub 0 (Fun n []) p # z). ¬ semantics e (f(n := λw. x)) g p›
      using DeExi by fast
```

```
    then consider
      ‹¬ semantics e (f(n := λw. x)) g (sub 0 (Fun n []) p)› |
      ‹∃p ∈ set z. ¬ semantics e (f(n := λw. x)) g p›
      by auto
    then show False
    proof cases
      case 1
      then show ?thesis
        using ** by simp
    next
      case 2
      then obtain p where ‹¬ semantics e (f(n := λw. x)) g p› ‹p ∈ set z›
        by blast
      then have ‹¬ semantics e f g p›
        using DeExi.hyps(3) by (metis Ball_set allnew map news.simps(2))
      then show False
        using * ‹p ∈ set z› by simp
    qed
  qed
next
  case (DeUni n p z)
  show ?case
  proof (rule ccontr)
    assume ‹¬ (∃p ∈ set (Neg (Uni p) # z). ¬ semantics e f g p)›
    then have *: ‹∀p ∈ set (Neg (Uni p) # z). semantics e f g p›
      by simp
```

```
    then obtain x where ‹semantics (put e 0 x) (f(n := λw. x)) g (Neg p)›
      using DeUni.hyps(3) by auto
    then have **: ‹semantics e (f(n := λw. x)) g (sub 0 (Fun n []) (Neg p))›
      by simp

    have ‹∃p ∈ set (Neg (sub 0 (Fun n []) p) # z). ¬ semantics e (f(n := λw. x)) g p›
      using DeUni by fast
    then consider
      ‹¬ semantics e (f(n := λw. x)) g (Neg (sub 0 (Fun n []) p))› |
      ‹∃p ∈ set z. ¬ semantics e (f(n := λw. x)) g p›
      by auto
    then show False
    proof cases
      case 1
      then show ?thesis
        using ** by simp
    next
      case 2
      then obtain p where ‹¬ semantics e (f(n := λw. x)) g p› ‹p ∈ set z›
        by blast
      then have ‹¬ semantics e f g p›
        using DeUni.hyps(3) by (metis Ball_set allnew map news.simps(2))
      then show False
        using * ‹p ∈ set z› by simp
    qed
  qed
qed auto
```

```
theorem tableau_soundness:
 ‹tableauproof z p ⟹ list_all (semantics e f g) z ⟹ semantics e f g p›
 unfolding tableauproof_def list_all_def using TC_soundness compl_compl
 by (metis (no_types, hide_lams) compl.simps(1) semantics.simps(3) set_ConsD)

theorem sound:
 assumes ‹⊣ [Neg p]›
 shows ‹semantics e f g p›
proof -
 from assms consider ‹⊣ [compl p]› | ‹∃q. p = Neg q ∧ (⊣ [Neg (Neg q)])›
  using compl by metis
 then show ?thesis
 proof cases
  case 1
  then show ?thesis
   using tableau_soundness unfolding tableauproof_def by fastforce
 next
  case 2
  then obtain q where ‹p = Neg q› ‹⊣ [compl (Neg (Neg (Neg q)))]›
   by auto
  then have ‹semantics e f g (Neg (Neg (Neg q)))›
   using tableau_soundness unfolding tableauproof_def by fastforce
  then show ?thesis
   using ‹p = Neg q› by auto
 qed
qed
```

```
subsection ‹Completeness for Closed Formulas›

theorem infinite_nonempty: ‹infinite p ⟹ ∃x. x ∈ p›
  by (simp add: ex_in_conv infinite_imp_nonempty)

theorem TCd_consistency:
  assumes inf_param: ‹infinite (UNIV::'a set)›
  shows ‹consistency {S. ∃z. S = set z ∧ ¬ (⊣ z)}›
  unfolding consistency_def
proof (intro conjI allI impI notI)
  fix S
  assume ‹S ∈ {set z | z. ¬ (⊣ z)}› (is ‹S ∈ ?C›)
  then obtain z :: ‹fm list›
    where *: ‹S = set z› and ‹¬ (⊣ z)›
    by blast

  { fix p ts
    assume ‹Pre p ts ∈ S ∧ Neg (Pre p ts) ∈ S›
    then show False
      using * Basic ‹¬ (⊣ z)› Extra in_mono set_subset_Cons member_set by metis }

  { assume ‹Falsity ∈ S›
    then show False
      using * Dummy ‹¬ (⊣ z)› Extra member_set by blast }

  { fix p q
```

```
  assume ‹Con p q ∈ S›
  then have ‹¬ (⊣ p # q # z)›
    using * AlCon ‹¬ (⊣ z)› Extra member_set by blast
  moreover have ‹S ∪ {p, q} = set (p # q # z)›
    using * by simp
  ultimately show ‹S ∪ {p, q} ∈ ?C›
    by blast }

{ fix p q
  assume ‹Neg (Dis p q) ∈ S›
  then have ‹¬ (⊣ Neg p # Neg q # z)›
    using * AlDis ‹¬ (⊣ z)› Extra member_set by blast
  moreover have ‹S ∪ {Neg p, Neg q} = set (Neg p # Neg q # z)›
    using * by simp
  ultimately show ‹S ∪ {Neg p, Neg q} ∈ ?C›
    by blast }

{ fix p q
  assume ‹Neg (Imp p q) ∈ S›
  then have ‹¬ (⊣ p # Neg q # z)›
    using * AlImp ‹¬ (⊣ z)› Extra member_set by blast
  moreover have ‹{p, Neg q} ∪ S = set (p # Neg q # z)›
    using * by simp
  ultimately show ‹S ∪ {p, Neg q} ∈ ?C›
    by blast }

{ fix p q
```

```
  assume ‹Dis p q ∈ S›
  then have ‹¬ (⊣ p # z) ∨ ¬ (⊣ q # z)›
    using * BeDis ‹¬ (⊣ z)› Extra member_set by blast
  then show ‹S ∪ {p} ∈ ?C ∨ S ∪ {q} ∈ ?C›
    using * by auto }

{ fix p q
  assume ‹Neg (Con p q) ∈ S›
  then have ‹¬ (⊣ Neg p # z) ∨ ¬ (⊣ Neg q # z)›
    using * BeCon ‹¬ (⊣ z)› Extra member_set by blast
  then show ‹S ∪ {Neg p} ∈ ?C ∨ S ∪ {Neg q} ∈ ?C›
    using * by auto }

{ fix p q
  assume ‹Imp p q ∈ S›
  then have ‹¬ (⊣ Neg p # z) ∨ ¬ (⊣ q # z)›
    using * BeImp ‹¬ (⊣ z)› Extra member_set by blast
  then show ‹S ∪ {Neg p} ∈ ?C ∨ S ∪ {q} ∈ ?C›
    using * by auto }

{ fix P t
  assume ‹closed_term 0 t› and ‹Uni P ∈ S›
  then have ‹¬ (⊣ sub 0 t P # z)›
    using * GaUni ‹¬ (⊣ z)› Extra member_set by blast
  moreover have ‹S ∪ {sub 0 t P} = set (sub 0 t P # z)›
    using * by simp
  ultimately show ‹S ∪ {sub 0 t P} ∈ ?C›
```

```
    by blast }

  { fix P t
   assume ‹closed_term 0 t› and ‹Neg (Exi P) ∈ S›
   then have ‹¬ (⊣ Neg (sub 0 t P) # z)›
    using * GaExi ‹¬ (⊣ z)› Extra member_set by blast
   moreover have ‹S ∪ {Neg (sub 0 t P)} = set (Neg (sub 0 t P) # z)›
    using * by simp
   ultimately show ‹S ∪ {Neg (sub 0 t P)} ∈ ?C›
    by blast }

  { fix P
   assume ‹Exi P ∈ S›
   have ‹finite ((⋃p ∈ set z. params p) ∪ params P)›
    by simp
   then have ‹infinite (- ((⋃p ∈ set z. params p) ∪ params P))›
    using inf_param Diff_infinite_finite finite_compl infinite_UNIV_listI by blast
   then obtain x where **: ‹x ∈ - ((⋃p ∈ set z. params p) ∪ params P)›
    using infinite_imp_nonempty by blast
   then have ‹news x (P # z)›
    using Ball_set_list_all by auto
   then have ‹¬ (⊣ sub 0 (Fun x []) P # z)›
    using * ‹Exi P ∈ S› DeExi ‹¬ (⊣ z)› Extra member_set by blast
   moreover have ‹S ∪ {sub 0 (Fun x []) P} = set (sub 0 (Fun x []) P # z)›
    using * by simp
   ultimately show ‹∃x. S ∪ {sub 0 (Fun x []) P} ∈ ?C›
    by blast }
```

```
 { fix P
   assume ‹Neg (Uni P) ∈ S›
   have ‹finite ((⋃p ∈ set z. params p) ∪ params P)›
     by simp
   then have ‹infinite (- ((⋃p ∈ set z. params p) ∪ params P))›
     using inf_param Diff_infinite_finite finite_compl infinite_UNIV_listI by blast
   then obtain x where **: ‹x ∈ - ((⋃p ∈ set z. params p) ∪ params P)›
     using infinite_imp_nonempty by blast
   then have ‹news x (P # z)›
     using Ball_set_list_all by auto
   then have ‹¬ (⊣ Neg (sub 0 (Fun x []) P) # z)›
     using * ‹Neg (Uni P) ∈ S› DeUni ‹¬ (⊣ z)› Extra member_set by blast
   moreover have ‹S ∪ {Neg (sub 0 (Fun x []) P)} = set (Neg (sub 0 (Fun x []) P) # z)›
     using * by simp
   ultimately show ‹∃x. S ∪ {Neg (sub 0 (Fun x []) P)} ∈ ?C›
     by blast }
qed

theorem tableau_completeness':
  assumes ‹closed 0 p›
    and ‹list_all (closed 0) z›
    and mod: ‹∀(e :: _ ⇒ htm) f g. list_all (semantics e f g) z ⟶ semantics e f g p›
  shows ‹tableauproof z p›
proof (rule ccontr)
  fix e
  assume ‹¬ tableauproof z p›
```

```
let ?S = ‹set (compl p # z)›
let ?C = ‹{set (z :: fm list) | z. ¬ (⊣ z)}›
let ?f = HFun
let ?g = ‹(λa ts. Pre a (tms_of_htms ts) ∈ Extend ?S
      (mk_finite_char (mk_alt_consistency (close ?C))) from_nat)›

from ‹list_all (closed 0) z›
have ‹∀p ∈ set z. 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 z 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 z. closed 0 p› ‹x ∈ ?S›
 then have ‹closed 0 x›
  using closed_compl by auto
 ultimately have ‹semantics e ?f ?g x›
  using model_existence by simp }
then have ‹list_all (semantics e ?f ?g) (compl p # z)›
 by (simp add: list_all_iff)
moreover have ‹semantics e ?f ?g (compl p)›
```

```
    using calculation by simp
  moreover have ‹list_all (semantics e ?f ?g) z›
    using calculation by simp
  then have ‹semantics e ?f ?g p›
    using mod by blast
  ultimately show False
    using semantics_compl by blast
qed

subsection ‹Open Formulas›

lemma TC_psubst: ‹⊣ z ⟹ ⊣ map (psubst f) z›
proof (induct arbitrary: f rule: TC.induct)
  case (Extra p z)
  then show ?case
    by (metis TC.Extra list.simps(9) member_psubst)
next
  case (DeExi n p z)
  let ?params = ‹params p ∪(⋃p ∈ set z. params p)›

  have ‹finite ?params›
    by simp
  then obtain fresh where *: ‹fresh ∉ ?params ∪ {n} ∪ image f ?params›
    using ex_new_if_finite
    by (metis finite.emptyI finite.insertI finite_UnI finite_imageI infinite_UNIV_listI)

  let ?f = ‹f(n := fresh)›
```

```
  have ‹news n (p # z)›
    using DeExi by blast
  then have ‹new fresh (psubst ?f p)› ‹news fresh (map (psubst ?f) z)›
    using * new_psubst_image news_psubst by (fastforce simp add: image_Un)+
  then have z: ‹map (psubst ?f) z = map (psubst f) z›
    using DeExi allnew new_params
    by (metis (mono_tags, lifting) Ball_set map_eq_conv news.simps(2) psubst_upd)

  have ‹⊣ psubst ?f (sub 0 (Fun n []) p) # map (psubst ?f) z›
    using DeExi by (metis list.simps(9))
  then have ‹⊣ sub 0 (Fun fresh []) (psubst ?f p) # map (psubst ?f) z›
    by simp
  moreover have ‹news fresh (map (psubst ?f) (p # z))›
    using ‹new fresh (psubst ?f p)› ‹news fresh (map (psubst ?f) z)› by simp
  then have ‹news fresh (psubst ?f p # map (psubst ?f) z)›
    by simp
  ultimately have ‹⊣ map (psubst ?f) (Exi p # z)›
    using TC.DeExi by fastforce
  then show ?case
    using DeExi z by simp
next
  case (DeUni n p z)
  let ?params = ‹params p ∪(⋃p ∈ set z. params p)›

  have ‹finite ?params›
    by simp
```

```
  then obtain fresh where *: ‹fresh ∉ ?params ∪ {n} ∪ image f ?params›
    using ex_new_if_finite
    by (metis finite.emptyI finite.insertI finite_UnI finite_imageI infinite_UNIV_listI)

  let ?f = ‹f(n := fresh)›

  have ‹news n (p # z)›
    using DeUni by blast
  then have ‹new fresh (psubst ?f p)› ‹news fresh (map (psubst ?f) z)›
    using * new_psubst_image news_psubst by (fastforce simp add: image_Un)+
  then have z: ‹map (psubst ?f) z = map (psubst f) z›
    using DeUni allnew new_params
    by (metis (mono_tags, lifting) Ball_set map_eq_conv news.simps(2) psubst_upd)

  have ‹⊣ psubst ?f (Neg (sub 0 (Fun n []) p)) # map (psubst ?f) z›
    using DeUni by (metis list.simps(9))
  then have ‹⊣ Neg (sub 0 (Fun fresh []) (psubst ?f p)) # map (psubst ?f) z›
    by simp
  moreover have ‹news fresh (map (psubst ?f) (p # z))›
    using ‹new fresh (psubst ?f p)› ‹news fresh (map (psubst ?f) z)› by simp
  then have ‹news fresh (psubst ?f p # map (psubst ?f) z)›
    by simp
  ultimately have ‹⊣ map (psubst ?f) (Neg (Uni p) # z)›
    using TC.DeUni by fastforce
  then show ?case
    using DeUni z by simp
qed (auto intro: TC.intros)
```

```
lemma subcs_map: ‹subcs c s z = map (subc c s) z›
  by (induct z) simp_all

lemma TC_subcs: ‹⊣ z ⟹ ⊣ subcs c s z›
proof (induct arbitrary: c s rule: TC.induct)
  case (Extra p z)
  then show ?case
    by (metis TC.Extra member_subc subcs.simps(2))
next
  case (GaUni t p z)
  let ?params = ‹params p ∪ (⋃p ∈ set z. params p) ∪ params_term s ∪ params_term t ∪ {c}›

  have ‹finite ?params›
    by simp
  then obtain fresh where fresh: ‹fresh ∉ ?params›
    by (meson ex_new_if_finite infinite_UNIV_listI)

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

  have s: ‹psubst_term ?g ?s = s›
    using fresh psubst_new_away' by simp

  have ‹∀x ∈ (⋃p ∈ set (p # z). params p). x ≠ c ⟶ ?g x ≠ ?g c›
    using fresh by auto
```

```
  moreover have ‹map (psubst ?g) (Uni p # z) = Uni p # z›
    using fresh by (induct z) simp_all
  ultimately have z: ‹map (psubst ?g) (subcs c ?s (Uni p # z)) = subcs c s (Uni p # z)›
    using s by simp

  have ‹new_term c ?s›
    using fresh psubst_new_free' by simp
  then have ‹⊣ subc c ?s (sub 0 (subc_term c ?s t) p) # subcs c ?s z›
    using GaUni new_subc_put by (metis subcs.simps(2))
  moreover have ‹new_term c (subc_term c ?s t)›
    using ‹new_term c ?s› new_subc_same' by blast
  ultimately have ‹⊣ sub 0 (subc_term c ?s t) (subc c (inc_term ?s) p) # subcs c ?s z›
    by simp
  moreover have ‹Uni (subc c (inc_term ?s) p) ∈ set (subcs c ?s (Uni p # z))›
    by simp
  ultimately have ‹⊣ subcs c ?s (Uni p # z)›
    using TC.GaUni by simp
  then have ‹⊣ map (psubst ?g) (subcs c ?s (Uni p # z))›
    using TC_psubst by blast
  then show ‹⊣ subcs c s (Uni p # z)›
    using z by simp
next
  case (GaExi t p z)
  let ?params = ‹params p ∪ (⋃p ∈ set z. params p) ∪ params_term s ∪ params_term t ∪ {c}›

  have ‹finite ?params›
    by simp
```

```
then obtain fresh where fresh: ‹fresh ∉ ?params›
  by (meson ex_new_if_finite infinite_UNIV_listI)

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

have s: ‹psubst_term ?g ?s = s›
  using fresh psubst_new_away' by simp

have ‹∀x ∈ (⋃p ∈ set (p # z). params p). x ≠ c ⟶ ?g x ≠ ?g c›
  using fresh by auto
moreover have ‹map (psubst ?g) (Neg (Exi p) # z) = Neg (Exi p) # z›
  using fresh by (induct z) simp_all
ultimately have z: ‹map (psubst ?g) (subcs c ?s (Neg (Exi p) # z)) = subcs c s (Neg (Exi p) # z)›
  using s by simp

have ‹new_term c ?s›
  using fresh psubst_new_free' by simp
then have ‹⊣ Neg (subc c ?s (sub 0 (subc_term c ?s t) p)) # subcs c ?s z›
  using GaExi new_subc_put by (metis subc.simps(1) subc.simps(3) subcs.simps(2))
moreover have ‹new_term c (subc_term c ?s t)›
  using ‹new_term c ?s› new_subc_same' by blast
ultimately have ‹⊣ Neg (sub 0 (subc_term c ?s t) (subc c (inc_term ?s) p)) # subcs c ?s z›
  by simp
moreover have ‹Neg (Exi (subc c (inc_term ?s) p)) ∈ set (subcs c ?s (Neg (Exi p) # z))›
  by simp
```

```
  ultimately have ‹⊣ subcs c ?s (Neg (Exi p) # z)›
    using TC.GaExi by simp
  then have ‹⊣ map (psubst ?g) (subcs c ?s (Neg (Exi p) # z))›
    using TC_psubst by blast
  then show ‹⊣ subcs c s (Neg (Exi p) # z)›
    using z by simp
next
  case (DeExi n p z)
  then show ?case
  proof (cases ‹c = n›)
    case True
    then have ‹⊣ Exi p # z›
      using DeExi TC.DeExi by blast
    moreover have ‹new c p› and ‹news c z›
      using DeExi True by simp_all
    ultimately show ?thesis
      by simp
  next
    case False
    let ?params = ‹params p ∪ (⋃p ∈ set z. params p) ∪ params_term s ∪ {c} ∪ {n}›

    have ‹finite ?params›
      by simp
    then obtain fresh where fresh: ‹fresh ∉ ?params›
      by (meson ex_new_if_finite infinite_UNIV_listI)

    let ?s = ‹psubst_term (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 simp
then have ‹psubst_term ?f ?s = psubst_term (id(fresh := n)) ?s›
  using new_params' fun_upd_twist(1) psubst_upd'(1) by metis
then have psubst_s: ‹psubst_term ?f ?s = s›
  using fresh psubst_new_away' by simp

have ‹?f c = c› and ‹new_term c (Fun fresh [])›
  using False fresh by auto

have ‹psubst ?f (subc c ?s (sub 0 (Fun n []) p)) =
  subc (?f c) (psubst_term ?f ?s) (psubst ?f (sub 0 (Fun n []) p))›
  using subc_psubst by simp
also have ‹… = subc c s (sub 0 (Fun fresh []) (psubst ?f p))›
  using ‹?f c = c› psubst_sub psubst_s by simp
also have ‹… = subc c s (sub 0 (Fun fresh []) p)›
  using DeExi fresh by simp
finally have psubst_A: ‹psubst ?f (subc c ?s (sub 0 (Fun n []) p)) =
    sub 0 (Fun fresh []) (subc c (inc_term s) p)›
  using ‹new_term c (Fun fresh [])› by simp

have ‹news n z›
```

```
  using DeExi by simp
moreover have ‹news fresh z›
  using fresh by (induct z) simp_all
ultimately have ‹map (psubst ?f) z = z›
  by (induct z) simp_all
moreover have ‹∀x ∈ ⋃p ∈ set z. params p. x ≠ c ⟶ ?f x ≠ ?f c›
  by auto
ultimately have psubst_G: ‹map (psubst ?f) (subcs c ?s z) = subcs c s z›
  using ‹?f c = c› psubst_s by simp

have ‹⊣ subc c ?s (sub 0 (Fun n []) p) # subcs c ?s z›
  using DeExi by simp
then have ‹⊣ psubst ?f (subc c ?s (sub 0 (Fun n []) p)) # map (psubst ?f) (subcs c ?s z)›
  using TC_psubst DeExi.hyps(3) by (metis map_eq_Cons_conv subcs.simps(2))
then have ‹⊣ psubst ?f (subc c ?s (sub 0 (Fun n []) p)) # subcs c s z›
  using psubst_G by simp
then have sub_A: ‹⊣ sub 0 (Fun fresh []) (subc c (inc_term s) p) # subcs c s z›
  using psubst_A by simp

have ‹new_term fresh s›
  using fresh by simp
then have ‹new_term fresh (inc_term s)›
  by simp
then have ‹new fresh (subc c (inc_term s) p)›
  using fresh new_subc by simp
moreover have ‹news fresh (subcs c s z)›
  using ‹news fresh z› ‹new_term fresh s› news_subcs by fast
```

```
    ultimately show ‹⊣ subcs c s (Exi p # z)›
      using TC.DeExi sub_A by simp
  qed
next
  case (DeUni n p z)
  then show ?case
  proof (cases ‹c = n›)
    case True
    then have ‹⊣ Neg (Uni p) # z›
      using DeUni TC.DeUni by blast
    moreover have ‹new c p› and ‹news c z›
      using DeUni True by simp_all
    ultimately show ?thesis
      by simp
  next
    case False
    let ?params = ‹params p ∪ (⋃p ∈ set z. params p) ∪ params_term s ∪ {c} ∪ {n}›

    have ‹finite ?params›
      by simp
    then obtain fresh where fresh: ‹fresh ∉ ?params›
      by (meson ex_new_if_finite infinite_UNIV_listI)

    let ?s = ‹psubst_term (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 simp
then have ‹psubst_term ?f ?s = psubst_term (id(fresh := n)) ?s›
  using new_params' fun_upd_twist(1) psubst_upd'(1) by metis
then have psubst_s: ‹psubst_term ?f ?s = s›
  using fresh psubst_new_away' by simp

have ‹?f c = c› and ‹new_term c (Fun fresh [])›
  using False fresh by auto

have ‹psubst ?f (subc c ?s (Neg (sub 0 (Fun n []) p))) =
  subc (?f c) (psubst_term ?f ?s) (psubst ?f (Neg (sub 0 (Fun n []) p)))›
  using subc_psubst by simp
also have ‹… = subc c s (Neg (sub 0 (Fun fresh []) (psubst ?f p)))›
  using ‹?f c = c› psubst_sub psubst_s by simp
also have ‹… = subc c s (Neg (sub 0 (Fun fresh []) p))›
  using DeUni fresh by simp
finally have psubst_A: ‹psubst ?f (subc c ?s (Neg (sub 0 (Fun n []) p))) =
    Neg (sub 0 (Fun fresh []) (subc c (inc_term s) p))›
  using ‹new_term c (Fun fresh [])› by simp

have ‹news n z›
  using DeUni by simp
moreover have ‹news fresh z›
  using fresh by (induct z) simp_all
```

```
  ultimately have ‹map (psubst ?f) z = z›
    by (induct z) simp_all
  moreover have ‹∀x ∈ ⋃p ∈ set z. params p. x ≠ c ⟶ ?f x ≠ ?f c›
    by auto
  ultimately have psubst_G: ‹map (psubst ?f) (subcs c ?s z) = subcs c s z›
    using ‹?f c = c› psubst_s by simp

  have ‹⊣ subc c ?s (Neg (sub 0 (Fun n []) p)) # subcs c ?s z›
    using DeUni by simp
  then have ‹⊣ psubst ?f (subc c ?s (Neg (sub 0 (Fun n []) p))) # map (psubst ?f) (subcs c ?s z)›
    using TC_psubst DeUni.hyps(3) by (metis map_eq_Cons_conv subcs.simps(2))
  then have ‹⊣ psubst ?f (subc c ?s (Neg (sub 0 (Fun n []) p))) # subcs c s z›
    using psubst_G by simp
  then have sub_A: ‹⊣ Neg (sub 0 (Fun fresh []) (subc c (inc_term s) p)) # subcs c s z›
    using psubst_A by simp

  have ‹new_term fresh s›
    using fresh by simp
  then have ‹new_term fresh (inc_term s)›
    by simp
  then have ‹new fresh (subc c (inc_term s) p)›
    using fresh new_subc by simp
  moreover have ‹news fresh (subcs c s z)›
    using ‹news fresh z› ‹new_term fresh s› news_subcs by fast
  ultimately show ‹⊣ subcs c s (Neg (Uni p) # z)›
    using TC.DeUni sub_A by simp
qed
```

```
qed (auto intro: TC.intros)

lemma TC_map_subc: ‹⊣ z ⟹ ⊣ map (subc c s) z›
  using subcs_map TC_subcs by simp

lemma ex_all_closed: ‹∃m. list_all (closed m) z›
proof (induct z)
  case Nil
  then show ?case
    by simp
next
  case (Cons a z)
  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 :: ‹id list ⇒ fm => fm› where
  ‹sub_consts [] p = p› |
  ‹sub_consts (c # cs) p = sub_consts cs (sub (length cs) (Fun c []) p)›

lemma valid_sub_consts:
  assumes ‹∀(e :: _ ⇒ 'a) f g. semantics e f g p›
  shows ‹semantics (e :: _ => 'a) f g (sub_consts cs p)›
  using assms by (induct cs arbitrary: p) simp_all
```

```
lemma closed_sub' [simp]:
  assumes ‹k ≤ m›
  shows
    ‹closed_term (Suc m) t = closed_term m (sub_term k (Fun c []) t)›
    ‹closed_list (Suc m) l = closed_list m (sub_list k (Fun c []) l)›
  using assms by (induct t and l rule: closed_term.induct closed_list.induct) auto

lemma closed_sub: ‹k ≤ m ⟹ closed (Suc m) p = closed m (sub k (Fun c []) p)›
  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›
  using closed_sub by (induct cs arbitrary: k p) auto

lemma map_sub_consts_Nil: ‹map (sub_consts []) z = z›
  by (induct z) simp_all

primrec conjoin :: ‹fm list ⇒ fm› where
  ‹conjoin [] = Truth› |
  ‹conjoin (p # z) = Con p (conjoin z)›

lemma semantics_conjoin: ‹list_all (semantics e f g) z = semantics e f g (conjoin z)›
  by (induct z) simp_all

lemma valid_sub:
  fixes e :: ‹nat ⇒ 'a›
  assumes ‹∀(e :: _ ⇒ 'a) f g. semantics e f g p ⟶ semantics e f g q›
  shows ‹semantics e f g (sub m t p) ⟶ semantics e f g (sub m t q)›
```

```
  using assms by simp

lemma semantics_sub_consts:
  fixes e :: ‹nat ⇒ 'a›
  assumes ‹∀(e :: _ ⇒ 'a) f g. semantics e f g p ⟶ semantics e f g q›
    and ‹semantics e f g (sub_consts cs p)›
  shows ‹semantics e f g (sub_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
    using substitute by (metis sub_consts.simps(2))
qed

lemma sub_consts_Con [simp]: ‹sub_consts cs (Con p q) = Con (sub_consts cs p) (sub_consts cs q)›
  by (induct cs arbitrary: p q) simp_all

lemma sub_consts_conjoin:
  ‹semantics e f g (sub_consts cs (conjoin z)) = semantics e f g (conjoin (map (sub_consts cs) z))›
proof (induct z)
  case Nil
  then show ?case
    by (induct cs) simp_all
```

```
next
  case (Cons p z)
  then show ?case
    using sub_consts_Con by simp
qed

lemma all_sub_consts_conjoin:
  ‹list_all (semantics e f g) (map (sub_consts cs) z) = semantics e f g (sub_consts cs (conjoin z))›
  by (induct z) (simp_all add: valid_sub_consts)

lemma valid_all_sub_consts:
  fixes e :: ‹nat ⇒ 'a›
  assumes ‹∀(e :: _ ⇒ 'a) f g. list_all (semantics e f g) z ⟶ semantics e f g p›
  shows ‹list_all (semantics e f g) (map (sub_consts cs) z) ⟶ semantics e f g (sub_consts cs p)›
  using assms semantics_conjoin semantics_sub_consts all_sub_consts_conjoin by metis

lemma TC_vars_for_consts: ‹⊣ z ⟹ ⊣ map (λp. vars_for_consts p cs) z›
proof (induct cs)
  case Nil
  then show ?case
    by simp
next
  case (Cons c cs)
  have ‹(⊣ map (λp. vars_for_consts p (c # cs)) z) =
      (⊣ map (λp. subc c (Var (length cs)) (vars_for_consts p cs)) z)›
    by simp
  also have ‹... = (⊣ map (subc c (Var (length cs)) o (λp. vars_for_consts p cs)) z)›
```

```
    unfolding comp_def by simp
  also have ‹… = (⊣ map (subc c (Var (length cs))) (map (λp. vars_for_consts p cs) z))›
    by simp
  finally show ?case
    using Cons TC_map_subc by blast
qed

lemma vars_for_consts_sub_consts:
  ‹closed (length cs) p ⟹ list_all (λc. new c p) cs ⟹ distinct cs ⟹
   vars_for_consts (sub_consts cs p) cs = p›
  using sub_new_all closed_sub
  by (induct cs arbitrary: p) (auto simp: list_all_def)

lemma all_vars_for_consts_sub_consts:
  ‹list_all (closed (length cs)) z ⟹ list_all (λc. list_all (new c) z) cs ⟹ distinct cs ⟹
   map (λp. vars_for_consts p cs) (map (sub_consts cs) z) = z›
  using vars_for_consts_sub_consts unfolding list_all_def
  by (induct z) simp_all

lemma new_conjoin: ‹new c (conjoin z) ⟹ list_all (new c) z›
  by (induct z) simp_all

lemma all_fresh_constants:
  ‹∃cs. length cs = m ∧ list_all (λc. list_all (new c) z) cs ∧ distinct cs›
proof -
  obtain cs where ‹length cs = m› ‹list_all (λc. new c (conjoin z)) cs› ‹distinct cs›
    using fresh_constants by blast
```

```
  then show ?thesis
    using new_conjoin unfolding list_all_def by blast
qed

lemma sub_consts_Neg: ‹sub_consts cs (Neg p) = Neg (sub_consts cs p)›
  by (induct cs arbitrary: p) simp_all

lemma sub_compl: ‹sub m t (compl p) = compl (sub m t p)›
proof (induct p arbitrary: m t)
  case (Imp p1 p2)
  then show ?case
  proof (cases ‹p2 = Falsity›)
    case True
    then show ?thesis
      using Imp by simp
  next
    case False
    then have ‹sub m t p2 ≠ Falsity›
      by (induct p2) simp_all
    have ‹sub m t (compl (Imp p1 p2)) = sub m t (Neg (Imp p1 p2))›
      using False compl by (metis fm.inject(2))
    also have ‹… = Neg (Imp (sub m t p1) (sub m t p2))›
      by simp
    also have ‹… = compl (Imp (sub m t p1) (sub m t p2))›
      using ‹sub m t p2 ≠ Falsity› compl by (metis fm.inject(2))
    finally show ?thesis
      by simp
```

```
  qed
qed simp_all

lemma sub_consts_compl: ‹sub_consts cs (compl p) = compl (sub_consts cs p)›
  by (induct cs arbitrary: p) (simp_all add: sub_compl)

subsection ‹Completeness›

theorem tableau_completeness:
  assumes ‹∀(e :: _ ⇒ htm) f g. list_all (semantics e f g) z ⟶ semantics e f g p›
  shows ‹tableauproof z p›
proof -
  obtain m where *: ‹list_all (closed m) (p # z)›
    using ex_all_closed by blast
  moreover obtain cs :: ‹id list› where **:
    ‹length cs = m›
    ‹distinct cs›
    ‹list_all (λc. list_all (new c) (p # z)) cs›
    using all_fresh_constants by blast
  ultimately have ‹sentence (sub_consts cs p)›
    using closed_sub_consts by simp
  moreover have ‹list_all sentence (map (sub_consts cs) z)›
    using closed_sub_consts * ‹length cs = m› by (induct z) simp_all

  moreover have ‹∀(e :: _ ⇒ htm) f g. list_all (semantics e f g) (map (sub_consts cs) z) ⟶
    semantics e f g (sub_consts cs p)›
    using assms valid_all_sub_consts by blast
```

```
  ultimately have ‹⊣ compl (sub_consts cs p) # map (sub_consts cs) z›
    using tableau_completeness' unfolding tableauproof_def by simp
  then have ‹⊣ map (sub_consts cs) (compl p # z)›
    using sub_consts_compl by simp
  then have ‹⊣ map (λp. vars_for_consts p cs) (map (sub_consts cs) (compl p # z))›
    using TC_vars_for_consts by blast
  moreover have ‹list_all (closed (length cs)) (compl p # z)›
    using * ** closed_compl by auto
  moreover have ‹list_all (λc. list_all (new c) (compl p # z)) cs›
    using ** new_compl unfolding list_all_def by simp
  ultimately have ‹⊣ compl p # z›
    using all_vars_for_consts_sub_consts[where z=‹compl p # z›] ** by simp
  then show ?thesis
    unfolding tableauproof_def .
qed

theorem complete:
  assumes ‹∀(e :: _ ⇒ htm) f g. semantics e f g p›
  shows ‹⊣ [compl p]›
  using assms tableau_completeness unfolding tableauproof_def by simp

lemma TC_compl_compl:
  assumes ‹⊣ compl (compl p) # z›
  shows ‹⊣ p # z›
proof -
  have ‹∃p ∈ set (compl (compl p) # z). ¬ semantics e f g p› for e :: ‹nat ⇒ htm› and f g
    using TC_soundness assms by blast
```

```
  then have ‹list_all (semantics e f g) z ⟶ semantics e f g (compl p)›
    for e :: ‹nat ⇒ htm› and f g
    unfolding list_all_def using compl by (metis semantics.simps(3) set_ConsD)
  then obtain q where
    ‹compl q = p›
    ‹∀(e :: _ ⇒ htm) f g. list_all (semantics e f g) z ⟶ semantics e f g q›
    using compl_compl by (metis compl.simps(1))
  then have ‹⊣ compl q # z›
    using tableau_completeness unfolding tableauproof_def by blast
  then show ?thesis
    using ‹compl q = p› by blast
qed

lemma TC_Neg_Neg:
  assumes ‹⊣ Neg (Neg p) # z›
  shows ‹⊣ p # z›
proof -
  have ‹∃p ∈ set (Neg (Neg p) # z). ¬ semantics e f g p› for e :: ‹nat ⇒ htm› and f g
    using TC_soundness assms by blast
  then have ‹list_all (semantics e f g) z ⟶ semantics e f g (compl p)›
    for e :: ‹nat ⇒ htm› and f g
    unfolding list_all_def using compl by (metis semantics.simps(3) set_ConsD)
  then obtain q where
    ‹compl q = p›
    ‹∀(e :: _ ⇒ htm) f g. list_all (semantics e f g) z ⟶ semantics e f g q›
    using compl_compl by (metis compl.simps(1))
  then have ‹⊣ compl q # z›
```

```
    using tableau_completeness unfolding tableauproof_def by blast
  then show ?thesis
    using ‹compl q = p› by blast
qed

lemma TC_complete:
  assumes ‹∀(e :: _ ⇒ htm) f g. ∃p ∈ set z. ¬ semantics e f g p›
  shows ‹⊣ z›
proof -
  have ‹∀(e :: _ ⇒ htm) f g. ∃p ∈ set z. semantics e f g (compl p)›
    using assms semantics_compl by fast
  then obtain p where
    ‹p ∈ set z›
    ‹∀(e :: _ ⇒ htm) f g. (∀p ∈ set z. (semantics e f g) p) ⟶ semantics e f g (compl p)›
    using assms by blast
  then have ‹⊣ compl (compl p) # z›
    using tableau_completeness Ball_set unfolding tableauproof_def by metis
  then have ‹⊣ p # z›
    using TC_compl_compl by simp
  with ‹p ∈ set z› show ?thesis
    using TC.Extra member_set by simp
qed

lemma Order': ‹⊣ z ⟹ set z ⊆ set G' ⟹ ⊣ G'›
  using TC_soundness in_mono TC_complete by metis

lemma Swap': ‹⊣ q # p # z ⟹ ⊣ p # q # z›
```

```
  using Order' by (simp add: insert_commute)

lemma AlNegNeg': ‹⊣ p # z ⟹ ⊣ Neg (Neg p) # z›
  using AlImp Order' Swap' set_subset_Cons by metis

section ‹Sequent Calculus› — ‹NaDeA variant›

inductive SC :: ‹fm list ⇒ bool› (‹⊢ _› 0) where
  Dummy: ‹⊢ Truth # z› |
  Basic: ‹⊢ Pre i l # Neg (Pre i l) # z› |
  AlImp: ‹⊢ Neg p # q # z ⟹ ⊢ Imp p q # z› |
  AlDis: ‹⊢ p # q # z ⟹ ⊢ Dis p q # z› |
  AlCon: ‹⊢ Neg p # Neg q # z ⟹ ⊢ Neg (Con p q) # z› |
  BeImp: ‹⊢ p # z ⟹ ⊢ Neg q # z ⟹ ⊢ Neg (Imp p q) # z› |
  BeDis: ‹⊢ Neg p # z ⟹ ⊢ Neg q # z ⟹ ⊢ Neg (Dis p q) # z› |
  BeCon: ‹⊢ p # z ⟹ ⊢ q # z ⟹ ⊢ Con p q # z› |
  GaExi: ‹⊢ sub 0 t p # z ⟹ ⊢ Exi p # z› |
  GaUni: ‹⊢ Neg (sub 0 t p) # z ⟹ ⊢ Neg (Uni p) # z› |
  DeExi: ‹⊢ Neg (sub 0 (Fun c []) p) # z ⟹ news c (p # z) ⟹ ⊢ Neg (Exi p) # z› |
  DeUni: ‹⊢ sub 0 (Fun c []) p # z ⟹ news c (p # z) ⟹ ⊢ Uni p # z› |
  Extra: ‹⊢ p # z ⟹ member p z ⟹ ⊢ z›

lemma AlNegNeg: ‹⊢ p # z ⟹ ⊢ Neg (Neg p) # z›
proof -
  assume ‹⊢ p # z›
  with BeImp show ?thesis
    using Dummy .
```

```
qed

lemma psubst_new_sub':
  ‹new_term n t ⟹ psubst_term (id(n := m)) (sub_term k (Fun n []) t) = sub_term k (Fun m []) t›
  ‹new_list n l ⟹ psubst_list (id(n := m)) (sub_list k (Fun n []) l) = sub_list k (Fun m []) l›
  by (induct t and l rule: sub_term.induct sub_list.induct) auto

lemma psubst_new_sub: ‹new n p ⟹ psubst (id(n := m)) (sub k (Fun n []) p) = sub k (Fun m []) p›
  using psubst_new_sub' by (induct p) simp_all

lemma SC_psubst: ‹⊢ z ⟹ ⊢ map (psubst f) z›
proof (induct arbitrary: f rule: SC.induct)
  case (DeUni n p z)
  let ?params = ‹params p ∪ (⋃p ∈ set z. params p)›

  have ‹finite ?params›
    by simp
  then obtain m where *: ‹m ∉ ?params ∪ {n} ∪ image f ?params›
    using ex_new_if_finite
    by (metis finite.emptyI finite.insertI finite_UnI finite_imageI infinite_UNIV_listI)

  let ?f = ‹f(n := m)›
  let ?G = ‹map (psubst ?f) z›

  have z: ‹?G = map (psubst f) z›
    using ‹news n (p # z)› by (induct z) simp_all
```

```
  have ‹⊢ psubst ?f (sub 0 (Fun n []) p) # ?G›
    using DeUni by (metis Cons_eq_map_conv)
  then have ‹⊢ sub 0 (Fun m []) (psubst f p) # ?G›
    using ‹news n (p # z)› by simp
  moreover have ‹news m (psubst f p # ?G)›
    using DeUni * Un_iff image_Un new_params news.simps(2) news_psubst psubst_image by metis
  ultimately have ‹⊢ psubst f (Uni p) # ?G›
    using SC.DeUni by simp
  then show ?case
    using z by simp
next
  case (DeExi n p z)
  let ?params = ‹params p ∪ (⋃p ∈ set z. params p)›

  have ‹finite ?params›
    by simp
  then obtain m where *: ‹m ∉ ?params ∪ {n} ∪ image f ?params›
    using ex_new_if_finite
    by (metis finite.emptyI finite.insertI finite_UnI finite_imageI infinite_UNIV_listI)

  let ?f = ‹f(n := m)›
  let ?G = ‹map (psubst ?f) z›

  have z: ‹?G = map (psubst f) z›
    using ‹news n (p # z)› by (induct z) simp_all

  have ‹⊢ psubst ?f (Neg (sub 0 (Fun n []) p)) # ?G›
```

```
    using DeExi by (metis Cons_eq_map_conv)
  then have ‹⊢ Neg (sub 0 (Fun m []) (psubst f p)) # ?G›
    using ‹news n (p # z)› by simp
  moreover have ‹news m (psubst f p # ?G)›
    using DeExi * Un_iff image_Un new_params news.simps(2) news_psubst psubst_image by metis
  ultimately have ‹⊢ psubst f (Neg (Exi p)) # ?G›
    using SC.DeExi by simp
  then show ?case
    using z by simp
next
  case (Extra p z)
  then show ?case
    using SC.Extra member_psubst by fastforce
qed (auto intro: SC.intros)

lemma psubst_swap_twice':
  ‹psubst_term (id(n := m, m := n)) (psubst_term (id(n := m, m := n)) t) = t›
  ‹psubst_list (id(n := m, m := n)) (psubst_list (id(n := m, m := n)) l) = l›
  by (induct t and l rule: psubst_term.induct psubst_list.induct) simp_all

lemma psubst_swap_twice:
  ‹psubst (id(n := m, m := n)) (psubst (id(n := m, m := n)) p) = p›
  using psubst_swap_twice' by (induct p arbitrary: m n) simp_all

lemma Order: ‹⊢ z ⟹ set z ⊆ set G' ⟹ ⊢ G'›
proof (induct arbitrary: G' rule: SC.induct)
  case (Basic i l z)
```

```
  then show ?case
    using SC.Basic Extra member_set
    by (metis list.set_intros(1) set_subset_Cons subsetCE)
next
  case (Dummy z)
  then show ?case
    using SC.Dummy Extra member_set
    by (metis list.set_intros(1) subsetCE)
next
  case (AlCon p q z)
  then have ‹⊢ Neg p # Neg q # G'›
    by (metis order_trans set_subset_Cons subset_cons)
  then have ‹⊢ Neg (Con p q) # G'›
    using SC.AlCon by simp
  then show ?case
    using Extra AlCon
    by (metis list.set_intros(1) member_set subsetCE)
next
  case (AlDis p q z)
  then have ‹⊢ p # q # G'›
    by (metis order_trans set_subset_Cons subset_cons)
  then have ‹⊢ Dis p q # G'›
    using SC.AlDis by simp
  then show ?case
    using Extra AlDis
    by (metis list.set_intros(1) member_set subsetCE)
next
```

```
  case (AlImp p q z)
  then have ‹⊢ Neg p # q # G'›
    by (metis order_trans set_subset_Cons subset_cons)
  then have ‹⊢ Imp p q # G'›
    using SC.AlImp by simp
  then show ?case
    using Extra AlImp
    by (metis list.set_intros(1) member_set subsetCE)
next
  case (BeCon p z q)
  then have ‹⊢ p # G'› ‹⊢ q # G'›
    by (metis order_trans set_subset_Cons subset_cons)+
  then have ‹⊢ Con p q # G'›
    using SC.BeCon by simp
  then show ?case
    using Extra BeCon
    by (metis list.set_intros(1) member_set subsetCE)
next
  case (BeDis p z q)
  then have ‹⊢ Neg p # G'› ‹⊢ Neg q # G'›
    by (metis order_trans set_subset_Cons subset_cons)+
  then have ‹⊢ Neg (Dis p q) # G'›
    using SC.BeDis by simp
  then show ?case
    using Extra BeDis
    by (metis list.set_intros(1) member_set subsetCE)
next
```

```
  case (BeImp p z q)
  then have ‹⊢ p # G'› ‹⊢ Neg q # G'›
    by (metis order_trans set_subset_Cons subset_cons)+
  then have ‹⊢ Neg (Imp p q) # G'›
    using SC.BeImp by simp
  then show ?case
    using Extra BeImp
    by (metis list.set_intros(1) member_set subsetCE)
next
  case (GaExi t p z)
  then have ‹⊢ sub 0 t p # G'›
    by (metis order_trans set_subset_Cons subset_cons)
  then have ‹⊢ Exi p # G'›
    using SC.GaExi by simp
  then show ?case
    using Extra GaExi
    by (metis list.set_intros(1) member_set subsetCE)
next
  case (GaUni t p z)
  then have ‹⊢ Neg (sub 0 t p) # G'›
    by (metis order_trans set_subset_Cons subset_cons)
  then have ‹⊢ Neg (Uni p) # G'›
    using SC.GaUni by simp
  then show ?case
    using Extra GaUni
    by (metis list.set_intros(1) member_set subsetCE)
next
```

```
case (DeUni n p z)
then obtain m where ‹news m (p # G')›
  using allnew fresh_constant new_conjoin by blast
then have ‹news m (p # z)›
  using DeUni Ball_set allnew news.simps(2) subset_code(1) set_subset_Cons by metis

let ?f = ‹id(n := m, m := n)›
let ?A = ‹psubst ?f (sub 0 (Fun n []) p)›
have p: ‹?A = sub 0 (Fun m []) p›
  using ‹news n (p # z)› ‹news m (p # G')› psubst_new_sub by simp

let ?G' = ‹map (psubst ?f) G'›
have G': ‹map (psubst ?f) ?G' = G'›
  using psubst_swap_twice by (induct G') (simp, metis list.simps(9))

have ‹set z ⊆ set G'›
  using DeUni by simp
then have ‹set z ⊆ set ?G'›
  using ‹news n (p # z)› ‹news m (p # z)›
proof (induct z)
  case (Cons a z)
  then have ‹psubst ?f a = a›
    by simp
  then show ?case
    using Cons by (metis image_eqI insert_subset list.set(2) news.simps(2) set_map)
qed simp
```

```
  have ‹⊢ sub 0 (Fun n []) p # ?G'›
    using ‹set z ⊆ set ?G'› DeUni by (metis subset_cons)
  then have ‹⊢ ?A # map (psubst ?f) ?G'›
    using SC_psubst by (metis map_eq_Cons_conv)
  then have ‹⊢ sub 0 (Fun m []) p # G'›
    using p G' by simp
  then have ‹⊢ Uni p # G'›
    using SC.DeUni ‹news m (p # G')› by blast
  then show ?case
    using Extra ‹set (Uni p # z) ⊆ set G'› by simp
next
  case (DeExi n p z)
  then obtain m where ‹news m (p # G')›
    using allnew fresh_constant new_conjoin by blast
  then have ‹news m (p # z)›
    using DeExi Ball_set allnew news.simps(2) subset_code(1) set_subset_Cons by metis

  let ?f = ‹id(n := m, m := n)›
  let ?A = ‹psubst ?f (Neg (sub 0 (Fun n []) p))›
  have p: ‹?A = Neg (sub 0 (Fun m []) p)›
    using ‹news n (p # z)› ‹news m (p # G')› psubst_new_sub by simp

  let ?G' = ‹map (psubst ?f) G'›
  have G': ‹map (psubst ?f) ?G' = G'›
    using psubst_swap_twice by (induct G') (simp, metis list.simps(9))

  have ‹set z ⊆ set G'›
```

```
    using DeExi by simp
  then have ‹set z ⊆ set ?G'›
    using ‹news n (p # z)› ‹news m (p # z)›
  proof (induct z)
    case (Cons a z)
    then have ‹psubst ?f a = a›
      by simp
    then show ?case
      using Cons by (metis image_eqI insert_subset list.set(2) news.simps(2) set_map)
  qed simp

  have ‹⊢ Neg (sub 0 (Fun n []) p) # ?G'›
    using ‹set z ⊆ set ?G'› DeExi by (metis subset_cons)
  then have ‹⊢ ?A # map (psubst ?f) ?G'›
    using SC_psubst by (metis Cons_eq_map_conv)
  then have ‹⊢ Neg (sub 0 (Fun m []) p) # G'›
    using p G' by simp
  then have ‹⊢ Neg (Exi p) # G'›
    using SC.DeExi ‹news m (p # G')› by blast
  then show ?case
    using Extra ‹set (Neg (Exi p) # z) ⊆ set G'› by simp
next
  case (Extra p z)
  then show ?case
    using SC.Extra member_set
    by (metis set_rev_mp subset_cons)
qed
```

```
corollary ‹⊢ z ⟹ set z = set G' ⟹ ⊢ G'›
  using Order by simp

lemma Shift: ‹⊢ rotate1 z ⟹ ⊢ z›
  using Order by simp

lemma Swap: ‹⊢ q # p # z ⟹ ⊢ p # q # z›
  using Order by (simp add: insert_commute)

lemma ‹⊢ [Neg (Pre "A" []) , Pre "A" []]›
  by (rule Shift, simp) (rule Basic)

lemma ‹⊢ [Con (Pre "A" []) (Pre "B" []), Neg (Con (Pre "B" []) (Pre "A" []))]›
  apply (rule BeCon)
   apply (rule Swap)
   apply (rule AlCon)
   apply (rule Shift, simp, rule Swap)
   apply (rule Basic)
  apply (rule Swap)
  apply (rule AlCon)
  apply (rule Shift, rule Shift, simp)
  apply (rule Basic)
  done

subsection ‹Soundness›
```

```
lemma SC_soundness: ‹⊢ z ⟹ ∃p ∈ set z. semantics e f g p›
proof (induct arbitrary: f rule: SC.induct)
  case (Extra p z)
  then show ?case
    by fastforce
next
  case (DeUni n p z)
  then consider
    ‹∀x. semantics e (f(n := λw. x)) g (sub 0 (Fun n []) p)› |
    ‹∃x. ∃p ∈ set (Uni p # z). semantics e (f(n := λw. x)) g p›
    by fastforce
  then show ?case
  proof cases
    case 1
    then have ‹∀x. semantics (put e 0 x) (f(n := λw. x)) g p›
      by simp
    then have ‹∀x. semantics (put e 0 x) f g p›
      using ‹news n (p # z)› by simp
    then show ?thesis
      by simp
  next
    case 2
    then show ?thesis
      using ‹news n (p # z)›
      by (metis Ball_set allnew map new.simps(7) news.simps(2))
  qed
next
```

```
  case (DeExi n p z)
  then consider
    ‹∀x. semantics e (f(n := λw. x)) g (Neg (sub 0 (Fun n []) p))› |
    ‹∃x. ∃p ∈ set (Neg (Exi p) # z). semantics e (f(n := λw. x)) g p›
    by fastforce
  then show ?case
  proof cases
    case 1
    then have ‹∀x. semantics (put e 0 x) (f(n := λw. x)) g (Neg p)›
      by simp
    then have ‹∀x. semantics (put e 0 x) f g (Neg p)›
      using ‹news n (p # z)› by simp
    then show ?thesis
      by simp
  next
    case 2
    then show ?thesis
      using ‹news n (p # z)›
      by (metis Ball_set allnew map new.simps(1,3,6) news.simps(2))
  qed
qed auto

subsection ‹Tableau Equivalence›

lemma SC_remove_Falsity: ‹⊢ z ⟹ set z - {Falsity} ⊆ set G' ⟹ ⊢ G'›
proof (induct z arbitrary: G' rule: SC.induct)
  case (Basic i l z)
```

```
  then have ‹{Pre i l, Neg (Pre i l)} ∪ (set z - {Falsity}) ⊆ set G'›
    using subset_insert_iff by auto
  then show ?case
    using SC.Basic Order by fastforce
next
  case (Dummy z)
  then have ‹{Truth} ∪ (set z - {Falsity}) ⊆ set G'›
    using subset_insert_iff by auto
  then show ?case
    using SC.Dummy Order by fastforce
next
  case (AlCon p q z)
  then have *: ‹Neg (Con p q) ∈ set G'›
    by auto

  have ‹set (Neg p # Neg q # z) - {Falsity} ⊆ set (Neg p # Neg q # G')›
    using AlCon by auto
  then have ‹⊢ Neg p # Neg q # G'›
    using AlCon by simp
  then have ‹⊢ Neg (Con p q) # G'›
    using SC.AlCon by blast
  then show ?case
    using * Extra by simp
next
  case (AlDis p q z)
  then have *: ‹Dis p q ∈ set G'›
    by auto
```

```
  have ‹set (p # q # z) - {Falsity} ⊆ set (p # q # G')›
    using AlDis by auto
  then have ‹⊢ p # q # G'›
    using AlDis by simp
  then have ‹⊢ Dis p q # G'›
    using SC.AlDis by blast
  then show ?case
    using * Extra by simp
next
  case (AlImp p q z)
  then have *: ‹Imp p q ∈ set G'›
    by auto

  have ‹set (Neg p # q # z) - {Falsity} ⊆ set (Neg p # q # G')›
    using AlImp by auto
  then have ‹⊢ Neg p # q # G'›
    using AlImp by simp
  then have ‹⊢ Imp p q # G'›
    using SC.AlImp by blast
  then show ?case
    using * Extra by simp
next
  case (BeCon p z q)
  then have *: ‹Con p q ∈ set G'›
    by auto
```

```
  have ‹set (p # z) - {Falsity} ⊆ set (p # G')› ‹set (q # z) - {Falsity} ⊆ set (q # G')›
    using BeCon by auto
  then have ‹⊢ p # G'› ‹⊢ q # G'›
    using BeCon by simp_all
  then have ‹⊢ Con p q # G'›
    using SC.BeCon by blast
  then show ?case
    using * Extra by simp
next
  case (BeDis p z q)
  then have *: ‹Neg (Dis p q) ∈ set G'›
    by auto

  have
    ‹set (Neg p # z) - {Falsity} ⊆ set (Neg p # G')›
    ‹set (Neg q # z) - {Falsity} ⊆ set (Neg q # G')›
    using BeDis by auto
  then have ‹⊢ Neg p # G'› ‹⊢ Neg q # G'›
    using BeDis by simp_all
  then have ‹⊢ Neg (Dis p q) # G'›
    using SC.BeDis by blast
  then show ?case
    using * Extra by simp
next
  case (BeImp p z q)
  then have *: ‹Neg (Imp p q) ∈ set G'›
    by auto
```

```
  have ‹set (p # z) - {Falsity} ⊆ set (p # G')› ‹set (Neg q # z) - {Falsity} ⊆ set (Neg q # G')›
    using BeImp by auto
  then have ‹⊢ p # G'› ‹⊢ Neg q # G'›
    using BeImp by simp_all
  then have ‹⊢ Neg (Imp p q) # G'›
    using SC.BeImp by blast
  then show ?case
    using * Extra by simp
next
  case (GaExi t p z)
  then have *: ‹Exi p ∈ set G'›
    by auto

  have ‹set (sub 0 t p # z) - {Falsity} ⊆ set (sub 0 t p # G')›
    using GaExi by auto
  then have ‹⊢ sub 0 t p # G'›
    using GaExi by simp
  then have ‹⊢ Exi p # G'›
    using SC.GaExi by blast
  then show ?case
    using * Extra by simp
next
  case (GaUni t p z)
  then have *: ‹Neg (Uni p) ∈ set G'›
    by auto
```

```
  have ‹set (Neg (sub 0 t p) # z) - {Falsity} ⊆ set (Neg (sub 0 t p) # G')›
    using GaUni by auto
  then have ‹⊢ Neg (sub 0 t p) # G'›
    using GaUni by simp
  then have ‹⊢ Neg (Uni p) # G'›
    using SC.GaUni by blast
  then show ?case
    using * Extra by simp
next
  case (DeUni n p z)
  let ?params = ‹params p ∪ (⋃p ∈ set z. params p) ∪ (⋃p ∈ set G'. params p)›

  have ‹finite ?params›
    by simp
  then obtain m where *: ‹m ∉ ?params ∪ {n}›
    using ex_new_if_finite by (metis finite.emptyI finite.insertI finite_UnI infinite_UNIV_listI)

  let ?f = ‹id(n := m, m := n)›
  let ?A = ‹sub 0 (Fun m []) p›
  let ?G' = ‹map (psubst ?f) G'›

  have p: ‹psubst ?f (sub 0 (Fun n []) p) = ?A›
    using ‹news n (p # z)› * by simp
  have G': ‹map (psubst ?f) ?G' = G'›
    using psubst_swap_twice by (induct G') simp_all

  have ‹set z - {Falsity} ⊆ set G'›
```

```
    using DeUni by auto
  then have ‹set (map (psubst ?f) z) - {Falsity} ⊆ set ?G'›
    by auto
  moreover have ‹map (psubst ?f) z = z›
    using ‹news n (p # z)› * by (induct z) simp_all
  ultimately have ‹set z - {Falsity} ⊆ set ?G'›
    by simp

  then have ‹set (sub 0 (Fun n []) p # z) - {Falsity} ⊆ set (sub 0 (Fun n []) p # ?G')›
    by auto
  then have ‹⊢ sub 0 (Fun n []) p # ?G'›
    using * DeUni by simp
  then have ‹⊢ sub 0 (Fun n []) p # ?G'›
    using Swap by simp
  then have ‹⊢ map (psubst ?f) (sub 0 (Fun n []) p # ?G')›
    using SC_psubst by blast
  then have ‹⊢ sub 0 (Fun m []) p # G'›
    using p G' by simp
  moreover have ‹news m (p # G')›
    using * by (induct G') simp_all
  ultimately have ‹⊢ Uni p # G'›
    using SC.DeUni by blast
  moreover have ‹Uni p ∈ set G'›
    using DeUni by auto
  ultimately show ?case
    using Extra by simp
next
```

```
case (DeExi n p z)
let ?params = ‹params p ∪ (⋃p ∈ set z. params p) ∪ (⋃p ∈ set G'. params p)›

have ‹finite ?params›
  by simp
then obtain m where *: ‹m ∉ ?params ∪ {n}›
  using ex_new_if_finite by (metis finite.emptyI finite.insertI finite_UnI infinite_UNIV_listI)

let ?f = ‹id(n := m, m := n)›
let ?A = ‹sub 0 (Fun m []) p›
let ?G' = ‹map (psubst ?f) G'›

have p: ‹psubst ?f (sub 0 (Fun n []) p) = ?A›
  using ‹news n (p # z)› * by simp
have G': ‹map (psubst ?f) ?G' = G'›
  using psubst_swap_twice by (induct G') simp_all

have ‹set z - {Falsity} ⊆ set G'›
  using DeExi by auto
then have ‹set (map (psubst ?f) z) - {Falsity} ⊆ set ?G'›
  by auto
moreover have ‹map (psubst ?f) z = z›
  using ‹news n (p # z)› * by (induct z) simp_all
ultimately have ‹set z - {Falsity} ⊆ set ?G'›
  by simp

then have ‹set (Neg (sub 0 (Fun n []) p) # z) - {Falsity} ⊆ set (Neg (sub 0 (Fun n []) p) # ?G')›
```

```
    by auto
  then have ‹⊢ Neg (sub 0 (Fun n []) p) # ?G'›
    using * DeExi by simp
  then have ‹⊢ Neg (sub 0 (Fun n []) p) # ?G'›
    using Swap by simp
  then have ‹⊢ map (psubst ?f) (Neg (sub 0 (Fun n []) p) # ?G')›
    using SC_psubst by blast
  then have ‹⊢ Neg (sub 0 (Fun m []) p) # G'›
    using p G' by simp
  moreover have ‹news m (p # G')›
    using * by (induct G') simp_all
  ultimately have ‹⊢ Neg (Exi p) # G'›
    using SC.DeExi by blast
  moreover have ‹Neg (Exi p) ∈ set G'›
    using DeExi by auto
  ultimately show ?case
    using Extra by simp
next
  case (Extra p z)
  then show ?case
    by fastforce
qed

lemma special: ‹⊢ z ⟹ Neg (Neg X) ∈ set z ⟹ set z - {Neg (Neg X)} ⊆ set G' ⟹ ⊢ X # G'›
proof (induct z arbitrary: G' rule: SC.induct)
  case (Basic i l z)
  then obtain G'' where *: ‹set G' = set (Pre i l # Neg (Pre i l) # G'')›
```

```
    by auto
  then have ‹⊢ Pre i l # Neg (Pre i l) # G''›
    using SC.Basic by simp
  then show ?case
    using Order * by (metis set_subset_Cons)
next
  case (Dummy z)
  then obtain G'' where *: ‹set G' = set (Truth # G'')›
    by auto
  then have ‹⊢ Truth # G''›
    using SC.Dummy by simp
  then show ?case
    using Order * by (metis set_subset_Cons)
next
  case (AlCon p q z)
  then have *: ‹Neg (Neg X) ∈ set (Neg p # Neg q # z)›
    by auto
  then have ‹set (Neg p # Neg q # z) - {Neg (Neg X)} ⊆ set (Neg p # Neg q # G')›
    using AlCon by auto
  then have ‹⊢ X # Neg p # Neg q # G'›
    using * AlCon by blast
  then have ‹⊢ Neg p # Neg q # X # G'›
    using Order by (simp add: insert_commute)
  then have ‹⊢ Neg (Con p q) # X # G'›
    using SC.AlCon by blast
  moreover have ‹Neg (Con p q) ∈ set G'›
    using AlCon by auto
```

```
  ultimately show ?case
    using Extra by simp
next
  case (AlDis p q z)
  then have *: ‹Neg (Neg X) ∈ set (p # q # z)›
    by auto
  then have ‹set (p # q # z) - {Neg (Neg X)} ⊆ set (p # q # G')›
    using AlDis by auto
  then have ‹⊢ X # p # q # G'›
    using * AlDis by blast
  then have ‹⊢ p # q # X # G'›
    using Order by (simp add: insert_commute)
  then have ‹⊢ Dis p q # X # G'›
    using SC.AlDis by blast
  moreover have ‹Dis p q ∈ set G'›
    using AlDis by auto
  ultimately show ?case
    using Extra by simp
next
  case (AlImp p q z)
  then have *: ‹Neg (Neg X) ∈ set (Neg p # q # z)›
    by auto
  show ?case
  proof (cases ‹Imp p q = Neg (Neg X)›)
    case True
    then have ‹set (Neg p # q # z) - {Neg (Neg X)} ⊆ set (Falsity # G')›
      using AlImp by auto
```

```
    then have ‹⊢ X # Falsity # G'›
      using AlImp * by blast
    then show ?thesis
      using SC_remove_Falsity Swap
      by (metis eq_refl list.set_intros(1) list.simps(15) subset_insert_iff)
  next
    case False
    then have ‹set (Neg p # q # z) - {Neg (Neg X)} ⊆ set (Neg p # q # G')›
      using AlImp by auto
    then have ‹⊢ X # Neg p # q # G'›
      using * AlImp by blast
    then have ‹⊢ Neg p # q # X # G'›
      using Order by (simp add: insert_commute)
    then have ‹⊢ Imp p q # X # G'›
      using SC.AlImp by blast
    moreover have ‹Imp p q ∈ set G'›
      using False AlImp by auto
    ultimately show ?thesis
      using Extra by simp
  qed
next
  case (BeCon p z q)
  then have ‹Neg (Neg X) ∈ set (p # z)› ‹Neg (Neg X) ∈ set (q # z)›
    by auto
  moreover have
    ‹set (p # z) - {Neg (Neg X)} ⊆ set (p # G')›
    ‹set (q # z) - {Neg (Neg X)} ⊆ set (q # G')›
```

```
    using BeCon by auto
  ultimately have ‹⊢ X # p # G'› ‹⊢ X # q # G'›
    using BeCon by blast+
  then have ‹⊢ p # X # G'› ‹⊢ q # X # G'›
    by (simp_all add: Swap)
  then have ‹⊢ Con p q # X # G'›
    using SC.BeCon by blast
  moreover have ‹Con p q ∈ set G'›
    using BeCon by auto
  ultimately show ?case
    using Extra by simp
next
  case (BeDis p z q)
  then have ‹Neg (Neg X) ∈ set (Neg p # z)› ‹Neg (Neg X) ∈ set (Neg q # z)›
    using BeImp by auto
  moreover have
    ‹set (Neg p # z) - {Neg (Neg X)} ⊆ set (Neg p # G')›
    ‹set (Neg q # z) - {Neg (Neg X)} ⊆ set (Neg q # G')›
    using BeDis by auto
  ultimately have ‹⊢ X # Neg p # G'› ‹⊢ X # Neg q # G'›
    using BeDis by blast+
  then have ‹⊢ Neg p # X # G'› ‹⊢ Neg q # X # G'›
    by (simp_all add: Swap)
  then have ‹⊢ Neg (Dis p q) # X # G'›
    using SC.BeDis by blast
  moreover have ‹Neg (Dis p q) ∈ set G'›
    using BeDis by auto
```

```
  ultimately show ?case
    using Extra by simp
next
  case (BeImp p z q)
  show ?case
  proof (cases ‹Neg X = Imp p q›)
    case true: True
    then have ‹⊢ X # z›
      using BeImp by blast
    then show ?thesis
    proof (cases ‹Neg (Neg X) ∈ set z›)
      case True
      then show ?thesis
      proof -
        have ‹set (p # z) - {Neg (Neg X)} ⊆ insert X (set G')›
          using BeImp.prems(2) true by fastforce
        then have ‹⊢ X # X # G'›
          using BeImp.hyps(2) True by simp
        then show ?thesis
          using SC.Extra by simp
      qed
    next
      case False
      then have ‹set (X # z) ⊆ set (X # G')›
        using BeImp true by auto
      then show ?thesis
        using ‹⊢ X # z› Order by blast
```

```
    qed
  next
    case False
    then have ‹Neg (Neg X) ∈ set (p # z)› ‹Neg (Neg X) ∈ set (Neg q # z)›
      using BeImp by auto
    moreover have
      ‹set (p # z) - {Neg (Neg X)} ⊆ set (p # G')›
      ‹set (Neg q # z) - {Neg (Neg X)} ⊆ set (Neg q # G')›
      using False BeImp by auto
    ultimately have ‹⊢ X # p # G'› ‹⊢ X # Neg q # G'›
      using BeImp by blast+
    then have ‹⊢ p # X # G'› ‹⊢ Neg q # X # G'›
      by (simp_all add: Swap)
    then have ‹⊢ Neg (Imp p q) # X # G'›
      using SC.BeImp by blast
    moreover have ‹Neg (Imp p q) ∈ set G'›
      using False BeImp by auto
    ultimately show ?thesis
      using Extra by simp
  qed
next
  case (GaExi t p z)
  then have *: ‹Neg (Neg X) ∈ set (sub 0 t p # z)›
    by auto
  then have ‹set (sub 0 t p # z) - {Neg (Neg X)} ⊆ set (sub 0 t p # G')›
    using GaExi by auto
  then have ‹⊢ X # sub 0 t p # G'›
```

```
    using * GaExi by blast
  then have ‹⊢ sub 0 t p # X # G'›
    using Swap by simp
  then have ‹⊢ Exi p # X # G'›
    using SC.GaExi by blast
  moreover have ‹Exi p ∈ set G'›
    using GaExi by auto
  ultimately show ?case
    using Extra by simp
next
  case (GaUni t p z)
  then have *: ‹Neg (Neg X) ∈ set (Neg (sub 0 t p) # z)›
    by auto
  then have ‹set (Neg (sub 0 t p) # z) - {Neg (Neg X)} ⊆ set (Neg (sub 0 t p) # G')›
    using GaUni by auto
  then have ‹⊢ X # Neg (sub 0 t p) # G'›
    using * GaUni by blast
  then have ‹⊢ Neg (sub 0 t p) # X # G'›
    using Swap by simp
  then have ‹⊢ Neg (Uni p) # X # G'›
    using SC.GaUni by blast
  moreover have ‹Neg (Uni p) ∈ set G'›
    using GaUni by auto
  ultimately show ?case
    using Extra by simp
next
  case (DeUni n p z)
```

```
then have *: ‹Neg (Neg X) ∈ set (sub 0 (Fun n []) p # z)›
  by auto

have ‹Neg (Neg X) ∈ set z›
  using DeUni by simp
then have ‹new n (Neg (Neg X))›
  using ‹news n (p # z)› by (induct z) auto
then have ‹news n (p # X # z)›
  using ‹news n (p # z)› by simp

let ?params = ‹params p ∪ params X ∪ (⋃p ∈ set z. params p) ∪ (⋃p ∈ set G'. params p)›

have ‹finite ?params›
  by simp
then obtain m where *: ‹m ∉ ?params ∪ {n}›
  using ex_new_if_finite by (metis finite.emptyI finite.insertI finite_UnI infinite_UNIV_listI)

let ?f = ‹id(n := m, m := n)›
let ?A = ‹sub 0 (Fun m []) p›
let ?X = ‹psubst ?f X›
let ?G' = ‹map (psubst ?f) G'›

have p: ‹psubst ?f (sub 0 (Fun n []) p) = ?A›
  using ‹news n (p # z)› * by simp
have X: ‹psubst ?f X = X›
  using ‹new n (Neg (Neg X))› * by simp
have G': ‹map (psubst ?f) ?G' = G'›
```

```
  using psubst_swap_twice by (induct G') simp_all

have ‹set z - {Neg (Neg X)} ⊆ set G'›
  using DeUni by auto
then have ‹set (map (psubst ?f) z) - {psubst ?f (Neg (Neg X))} ⊆ set ?G'›
  by auto
moreover have ‹map (psubst ?f) z = z›
  using ‹news n (p # z)› * by (induct z) simp_all
ultimately have ‹set z - {Neg (Neg X)} ⊆ set ?G'›
  using X by simp

then have ‹set (sub 0 (Fun n []) p # z) - {Neg (Neg X)} ⊆ set (sub 0 (Fun n []) p # ?G')›
  using DeUni by auto
then have ‹⊢ X # sub 0 (Fun n []) p # ?G'›
  using * DeUni by simp
then have ‹⊢ sub 0 (Fun n []) p # X # ?G'›
  using Swap by simp
then have ‹⊢ map (psubst ?f) (sub 0 (Fun n []) p # X # ?G')›
  using SC_psubst by blast
then have ‹⊢ sub 0 (Fun m []) p # X # G'›
  using p X G' by simp
moreover have ‹news m (p # X # G')›
  using * by (induct G') simp_all
ultimately have ‹⊢ Uni p # X # G'›
  using SC.DeUni by blast
moreover have ‹Uni p ∈ set G'›
  using DeUni by auto
```

```
  ultimately show ?case
    using Extra by simp
next
  case (DeExi n p z)
  then have *: ‹Neg (Neg X) ∈ set (Neg (sub 0 (Fun n []) p) # z)›
    by auto

  have ‹Neg (Neg X) ∈ set z›
    using DeExi by simp
  then have ‹new n (Neg (Neg X))›
    using ‹news n (p # z)› by (induct z) auto
  then have ‹news n (p # X # z)›
    using ‹news n (p # z)› by simp

  let ?params = ‹params p ∪ params X ∪ (⋃p ∈ set z. params p) ∪ (⋃p ∈ set G'. params p)›

  have ‹finite ?params›
    by simp
  then obtain m where *: ‹m ∉ ?params ∪ {n}›
    using ex_new_if_finite by (metis finite.emptyI finite.insertI finite_UnI infinite_UNIV_listI)

  let ?f = ‹id(n := m, m := n)›
  let ?A = ‹sub 0 (Fun m []) p›
  let ?X = ‹psubst ?f X›
  let ?G' = ‹map (psubst ?f) G'›

  have p: ‹psubst ?f (sub 0 (Fun n []) p) = ?A›
```

```
  using ‹news n (p # z)› * by simp
have X: ‹psubst ?f X = X›
  using ‹new n (Neg (Neg X))› * by simp
have G': ‹map (psubst ?f) ?G' = G'›
  using psubst_swap_twice by (induct G') simp_all

have ‹set z - {Neg (Neg X)} ⊆ set G'›
  using DeExi by auto
then have ‹set (map (psubst ?f) z) - {psubst ?f (Neg (Neg X))} ⊆ set ?G'›
  by auto
moreover have ‹map (psubst ?f) z = z›
  using ‹news n (p # z)› * by (induct z) simp_all
ultimately have ‹set z - {Neg (Neg X)} ⊆ set ?G'›
  using X by simp

then have ‹set (Neg (sub 0 (Fun n []) p) # z) - {Neg (Neg X)} ⊆
    set (Neg (sub 0 (Fun n []) p) # ?G')›
  using DeExi by auto
then have ‹⊢ X # Neg (sub 0 (Fun n []) p) # ?G'›
  using * DeExi by simp
then have ‹⊢ Neg (sub 0 (Fun n []) p) # X # ?G'›
  using Swap by simp
then have ‹⊢ map (psubst ?f) (Neg (sub 0 (Fun n []) p) # X # ?G')›
  using SC_psubst by blast
then have ‹⊢ Neg (sub 0 (Fun m []) p) # X # G'›
  using p X G' by simp
moreover have ‹news m (p # X # G')›
```

```
    using * by (induct G') simp_all
  ultimately have ‹⊢ Neg (Exi p) # X # G'›
    using SC.DeExi by blast
  moreover have ‹Neg (Exi p) ∈ set G'›
    using DeExi by auto
  ultimately show ?case
    using Extra by simp
next
  case (Extra p z)
  then show ?case
    by (simp add: insert_absorb)
qed

lemma SC_Neg_Neg: ‹⊢ Neg (Neg p) # z ⟹ ⊢ p # z›
  using special by simp

theorem TC_SC: ‹⊣ z ⟹ ⊢ map compl z›
proof (induct rule: TC.induct)
  case (Extra p z)
  then show ?case
    by (metis SC.Extra image_eqI list.set_map list.simps(9) member_set)
next
  case (Basic i l z)
  then show ?case
  proof -
    have ‹⊢ compl (Pre i l) # Pre i l # map compl z›
      by (metis member_set SC.Basic Extra compl.simps(3) list.set_intros)
```

```
    then show ?thesis
      by simp
  qed
next
  case (AlCon p q z)
  then have ‹⊢ compl p # compl q # map compl z›
    by simp
  then have ‹⊢ Neg p # Neg q # map compl z›
    using compl Swap Dummy BeImp by metis
  then show ?case
    using SC.AlCon by simp
next
  case (AlImp p q z)
  then have ‹⊢ compl p # q # map compl z›
    by simp
  then have ‹⊢ Neg p # q # map compl z›
    using compl Dummy BeImp by metis
  then show ?case
    using SC.AlImp by simp
next
  case (BeDis p z q)
  then have ‹⊢ compl p # map compl z› ‹⊢ compl q # map compl z›
    by simp_all
  then have ‹⊢ Neg p # map compl z› ‹⊢ Neg q # map compl z›
    using compl Dummy BeImp by metis+
  then show ?case
    using SC.BeDis by simp
```

```
next
  case (BeImp p z q)
  then have ‹⊢ p # map compl z› ‹⊢ compl q # map compl z›
    by simp_all
  then have ‹⊢ p # map compl z› ‹⊢ Neg q # map compl z›
    using compl Dummy SC.BeImp by metis+
  then have ‹⊢ Neg (Imp p q) # map compl z›
    using SC.BeImp by simp
  then have ‹⊢ compl (Imp p q) # map compl z›
    using ‹⊢ p # map compl z› compl by (metis fm.inject(2))
  then show ?case
    by simp
next
  case (GaUni t p z)
  then have ‹⊢ compl (sub 0 t p) # map compl z›
    by simp
  then have ‹⊢ Neg (sub 0 t p) # map compl z›
    using compl Dummy BeImp by metis
  then show ?case
    using SC.GaUni by simp
next
  case (DeExi n p z)
  then have ‹⊢ compl (sub 0 (Fun n []) p) # map compl z›
    by simp
  then have ‹⊢ Neg (sub 0 (Fun n []) p) # map compl z›
    using compl Dummy BeImp by metis
  moreover have ‹news n (p # map compl z)›
```

```
    using DeExi news_compl by simp
  ultimately show ?case
    using SC.DeExi by simp
next
  case (DeUni n p z)
  then have ‹⊢ sub 0 (Fun n []) p # map compl z›
    by simp
  moreover have ‹news n (p # map compl z)›
    using DeUni news_compl by simp
  ultimately show ?case
    using SC.DeUni by simp
qed (auto intro: SC.intros)

subsection ‹Completeness›

theorem SC_completeness:
  assumes ‹∀(e :: _ ⇒ htm) f g. list_all (semantics e f g) z ⟶ semantics e f g p›
  shows ‹⊢ p # map compl z›
proof -
  have ‹⊣ compl p # z›
    using assms tableau_completeness compl_compl unfolding tableauproof_def by simp
  then show ?thesis
    using TC_SC compl AlNegNeg compl.simps(1) list.simps(9) by (metis (full_types))
qed

corollary
  assumes ‹∀(e :: _ ⇒ htm) f g. semantics e f g p›
```

```
  shows ‹⊢ [p]›
  using assms SC_completeness list.map(1) by metis

abbreviation herbrand_valid (‹⫺ _› 0) where
  ‹(⫺ p) ≡ ∀(e :: _ ⇒ htm) f g. semantics e f g p›

theorem herbrand_completeness_soundness: ‹⫺ p ⟹ ⊢ [p]› ‹⊢ [p] ⟹ semantics e f g p›
  by (use SC_completeness list.map(1) in metis) (use SC_soundness in fastforce)

corollary ‹(⫺ p) = (⊢ [p])›
  using herbrand_completeness_soundness by fast

lemma map_compl_Neg: ‹map compl (map Neg z) = z›
  by (induct z) simp_all

theorem SC_complete:
  assumes ‹∀(e :: _ ⇒ htm) f g. ∃p ∈ set z. semantics e f g p›
  shows ‹⊢ z›
proof -
  have ‹∀(e :: _ ⇒ htm) f g. ¬ (∀p ∈ set (map Neg z). semantics e f g p)›
    using assms by fastforce
  then obtain p where
    ‹p ∈ set z›
    ‹∀(e :: _ ⇒ htm) f g. (∀p ∈ set (map Neg z). semantics e f g p) ⟶ semantics e f g p›
    using assms by blast
  then have ‹⊢ p # map compl (map Neg z)›
    using SC_completeness Ball_set by metis
```

```
  then have ‹⊢ p # z›
    using map_compl_Neg by simp
  with ‹p ∈ set z› show ?thesis
    using SC.Extra member_set by simp
qed

theorem SC_TC: ‹⊢ z ⟹ ⊣ map compl z›
proof (induct rule: SC.induct)
  case (Basic i l z)
  then show ?case
  proof -
    have ‹⊣ compl (Pre i l) # Pre i l # map compl z›
      using tableau_completeness tableauproof_def by fastforce
    then show ?thesis
      by simp
  qed
next
  case (Dummy z)
  then show ?case
    by (simp add: TC.Dummy)
next
  case (AlCon p q z)
  then show ?case
    using AlCon
    by (simp add: TC.AlCon)
next
  case (AlDis p q z)
```

```
  then have ‹⊣ compl p # compl q # map compl z›
    by simp
  then have ‹⊣ Neg p # Neg q # map compl z›
    using compl Swap' AlNegNeg' by metis
  then show ?case
    using TC.AlDis by simp
next
  case (AlImp p q z)
  then have ‹⊣ p # compl q # map compl z›
    by simp
  then have ‹⊣ p # Neg q # map compl z›
    using compl Swap' AlNegNeg' by metis
  then show ?case
    by (metis TC.AlImp compl list.simps(9) TC_Neg_Neg)
next
  case (BeCon p z q)
  then have ‹⊣ compl p # map compl z› ‹⊣ compl q # map compl z›
    by simp_all
  then have ‹⊣ Neg p # map compl z› ‹⊣ Neg q # map compl z›
    using compl AlNegNeg' by metis+
  then show ?case
    using TC.BeCon by simp
next
  case (BeDis p z q)
  then show ?case
    by (simp add: TC.BeDis)
next
```

```
  case (BeImp p z q)
  then show ?case
    using TC.BeImp compl compl.simps(1) list.simps(9) AlNegNeg' by metis
next
  case (GaExi t p z)
  then show ?case
    using TC.GaExi compl compl.simps(12) list.simps(9) AlNegNeg' by (metis (no_types))
next
  case (GaUni t p z)
  then show ?case
    by (simp add: TC.GaUni)
next
  case (DeUni n p z)
  then have ‹⊣ compl (sub 0 (Fun n []) p) # map compl z›
    by simp
  then have ‹⊣ Neg (sub 0 (Fun n []) p) # map compl z›
    using compl AlNegNeg' by metis
  moreover have ‹news n (p # map compl z)›
    using DeUni news_compl by simp
  ultimately show ?case
    using TC.DeUni by simp
next
  case (DeExi n p z)
  then show ?case
    using TC.DeExi news_compl by auto
next
  case (Extra p z)
```

```
  then show ?case
    by (metis TC.Extra image_eqI list.set_map list.simps(9) member_set)
qed

lemma TC_neg_compl: ‹(⊣ [Neg p]) ⟷ (⊣ [compl p])›
  by (metis compl compl.simps(1) TC_Neg_Neg TC_compl_compl)

lemma supra:
  assumes ‹∀(e :: _ ⇒ 'a) f g. list_all (semantics e f g) z ⟶ semantics e f g p›
    and ‹denumerable (UNIV :: 'a set)›
  shows ‹⊢ p # map compl z›
  using SC_completeness soundness' completeness' assms by blast

lemma super:
  assumes ‹⊢ p # map compl z›
  shows ‹∀e f g. list_all (semantics e f g) z ⟶ semantics e f g p›
proof -
  have ‹∀e f g. ¬ (∀p ∈ set z. semantics e f g p) ∨ semantics e f g p›
    using assms SC_soundness semantics_compl by fastforce
  then show ?thesis
    using Ball_set by metis
qed

lemma SC_compl_Neg: ‹(⊢ compl p # z) ⟷ (⊢ Neg p # z)›
  by (metis AlNegNeg compl SC_Neg_Neg)

lemma TC_compl_Neg: ‹(⊣ compl p # z) ⟷ (⊣ Neg p # z)›
```

```
  by (metis AlNegNeg' compl TC_Neg_Neg)

lemma TC_map_compl:
  assumes ‹⊣ map compl z›
  shows ‹⊢ z›
proof -
  have ‹∀(e :: _ ⇒ htm) f g. ∃p ∈ set (map compl z). ¬ semantics e f g p›
    using assms TC_soundness by blast
  then have ‹∀(e :: _ ⇒ htm) f g. ∃p ∈ set z. ¬ semantics e f g (compl p)›
    by fastforce
  then show ?thesis
    using SC_complete semantics_compl by metis
qed

lemma SC_map_compl:
  assumes ‹⊢ map compl z›
  shows ‹⊣ z›
proof -
  have ‹∀(e :: _ ⇒ htm) f g. ∃p ∈ set (map compl z). semantics e f g p›
    using assms SC_soundness by blast
  then have ‹∀(e :: _ ⇒ htm) f g. ∃p ∈ set z. semantics e f g (compl p)›
    by fastforce
  then show ?thesis
    using TC_complete semantics_compl by metis
qed

section ‹The Sequent Calculus is Sound and Complete›
```

```
theorem sound_complete: ‹valid p ⟷ (⊢ [p])›
proof
  assume ‹valid p›
  then show ‹⊢ [p]›
    using herbrand_completeness_soundness(1) valid_semantics by fast
next
  assume ‹⊢ [p]›
  then show ‹valid p›
    using herbrand_completeness_soundness(2) by fast
qed

lemma 1: ‹OK p z ⟹ ⊢ p # map compl z›
  by (simp add: SC_completeness soundness')

lemma 2: ‹⊢ p # map compl z ⟹ OK p z›
  using completeness'' by (simp add: super)

lemma 3:
  assumes ‹∀(e :: _ ⇒ 'a) f g. semantics e f g p›
    and ‹denumerable (UNIV :: 'a set)›
  shows ‹⊢ [p]›
  using assms completeness 1 by fastforce

lemma helper: ‹⊢ [p] ⟹ ⊣ [Neg p]›
  using TC_compl_Neg complete herbrand_completeness_soundness(2) by blast
```

```
lemma 4:
  assumes ‹∀(e :: _ ⇒ 'a) f g. semantics e f g p›
    and ‹denumerable (UNIV :: 'a set)›
  shows ‹⊣ [Neg p]›
  using assms 3 helper by fastforce

theorem OK_TC: ‹OK p z ⟷ (⊣ compl p # z)›
  using 1 2 SC_map_compl TC_compl_Neg TC_SC compl.simps list.simps(9) by metis

theorem OK_SC: ‹OK p z ⟷ (⊢ p # map compl z)›
  using 1 2 by fast

theorem TC: ‹(⊣ z) ⟷ (⊢ map compl z)›
  using SC_map_compl TC_SC by fast

theorem SC: ‹(⊢ z) ⟷ (⊣ map compl z)›
  using TC_map_compl SC_TC by fast

corollary ‹OK p z ⟷ (⊣ Neg p # z)›
  using TC OK_SC map_compl_Neg by simp

corollary ‹OK p z ⟷ (⊢ p # map Neg z)›
  using SC OK_TC map_compl_Neg by simp

corollary ‹(⊣ z) ⟷ (⊢ map Neg z)›
  using SC map_compl_Neg by simp
```

```
corollary ‹(⊢ z) ⟷ (⊣ map Neg z)›
  using TC map_compl_Neg by simp

section ‹Acknowledgements›

text ‹

Based on:

  • Stefan Berghofer:
    First-Order Logic According to Fitting
    □‹https://www.isa-afp.org/entries/FOL-Fitting.shtml›

  • Anders Schlichtkrull:
    The Resolution Calculus for First-Order Logic
    □‹https://www.isa-afp.org/entries/Resolution_FOL.shtml›

  • Jørgen Villadsen, Andreas Halkjær From, Alexander Birch Jensen & Anders Schlichtkrull:
    NaDeA - Natural Deduction Assistant.
    □‹https://github.com/logic-tools/nadea›
›

end
```