Basic First-Order Model Theory – A Translation from HOL Light

Sophie Tourret and Lawrence Paulson

February 27, 2025

Abstract

This AFP entry presents a proof of compactness of first-order logic, the Löwenheim-Skolem theorem and the Uniformity lemma. It is a translation of a HOL Light formalization work by John Harrison [1]. Whenever possible, existing Isabelle/HOL theories have been used instead of direct translation.

Contents

theory *FOL-Syntax*
  imports
    Main
    Propositional-Proof-Systems.Compactness
    First-Order-Terms.Term
    First-Order-Terms.Subterm-and-Context
begin

no-notation *Not* (¬)
no-notation *And* (**infix** ∧ *68*)
no-notation *Or* (**infix** ∨ *68*)

lemma *count-terms*:
  OFCLASS((*'f*::*countable*, *'v*::*countable*) *term*,*countable-class*)
  by *countable-datatype*

instance *term* :: (*countable*, *countable*) *countable*

  using *count-terms* **by** *simp*

type-synonym *nterm* = ‹(*nat*, *nat*) *term*›

lemma *count-nterms*: *OFCLASS*(*nterm*,*countable-class*)
using *count-terms* **by** *simp*

instance *formula* :: (*countable*) *countable*
by *countable-datatype*

term *test* (*0*, [*Var 0*])

abbreviation *functions-term* :: ‹*nterm* ⇒ (*nat* × *nat*) *set*› **where**
‹*functions-term t* ≡ *funas-term t*›

datatype *form* =
Bot (‹⊥›)
| *Atom* (*pred*:*nat*) (*args*:‹*nterm list*›)
| *Implies form form* (**infixl** ‹⟶› *85*)
| *Forall nat form* (‹∀ -. -› [*0*, *70*] *70*)

fun *functions-form* :: ‹*form* ⇒ (*nat* × *nat*) *set*› **where**
‹*functions-form* ⊥ = {}›
| ‹*functions-form* (*Atom p ts*) = (⋃ *t* ∈ *set ts*. *functions-term t*)›
| ‹*functions-form* (*φ* ⟶ *ψ*) = *functions-form φ* ∪ *functions-form ψ*›
| ‹*functions-form* (∀ *x*. *φ*) = *functions-form φ*›

fun *predicates-form* :: ‹*form* ⇒ (*nat* × *nat*) *set*› **where**
‹*predicates-form* ⊥ = {}›
| ‹*predicates-form* (*Atom p ts*) = {(*p*, *length ts*)}›
| ‹*predicates-form* (*φ* ⟶ *ψ*) = *predicates-form φ* ∪ *predicates-form ψ*›
| ‹*predicates-form* (∀ *x*. *φ*) = *predicates-form φ*›

definition *functions-forms* :: ‹*form set* ⇒ (*nat* × *nat*) *set*› **where**
‹*functions-forms fms* ≡ ⋃*f* ∈ *fms*. *functions-form f*›

definition *predicates* :: ‹*form set* ⇒ (*nat* × *nat*) *set*› **where**
‹*predicates fms* ≡ ⋃*f* ∈ *fms*. *predicates-form f*›

definition *language* :: ‹*form set* ⇒ ((*nat* × *nat*) *set* × (*nat* × *nat*) *set*)› **where**
‹*language fms* = (*functions-forms fms*, *predicates fms*)›

lemma *lang-singleton*: ‹*language* {*p*} = (*functions-form p*, *predicates-form p*)›
unfolding *language-def functions-forms-def predicates-def* **by** *simp*

abbreviation *Not* :: ‹*form* ⇒ *form*› (‹¬ -› [*90*] *90*) **where**
‹¬ *φ* ≡ *φ* ⟶ ⊥›

abbreviation *Top* :: ‹*form*› (‹⊤›) **where**
‹⊤ ≡ ¬ ⊥›

abbreviation *Or* :: ‹*form* ⇒ *form* ⇒ *form*›
(**infixl** ‹∨› *84*) **where**
‹$\varphi \vee \psi \equiv (\varphi \longrightarrow \psi) \longrightarrow \psi$›

abbreviation *And* :: ‹*form* ⇒ *form* ⇒ *form*›
(**infixl** ‹∧› *84*) **where**
‹$\varphi \wedge \psi \equiv \neg\ (\neg\ \varphi \vee \neg\ \psi)$›

abbreviation *Equiv* :: ‹*form* ⇒ *form* ⇒ *form*›
(**infix** ‹⟷› *70*) **where**
‹$\varphi \longleftrightarrow \psi \equiv (\varphi \longrightarrow \psi \wedge \psi \longrightarrow \varphi)$›

abbreviation *Exists* :: ‹*nat* ⇒ *form* ⇒ *form*›
(‹∃ -. -› [*0*, *70*] *70*) **where**
‹$\exists\ x.\ \varphi \equiv \neg\ (\forall\ x.\ \neg\ \varphi)$›

lemma *ex-all-distinct*: ‹$\forall\, x.\ \varphi \neq \exists\, y.\ \psi$›
by *simp*

abbreviation *FVT* :: ‹*nterm* ⇒ *nat set*› **where**
‹*FVT* ≡ *vars-term*›

fun *FV* :: ‹*form* ⇒ *nat set*› **where**
‹*FV* ⊥ = {}›
| ‹*FV* (*Atom* - *ts*) = (⋃ *a* ∈ *set ts*. *FVT a*)›
| ‹*FV* (φ ⟶ ψ) = *FV* φ ∪ *FV* ψ›
| ‹*FV* (∀ *x*. φ) = *FV* φ − {*x*}›

lemma *FV-all-subs*: ‹*FV* φ ⊆ *FV* (∀ *x*. φ) ∪ {*x*}›
by *fastforce*

lemma *FV-exists*: ‹*FV* (∃ *x*. φ) = *FV* φ − {*x*}›
by *simp*

lemma *finite-FV*: ‹*finite* (*FV* φ)›
by (*induction* φ, *auto*)

fun *BV* :: ‹*form* ⇒ *nat set*› **where**
‹*BV* ⊥ = {}›
| ‹*BV* (*Atom* - *args'*) = {}›
| ‹*BV* (φ ⟶ ψ) = *BV* φ ∪ *BV* ψ›
| ‹*BV* (∀ *x*. φ) = *BV* φ ∪ {*x*}›

lemma *finite-BV*: ‹*finite* (*BV* φ)›
by (*induction* φ, *auto*)

definition *variant* :: ‹*nat set* ⇒ *nat*› **where**
‹*variant s* = *Max s* + *1*›

lemma *variant-finite*: ‹*finite s* ⟹ ¬ (*variant s* ∈ *s*)›
unfolding *variant-def* **using** *Max-ge less-le-not-le* **by** *auto*

lemma *variant-form*: ‹¬ *variant* (*FV* φ) ∈ *FV* φ›
using *variant-finite finite-FV* **by** *blast*

fun *formsubst* :: ‹*form* ⇒ (*nat*, *nat*) *subst* ⇒ *form*› (**infixl** ‹·$_{fm}$› *75*) **where**
‹⊥ ·$_{fm}$ - = ⊥›
| ‹(*Atom p ts*) ·$_{fm}$ σ = *Atom p* [*t* · σ. *t* ← *ts*]›
| ‹(φ ⟶ ψ) ·$_{fm}$ σ = (φ ·$_{fm}$ σ) ⟶ (ψ ·$_{fm}$ σ)›
| ‹(∀ *x*. φ) ·$_{fm}$ σ =
(*let* σ′ = σ(*x* := *Var x*);
z = *if* ∃ *y*. *y* ∈ *FV* (∀ *x*. φ) ∧ *x* ∈ *FVT* (σ′ *y*)
then variant (*FV* (φ ·$_{fm}$ σ′)) *else x in*
∀ *z*. (φ ·$_{fm}$ σ(*x* := *Var z*)))›

fun *formsubst2* :: ‹*form* ⇒ (*nat*, *nat*) *subst* ⇒ *form*› (**infixl** ‹·$_{fm}$*2*› *75*) **where**
‹⊥ ·$_{fm}$*2* - = ⊥›
| ‹(*Atom p ts*) ·$_{fm}$*2* σ = *Atom p* [*t* · σ. *t* ← *ts*]›
| ‹(φ ⟶ ψ) ·$_{fm}$*2* σ = (φ ·$_{fm}$*2* σ) ⟶ (ψ ·$_{fm}$*2* σ)›
| ‹(∀ *x*. φ) ·$_{fm}$*2* σ = (*let* σ′ = σ(*x* := *Var x*) *in*
(*if* ∃ *y*. *y* ∈ *FV* (∀ *x*. φ) ∧ *x* ∈ *FVT* (σ′ *y*)
then (*let z* = *variant* (*FV* (φ ·$_{fm}$*2* σ′)) *in*
∀ *z*. (φ ·$_{fm}$*2* σ(*x* := *Var z*)))
else ∀ *x*. (φ ·$_{fm}$*2* σ′)))›

lemma *formsubst-def-switch*: ‹φ ·$_{fm}$ σ = φ ·$_{fm}$*2* σ›
proof (*induction* φ *arbitrary*: σ *rule*: *form.induct*)
case *Bot*
then show *?case*
by *fastforce*
next
case (*Atom x1 x2*)
then show *?case*
by *fastforce*
next
case (*Implies x1 x2*)
then show *?case*
by *fastforce*
next
case (*Forall x1 x2*)
then show *?case*
by (*smt* (*verit*, *best*) *formsubst.simps*(*4*) *formsubst2.simps*(*4*))
qed

lemma *termsubst-valuation*: ‹∀ *x*∈*FVT t*. σ *x* = σ′ *x* ⟹ *t* · σ = *t* · σ′›

using *eval-same-vars* **by** *blast*

lemma *termsetsubst-valuation*: ‹$\forall\, y \in T.\ \forall\, x{\in}FVT\ y.\ \sigma\ x = \sigma'\ x \Longrightarrow t \in T \Longrightarrow t \cdot \sigma = t \cdot \sigma'$›
using *termsubst-valuation* **by** *fast*

lemma *formsubst-valuation*: ‹$\forall\, x{\in}(FV\ \varphi).\ (Var\ x) \cdot \sigma = (Var\ x) \cdot \sigma' \Longrightarrow \varphi \cdot_{fm} \sigma = \varphi \cdot_{fm} \sigma'$›
proof (*induction* φ *arbitrary*: $\sigma\ \sigma'$ *rule*:*form.induct*)
case *Bot*
then show *?case* **by** *simp*
next
case (*Atom x1 x2*)
then show *?case*
using *termsetsubst-valuation*
by *auto*
next
case (*Implies x1 x2*)
then show *?case* **by** *simp*
next
case (*Forall x* φ)
define σ'' **where** $\sigma'' = \sigma(x := Var\ x)$
define σ''' **where** $\sigma''' = \sigma'(x := Var\ x)$
have *ex-var-equiv*: ‹$\exists\ y.\ y \in FV\ (\forall\ x.\ \varphi) \wedge x \in FVT\ (\sigma''\ y) \equiv \exists\ y.\ y \in FV\ (\forall\ x.\ \varphi) \wedge x \in FVT\ (\sigma'''\ y)$›
using σ''*-def* σ'''*-def Forall*(*2*)
by (*smt* (*verit*, *ccfv-threshold*) *eval-term.simps*(*1*) *fun-upd-other fun-upd-same*)
have *sig-x-subst*: ‹$\forall\, y{\in}FV\ \varphi.\ Var\ y \cdot \sigma(x := Var\ z) = Var\ y \cdot \sigma'(x := Var\ z)$›
for *z*
using *Forall*(*2*) **by** *simp*
show *?case*
proof (*cases* ‹$\exists\ y.\ y \in FV\ (\forall\ x.\ \varphi) \wedge x \in FVT\ (\sigma''\ y)$›)
case *True*
then have ‹$(\forall\ x.\ \varphi) \cdot_{fm} \sigma = (let\ z = variant\ (FV\ (\varphi \cdot_{fm} \sigma''))\ in \forall\ z.\ (\varphi \cdot_{fm} \sigma(x := Var\ z)))$›
by (*simp add*: σ''*-def*)
also have ‹$... = (let\ z = variant\ (FV\ (\varphi \cdot_{fm} \sigma'''))\ in \forall\ z.\ (\varphi \cdot_{fm} \sigma'(x := Var\ z)))$›
using *sig-x-subst* σ'''*-def* **by** (*metis Forall.IH* σ''*-def*)
also have ‹$... = (\forall\ x.\ \varphi) \cdot_{fm} \sigma'$›
using *True* σ'''*-def ex-var-equiv formsubst.simps*(*4*) **by** *presburger*
finally show *?thesis* **.**
next
case *False*
then show *?thesis*
using *Forall.IH* σ'''*-def* σ''*-def ex-var-equiv sig-x-subst* **by** *auto*
qed
qed

lemma ‹$\{x.\ \exists y.\ y \in (s \cup t) \wedge P\ x\ y\} = \{x.\ \exists y.\ y \in s \wedge P\ x\ y\} \cup \{x.\ \exists y.\ y \in t \wedge P\ x\ y\}$›
by *blast*

lemma *formsubst-structure-bot*: ‹$\varphi \cdot_{fm} \sigma = \bot \longleftrightarrow \varphi = \bot$›
by (*smt* (*verit*) *form.distinct*(5) *form.simps*(5) *form.simps*(7) *formsubst.elims*)

lemma *formsubst-structure-pred*: ‹$(\exists p\ ts.\ \varphi \cdot_{fm} \sigma = Atom\ p\ ts) \longleftrightarrow (\exists p\ ts.\ \varphi = Atom\ p\ ts)$›
proof (*cases* φ)
case (*Forall x* ψ)
then show *?thesis*
using *formsubst-def-switch* **by** (*metis* (*no-types*, *lifting*) *form.distinct*(10) *formsubst.simps*(4))
qed *auto*

lemma *formsubst-structure-imp*: ‹$(\exists \varphi 1\ \varphi 2.\ \varphi \cdot_{fm} \sigma = \varphi 1 \longrightarrow \varphi 2) \longleftrightarrow (\exists \psi 1\ \psi 2.\ \varphi = \psi 1 \longrightarrow \psi 2)$›
proof (*cases* φ)
case (*Forall x* ψ)
then show *?thesis*
using *formsubst-def-switch*
by (*metis* (*no-types*, *lifting*) *form.distinct*(11) *formsubst.simps*(4))
qed *auto*

lemma *formsubst-structure-all*: ‹$(\exists x\ \psi.\ \varphi \cdot_{fm} \sigma = (\forall x.\ \psi)) \longleftrightarrow (\exists x\ \psi.\ \varphi = (\forall x.\ \psi))$›
proof (*cases* φ)
case (*Forall x* ψ)
then show *?thesis*
using *formsubst-def-switch*
by (*metis* (*no-types*, *lifting*) *formsubst.simps*(4))
qed *auto*

lemma *formsubst-structure-not*: ‹$(\exists \psi.\ \varphi \cdot_{fm} \sigma = Not\ \psi) \longleftrightarrow (\exists \psi.\ \varphi = Not\ \psi)$›
using *formsubst-structure-imp formsubst-structure-bot*
by (*metis form.sel*(4) *formsubst.simps*(3))

lemma *formsubst-structure-not-all-imp*:
‹$(\exists x\ \psi.\ \varphi \cdot_{fm} \sigma = (\forall x.\ \psi) \longrightarrow \bot) \longleftrightarrow (\exists x\ \psi.\ \varphi = (\forall x.\ \psi) \longrightarrow \bot)$›
proof (*cases* φ)
case *Bot*
then show *?thesis* **by** *simp*
next
case (*Atom p ts*)
then show *?thesis* **by** *simp*
next
case (*Implies* $\varphi 1\ \varphi 2$)

then show *?thesis*
by (*metis form.inject*(*2*) *formsubst.simps*(*3*) *formsubst-structure-all formsubst-structure-not*)
next
case (*Forall y ψ*)
then show *?thesis*
by (*metis form.distinct*(*11*) *formsubst-structure-not*)
qed

lemma *formsubst-structure-all-not*:
‹(∃ *x ψ*. *φ* ·$_{fm}$ *σ* = (∀ *x*. *ψ* ⟶ ⊥)) ⟷ (∃ *x ψ*. *φ* = (∀ *x*. *ψ* ⟶ ⊥))›
proof
show ‹∃ *x ψ*. *φ* = ∀ *x*. *ψ* ⟶ ⊥ ⟹ ∃ *x ψ*. *φ* ·$_{fm}$ *σ* = ∀ *x*. *ψ* ⟶ ⊥›
by (*smt* (*verit*, *ccfv-threshold*) *formsubst.simps*(*1*) *formsubst.simps*(*3*) *formsubst.simps*(*4*))
next
assume ‹∃ *x ψ*. *φ* ·$_{fm}$ *σ* = ∀ *x*. *ψ* ⟶ ⊥›
then obtain *z ψ′* **where** *phi-sub-is*: ‹*φ* ·$_{fm}$ *σ* = ∀ *z*. *ψ′* ⟶ ⊥›
by *blast*
then obtain *x φ′* **where** *phi-is*: ‹*φ* = ∀ *x*. *φ′*›
using *formsubst-structure-all* **by** *blast*
then have ‹∃ *σ′*. *φ′* ·$_{fm}$ *σ′* = *ψ′* ⟶ ⊥›
using *phi-sub-is*
by (*metis* (*no-types*, *lifting*) *form.sel*(*6*) *formsubst.simps*(*4*))
then obtain *σ′* **where** ‹*φ′* ·$_{fm}$ *σ′* = *ψ′* ⟶ ⊥›
by *blast*
then have ‹∃ *ψ*. *φ′* = *ψ* ⟶ ⊥›
using *formsubst-structure-imp formsubst-structure-not* **by** *blast*
then show ‹∃ *x ψ*. *φ* = ∀ *x*. *ψ* ⟶ ⊥›
using *phi-is* **by** *blast*
qed

lemma *formsubst-structure-ex*: ‹(∃ *x ψ*. *φ* ·$_{fm}$ *σ* = (∃ *x*. *ψ*)) ⟷ (∃ *x ψ*. *φ* = (∃ *x*. *ψ*))›
proof
assume ‹∃ *x ψ*. *φ* ·$_{fm}$ *σ* = (∃ *x*. *ψ*)›
then show ‹∃ *x ψ*. *φ* = (∃ *x*. *ψ*)›
by (*metis form.inject*(*2*) *formsubst.simps*(*3*) *formsubst-structure-all-not formsubst-structure-not*)
next
assume ‹(∃ *x ψ*. *φ* = (∃ *x*. *ψ*))›
then show ‹∃ *x ψ*. *φ* ·$_{fm}$ *σ* = (∃ *x*. *ψ*)›
by (*smt* (*verit*, *ccfv-threshold*) *formsubst.simps*(*1*) *formsubst.simps*(*3*) *formsubst.simps*(*4*))
qed

lemma *formsubst-structure*: ‹(*φ* ·$_{fm}$ *σ* = ⊥ ⟷ *φ* = ⊥) ∧
((∃ *p ts*. *φ* ·$_{fm}$ *σ* = *Atom p ts*) ⟷ (∃ *p ts*. *φ* = *Atom p ts*)) ∧
((∃ *φ1 φ2*. *φ* ·$_{fm}$ *σ* = *φ1* ⟶ *φ2*) ⟷ (∃ *ψ1 ψ2*. *φ* = *ψ1* ⟶ *ψ2*)) ∧
((∃ *x ψ*. *φ* ·$_{fm}$ *σ* = (∀ *x*. *ψ*)) ⟷ (∃ *x ψ*. *φ* = (∀ *x*. *ψ*)))›

using *formsubst-structure-bot formsubst-structure-pred formsubst-structure-imp formsubst-structure-all*
by *auto*

lemma *formsubst-fv*: ‹$FV\ (\varphi \cdot_{fm} \sigma) = \{x.\ \exists\, y.\ y \in (FV\ \varphi) \wedge x \in FVT\ ((Var\ y) \cdot \sigma)\}$›
proof (*induction* φ *arbitrary*: σ *rule*:*form.induct*)
case (*Atom x1 x2*)
have ‹$FV\ (Atom\ x1\ x2 \cdot_{fm} \sigma) = (\bigcup a \in set\ x2.\ FVT\ (a\ \cdot \sigma))$›
by *auto*
also have ‹$... = \{x.\ \exists\, y.\ y \in (\bigcup a \in set\ x2.\ FVT\ a) \wedge x \in FVT\ ((Var\ y) \cdot \sigma)\}$›
proof
show ‹$(\bigcup a{\in}set\ x2.\ FVT\ (a \cdot \sigma)) \subseteq \{x.\ \exists\, y.\ y \in (\bigcup a{\in}set\ x2.\ FVT\ a) \wedge x \in FVT\ (Var\ y \cdot \sigma)\}$›
proof
fix *v*
assume ‹$v \in (\bigcup a{\in}set\ x2.\ FVT\ (a \cdot \sigma))$›
then obtain *a* **where** *a-is*: ‹$a \in set\ x2$› ‹$v \in FVT\ (a \cdot \sigma)$›
by *auto*
then obtain *ya* **where** ‹$ya \in FVT\ a$› ‹$v \in FVT\ (Var\ ya \cdot \sigma)$›
using *eval-term.simps*(*1*) *vars-term-subst-apply-term* **by** *force*
then show ‹$v \in \{x.\ \exists\, y.\ y \in (\bigcup a{\in}set\ x2.\ FVT\ a) \wedge x \in FVT\ (Var\ y \cdot \sigma)\}$›
using *a-is* **by** *auto*
qed
next
show ‹$\{x.\ \exists\, y.\ y \in (\bigcup a{\in}set\ x2.\ FVT\ a) \wedge x \in FVT\ (Var\ y \cdot \sigma)\} \subseteq (\bigcup a{\in}set\ x2.\ FVT\ (a \cdot \sigma))$›
proof
fix *v*
assume ‹$v \in \{x.\ \exists\, y.\ y \in (\bigcup a{\in}set\ x2.\ FVT\ a) \wedge x \in FVT\ (Var\ y \cdot \sigma)\}$›
then obtain *yv* **where** ‹$yv \in (\bigcup a{\in}set\ x2.\ FVT\ a)$› ‹$v \in FVT\ (Var\ yv \cdot \sigma)$›
by *auto*
then show ‹$v \in (\bigcup a{\in}set\ x2.\ FVT\ (a \cdot \sigma))$›
using *vars-term-subst-apply-term* **by** *fastforce*
qed
qed
also have ‹$... = \{x.\ \exists\, y.\ y \in (FV\ (Atom\ x1\ x2)) \wedge x \in FVT\ ((Var\ y) \cdot \sigma)\}$›
by *auto*
finally show *?case* **.**
next
case (*Forall x* φ)
define σ' **where** $\sigma' = \sigma(x := Var\ x)$
show *?case*
proof (*cases* $\exists\ y.\ y \in FV\ (\forall\ x.\ \varphi) \wedge x \in FVT\ (\sigma'\ y)$)
case *True*
then obtain *y* **where** *y-in*: ‹$y \in FV\ (\forall\ x.\ \varphi)$› **and** *x-in*: ‹$x \in FVT\ (\sigma'\ y)$›
by *blast*
then have *y-neq-x*: ‹$y \neq x$› **by** *simp*
then have *y-in2*: ‹$y \in FV\ \varphi$›

using *y-in* **by** *fastforce*
have *x-in2*: ‹$x \in FVT\ (Var\ y \cdot \sigma)$›
using *x-in y-neq-x* **unfolding** σ'*-def* **by** *simp*

define z **where** $z = variant\ (FV\ (\varphi \cdot_{fm} \sigma'))$
have *x-in3*: ‹$x \in FVT\ (Var\ y \cdot \sigma(x := Var\ z))$›
using *x-in2 y-neq-x* **by** *simp*

have ‹$(\forall\ x.\ \varphi) \cdot_{fm} \sigma = \forall\ z.\ (\varphi \cdot_{fm} \sigma(x := Var\ z))$›
using *z-def formsubst-def-switch*
by (*smt* (*verit*, *ccfv-threshold*) *True* σ'*-def formsubst.simps*(4))
then have ‹$FV\ ((\forall\ x.\ \varphi) \cdot_{fm} \sigma) = \{xa.\ \exists y.\ y \in FV\ \varphi \wedge xa \in FVT\ (Var\ y \cdot \sigma(x := Var\ z))\} - \{z\}$›
using *Forall*[*of* $\sigma(x := Var\ z)$] **using** *FV.simps*(4) **by** *presburger*
also have ‹$... = \{xa.\ \exists y.\ y \in FV\ \varphi - \{x\} \wedge xa \in FVT\ (Var\ y \cdot \sigma)\}$›
proof
show ‹$\{xa.\ \exists y.\ y \in FV\ \varphi \wedge xa \in FVT\ (Var\ y \cdot \sigma(x := Var\ z))\} - \{z\} \subseteq \{xa.\ \exists y.\ y \in FV\ \varphi - \{x\} \wedge xa \in FVT\ (Var\ y \cdot \sigma)\}$›
proof
fix *xa*
assume *xa-in*: ‹$xa \in \{xa.\ \exists y.\ y \in FV\ \varphi \wedge xa \in FVT\ (Var\ y \cdot \sigma(x := Var\ z))\} - \{z\}$›
then obtain *ya* **where** *ya-in*: ‹$ya \in FV\ \varphi$› **and** *xa-image*: ‹$xa \in FVT\ (Var\ ya \cdot \sigma(x := Var\ z))$›
by *blast*
have *ya-neq-x*: ‹$ya \neq x$› **using** *xa-image xa-in* **by** *fastforce*
then have ‹$xa \in FVT\ (Var\ ya \cdot \sigma)$› **using** *xa-image* **by** *simp*
moreover have ‹$ya \in FV\ \varphi - \{x\}$›
using *ya-neq-x ya-in* **by** *blast*
ultimately show ‹$xa \in \{xa.\ \exists y.\ y \in FV\ \varphi - \{x\} \wedge xa \in FVT\ (Var\ y \cdot \sigma)\}$›
by *auto*
qed
next
show ‹$\{xa.\ \exists y.\ y \in FV\ \varphi - \{x\} \wedge xa \in FVT\ (Var\ y \cdot \sigma)\} \subseteq \{xa.\ \exists y.\ y \in FV\ \varphi \wedge xa \in FVT\ (Var\ y \cdot \sigma(x := Var\ z))\} - \{z\}$›
proof
fix *xa*
assume *xa-in*: ‹$xa \in \{xa.\ \exists y.\ y \in FV\ \varphi - \{x\} \wedge xa \in FVT\ (Var\ y \cdot \sigma)\}$›
then obtain *ya* **where** *ya-in*: ‹$ya \in FV\ \varphi - \{x\}$› **and** *xa-image*: ‹$xa \in FVT\ (Var\ ya \cdot \sigma)$›
by *blast*
have *ya-neq*: ‹$ya \neq x$› **using** *ya-in* **by** *blast*
then have *xa-in2*: ‹$xa \in FVT\ (Var\ ya \cdot \sigma(x := Var\ z))$› **using** *xa-image* **by** *simp*
then have ‹$xa \in FV\ (\varphi \cdot_{fm} \sigma(x := Var\ z))$›
using *ya-in Forall* **by** *force*
then have ‹$xa \in FV\ (\varphi \cdot_{fm} \sigma')$›
using *ya-neq Forall xa-image ya-in* **unfolding** σ'*-def* **by** *auto*

```
      then have ‹xa ≠ z› using z-def unfolding variant-def
        by (metis Max-ge Suc-eq-plus1 finite-FV lessI less-le-not-le)
      moreover have ‹ya ∈ FV φ› using ya-in by blast
      ultimately show ‹xa ∈ {xa. ∃ y. y ∈ FV φ ∧ xa ∈ FVT (Var y · σ(x :=
Var z))} − {z}›
        using xa-in2 by blast
    qed
  qed
  finally show ?thesis by simp
 next
  case False
  then have ‹(∀ x. φ) ·fm σ = ∀ x. (φ ·fm σ')›
    using formsubst-def-switch σ'-def by fastforce
   then have ‹FV ((∀ x. φ) ·fm σ) = {z. ∃ y. y ∈ FV φ ∧ z ∈ FVT (Var y ·
σ')} − {x}›
    using Forall by simp
  also have ‹... = {z. ∃ y. y ∈ FV (∀ x. φ) ∧ z ∈ FVT (Var y · σ)}›
    using False unfolding σ'-def by auto
  finally show ?thesis .
 qed
qed auto

lemma subst-var [simp]: ‹φ ·fm Var = φ›
 by (induction φ) auto

lemma formsubst-rename: ‹FV (φ ·fm (subst x (Var y))) − {y} = FV φ − {x}
− {y}›
proof (cases y = x)
 case True
 then have ‹subst x (Var y) = Var›
  by simp
 then have ‹FV (φ ·fm (subst x (Var y))) = FV φ›
  using subst-var by metis
 then show ?thesis
  using True by simp
next
 case False
 show ?thesis
 proof
  show ‹FV (φ ·fm subst x (Var y)) − {y} ⊆ FV φ − {x} − {y}›
  proof
   fix v
   assume v-in: ‹v ∈ FV (φ ·fm subst x (Var y)) − {y}›
   moreover have ‹v ≠ x›
    using v-in
   by (smt (verit, ccfv-threshold) DiffE Term.term.simps(17) eval-term.simps(1)
     formsubst-fv fun-upd-other insert-iff mem-Collect-eq subst-def subst-simps(1))
   moreover have ‹v ∈ FV φ›
     by (smt (verit, del-insts) DiffE Term.term.simps(17) eval-term.simps(1)
```

formsubst-fv
fun-upd-other mem-Collect-eq subst-def subst-simps(1) subst-var v-in)
ultimately show ‹$v \in FV\ \varphi - \{x\} - \{y\}$›
by *blast*
qed
next
show ‹$FV\ \varphi - \{x\} - \{y\} \subseteq FV\ (\varphi \cdot_{fm} subst\ x\ (Var\ y)) - \{y\}$›
using *formsubst-fv False* **by** (*smt* (*verit*, *del-insts*) *Diff-iff Term.term.simps*(17)

mem-Collect-eq singleton-iff subsetI subst-ident)
qed
qed

lemma *termsubst-functions-term*:
‹*functions-term* $(t \cdot \sigma) =$ *functions-term* $t \cup \{x.\ \exists y.\ y \in FVT\ t \wedge x \in$ *functions-term* $((Var\ y) \cdot \sigma)\}$›
by (*induction t arbitrary*: σ) *auto*

lemma *formsubst-functions-form*:
‹*functions-form* $(\varphi \cdot_{fm} \sigma) =$ *functions-form* $\varphi \cup \{x.\ \exists y.\ y \in FV\ \varphi \wedge x \in$ *functions-term* $((Var\ y) \cdot \sigma)\}$›
proof (*induction* φ *arbitrary*: σ)
case *Bot*
then show *?case* **by** *simp*
next
case (*Atom p ts*)
show *?case*
using *termsubst-functions-term* **by** *auto*
next
case (*Implies* $\varphi\ \psi$)
then show *?case* **by** *auto*
next
case (*Forall x* φ)
define σ' **where** ‹$\sigma' = \sigma(x := Var\ x)$›
define z **where** ‹$z = variant\ (FV\ (\varphi \cdot_{fm} 2\ \sigma'))$›
have *fun-terms-set-eq*: ‹$\{xa.\ \exists y.\ y \in FV\ (\forall\ x.\ \varphi) \wedge xa \in$ *functions-term* $(Var\ y \cdot \sigma)\} =$
(*if* $(\exists y.\ y \in FV\ (\forall x.\ \varphi) \wedge x \in FVT\ (\sigma'\ y))$
then $\{xa.\ \exists y.\ y \in FV\ \varphi \wedge xa \in$ *functions-term* $((Var\ y) \cdot \sigma(x := Var\ z))\}$
else $\{x.\ \exists y.\ y \in FV\ \varphi \wedge x \in$ *functions-term* $((Var\ y) \cdot \sigma')\}$›) (**is** *?lhs = ?rhs*)
proof
show ‹*?lhs* $\subseteq$ *?rhs*›
proof
fix v
assume *v-in*: $v \in$ *?lhs*
have ‹$\exists y.\ y \in FV\ (\forall x.\ \varphi) \wedge x \in FVT\ (\sigma'\ y) \Longrightarrow v \in \{xa.\ \exists y.\ y \in FV\ \varphi \wedge xa \in$ *functions-term* $((Var\ y) \cdot \sigma(x := Var\ z))\}$›
using *v-in* **by** *auto*
moreover have ‹$\nexists y.\ y \in FV\ (\forall x.\ \varphi) \wedge x \in FVT\ (\sigma'\ y) \Longrightarrow v \in \{x.\ \exists y.\ y$

$\in FV\ \varphi \wedge x \in functions\text{-}term\ ((Var\ y) \cdot \sigma')\}\rangle$
using *v-in* σ'*-def* **by** *auto*
ultimately show $v \in$ *?rhs*
by *argo*
qed
next
show ‹*?rhs* $\subseteq$ *?lhs*›
proof
fix v
assume *v-in*: ‹$v \in$ *?rhs*›
have ‹$\exists\, y.\ y \in FV\ (\forall\, x.\ \varphi) \wedge x \in FVT\ (\sigma'\ y) \Longrightarrow v \in$ *?lhs*›
using *v-in* **by** (*smt* (*verit*, *del-insts*) *Diff-empty Diff-insert0 FV.simps*(4) *Term.term.simps*(17) *empty-iff eval-term.simps*(1) *eval-with-fresh-var fun-upd-same funas-term.simps*(1) *insertE insert-Diff mem-Collect-eq*)
moreover have ‹$\nexists\, y.\ y \in FV\ (\forall\, x.\ \varphi) \wedge x \in FVT\ (\sigma'\ y) \Longrightarrow v \in$ *?lhs*›
using *v-in* **by** (*smt* (*verit*, *ccfv-threshold*) *Diff-iff FV.simps*(4) σ'*-def empty-iff eval-term.simps*(1) *fun-upd-other fun-upd-same funas-term.simps*(1) *insertE mem-Collect-eq*)
ultimately show ‹$v \in$ *?lhs*›
by *argo*
qed
qed
have ‹*functions-form* $((\forall\, x.\ \varphi) \cdot_{fm} \sigma) =$ *functions-form* $((\forall\ x.\ \varphi) \cdot_{fm} 2\ \sigma)$›
using *formsubst-def-switch* **by** *simp*
also have ‹... = (*if* $(\exists\, y.\ y \in FV\ (\forall\, x.\ \varphi) \wedge x \in FVT\ (\sigma'\ y))$
then functions-form $(\forall\ z.\ (\varphi \cdot_{fm} 2\ \sigma(x := Var\ z)))$
else functions-form $(\forall\ x.\ (\varphi \cdot_{fm} 2\ \sigma'))$)›
using σ'*-def z-def* **by** (*smt* (*verit*) *formsubst2.simps*(4))
also have ‹... = (*if* $(\exists\, y.\ y \in FV\ (\forall\, x.\ \varphi) \wedge x \in FVT\ (\sigma'\ y))$
then functions-form $\varphi \cup \{xa.\ \exists\, y.\ y \in FV\ \varphi \wedge xa \in functions\text{-}term\ ((Var\ y) \cdot \sigma(x := Var\ z))\}$
else functions-form $\varphi \cup \{x.\ \exists\, y.\ y \in FV\ \varphi \wedge x \in functions\text{-}term\ ((Var\ y) \cdot \sigma')\}$)›
using *formsubst-def-switch Forall* **by** *auto*
also have ‹... = *functions-form* $\varphi \cup$ (*if* $(\exists\, y.\ y \in FV\ (\forall\, x.\ \varphi) \wedge x \in FVT\ (\sigma'\ y))$
then $\{xa.\ \exists\, y.\ y \in FV\ \varphi \wedge xa \in functions\text{-}term\ ((Var\ y) \cdot \sigma(x := Var\ z))\}$
else $\{x.\ \exists\, y.\ y \in FV\ \varphi \wedge x \in functions\text{-}term\ ((Var\ y) \cdot \sigma')\}$)›
by *auto*
finally show *?case*
using *fun-terms-set-eq* **by** *auto*
qed

lemma *formsubst-predicates*: ‹*predicates-form* $(\varphi \cdot_{fm} \sigma) =$ *predicates-form* φ›
proof (*induction* φ *arbitrary*: σ *rule*: *predicates-form.induct*)
case (4 x φ)

then show *?case*
by (*metis* (*no-types*, *lifting*) *formsubst.simps*(*4*) *predicates-form.simps*(*4*))
qed *auto*

lemma *formsubst-language-rename*: ‹*language* {$\varphi \cdot_{fm}$ *subst x* (*Var y*)} = *language* {φ}›
using *lang-singleton formsubst-predicates formsubst-functions-form* **by** (*simp add*: *subst-def*)

end

theory *FOL-Semantics*
imports *FOL-Syntax*
begin

locale *struct* =
fixes
M :: ‹'*m set*› **and**
FN :: ‹*nat* ⇒ '*m list* ⇒ '*m*› **and**
REL :: ‹*nat* ⇒ '*m list set*›
assumes
M-nonempty: ‹*M* ≠ {}›

typedef '*m intrp* =
‹{ (*M* :: '*m set*, *FN* :: *nat* ⇒ '*m list* ⇒ '*m*, *REL* :: *nat* ⇒ '*m list set*). *struct M*}›
using *struct.intro*
by *fastforce*

declare *Abs-intrp-inverse* [*simp*] *Rep-intrp-inverse* [*simp*]

setup-lifting *type-definition-intrp*

lift-definition *dom* :: ‹'*m intrp* ⇒ '*m set*› **is** *fst* .
lift-definition *intrp-fn* :: ‹'*m intrp* ⇒ (*nat* ⇒ '*m list* ⇒ '*m*)› **is** ‹*fst* ∘ *snd*› .
lift-definition *intrp-rel* :: ‹'*m intrp* ⇒ (*nat* ⇒ '*m list set*)› **is** ‹*snd* ∘ *snd*› .

lemma *intrp-is-struct* [*iff*]: ‹*struct* (*dom* $\mathcal{M}$)›
by *transfer auto*

lemma *dom-Abs-is-fst* [*simp*]: ‹*struct M* ⟹ *dom* (*Abs-intrp* (*M*, *FN*, *REL*)) = *M*›
by (*simp add*: *dom.rep-eq*)

lemma *intrp-fn-Abs-is-fst-snd* [*simp*]: ‹*struct M* ⟹ *intrp-fn* (*Abs-intrp* (*M*, *FN*, *REL*)) = *FN*›
by (*simp add*: *intrp-fn.rep-eq*)

lemma *intrp-rel-Abs-is-snd-snd* [*simp*]:

‹*struct M* ⟹ *intrp-rel* (*Abs-intrp* (*M*, *FN*, *REL*)) = *REL*›
by (*simp add*: *intrp-rel.rep-eq*)

definition *is-valuation* :: ‹'*m intrp* ⇒ (*nat* ⇒ '*m*) ⇒ *bool*› **where**
‹*is-valuation* $\mathcal{M}$ β ⟷ (∀ *v*. β *v* ∈ *dom* $\mathcal{M}$)›

lemma *valuation-valmod*: ‹⟦*is-valuation* $\mathcal{M}$ β; *a* ∈ *dom* $\mathcal{M}$⟧ ⟹ *is-valuation* $\mathcal{M}$ (β(*x* := *a*))›
by (*simp add*: *is-valuation-def*)

fun *eval*
:: ‹*nterm* ⇒ '*m intrp* ⇒ (*nat* ⇒ '*m*) ⇒ '*m*›
(‹⟦-⟧⁻,⁻› [*50*, *0*, *0*] *70*) **where**
‹⟦*Var v*⟧⁻,β = β *v*›
| ‹⟦*Fun f ts*⟧$^{\mathcal{M},\beta}$ = *intrp-fn* $\mathcal{M}$ *f* [⟦*t*⟧$^{\mathcal{M},\beta}$. *t* ← *ts*]›

definition *list-all* :: ‹('*a* ⇒ *bool*) ⇒ '*a list* ⇒ *bool*› **where**
[*simp*]: ‹*list-all P ls* ⟷ (*fold* (λ*l b*. *b* ∧ *P l*) *ls True*)›

lemma *term-subst-eval*: ‹*intrp-fn M* = *Fun* ⟹ *t* · *v* = *eval t M v*›
by (*induction t*) *auto*

lemma *term-eval-triv*[*simp*]: ‹*intrp-fn M* = *Fun* ⟹ *eval t M Var* = *t*›
by (*metis subst-apply-term-empty term-subst-eval*)

lemma *fold-bool-prop*: ‹(*fold* (λ*l b*. *b* ∧ *P l*) *ls b*) = (*b* ∧ (∀ *l*∈*set ls*. *P l*))›
by (*induction ls arbitrary*: *b*) *auto*

lemma *list-all-set*: ‹*list-all P ls* = (∀ *l*∈*set ls*. *P l*)›
unfolding *list-all-def* **using** *fold-bool-prop* **by** *auto*

hide-const *lang*

definition *is-interpretation* **where**
‹*is-interpretation lang* $\mathcal{M}$ ⟷
((∀ *f l*. (*f*, *length*(*l*)) ∈ *fst lang* ∧ *set l* ⊆ *dom* $\mathcal{M}$ ⟶ *intrp-fn* $\mathcal{M}$ *f l* ∈ *dom* $\mathcal{M}$))›

lemma *interpretation-sublanguage*: ‹*funs2* ⊆ *funs1* ⟹ *is-interpretation* (*funs1*, *pred1*) $\mathcal{M}$
⟹ *is-interpretation* (*funs2*, *preds2*) $\mathcal{M}$›
unfolding *is-interpretation-def* **by** *auto*

lemma *interpretation-eval*:
assumes $\mathcal{M}$: ‹*is-interpretation* (*functions-term t*,*any*) $\mathcal{M}$› **and** *val*: ‹*is-valuation*

```
𝓜 β›
  shows ‹⟦t⟧^{𝓜,β} ∈ dom 𝓜›
  using 𝓜
proof (induction t)
  case (Var x)
  with val show ?case
    by (simp add: is-valuation-def)
next
  case (Fun f ts)
  then have ‹⟦t⟧^{𝓜,β} ∈ dom 𝓜› if ‹t ∈ set ts› for t
   by (meson interpretation-sublanguage supt.arg supt-imp-funas-term-subset that)
  then show ?case
    using Fun by (simp add: is-interpretation-def image-subsetI)
qed


fun holds
  :: ‹'m intrp ⇒ (nat ⇒ 'm) ⇒ form ⇒ bool› (‹-,- ⊨ -› [30, 30, 80] 80) where
  ‹𝓜,β ⊨ ⊥ ⟷ False›
| ‹𝓜,β ⊨ Atom p ts ⟷ [⟦t⟧^{𝓜,β}. t ← ts] ∈ intrp-rel 𝓜 p›
| ‹𝓜,β ⊨ φ ⟶ ψ ⟷ ((𝓜,β ⊨ φ) ⟶ (𝓜,β ⊨ ψ))›
| ‹𝓜,β ⊨ (∀ x. φ) ⟷ (∀ a ∈ dom 𝓜. 𝓜,β(x := a) ⊨ φ)›

lemma holds-exists: ‹𝓜,β ⊨ (∃ x. φ) ⟷ (∃ a ∈ dom 𝓜. 𝓜,β(x := a) ⊨ φ)›
  by simp

lemma holds-indep-β-if:
  ‹∀ v ∈ FV φ. β₁ v = β₂ v ⟹ 𝓜,β₁ ⊨ φ ⟷ 𝓜,β₂ ⊨ φ›
proof (induction φ arbitrary: β₁ β₂)
  case Bot
  then show ?case
    by simp
next
  case (Atom p ts)
  then have ‹∀ t ∈ set ts. ∀ v ∈ FVT t. β₁ v = β₂ v›
    by simp
  then have ‹[⟦t⟧^{𝓜,β₁}. t ← ts] = [⟦t⟧^{𝓜,β₂}. t ← ts]›
  proof (induction ts)
    case Nil
    then show ?case
      by simp
  next
    case (Cons a ts)
    then show ?case
    proof (induction a)
      case (Var x)
      then show ?case
        by simp
    next
```

case (*Fun f ts'*)
then have ‹∀ $t \in set\ ts'.\ \forall\ v \in FVT\ t.\ \beta_1\ v = \beta_2\ v$›
by *simp*
then have ‹$[\![t]\!]^{\mathcal{M},\beta_1}.\ t \leftarrow ts'] = [\![t]\!]^{\mathcal{M},\beta_2}.\ t \leftarrow ts']$›
using *Cons.IH Fun.IH Fun.prems*(*2*)
by *force*
then have ‹*intrp-fn* $\mathcal{M}\ f\ [\![t]\!]^{\mathcal{M},\beta_1}.\ t \leftarrow ts'] =$ *intrp-fn* $\mathcal{M}\ f\ [\![t]\!]^{\mathcal{M},\beta_2}.\ t \leftarrow ts']$›
by *argo*
then show *?case*
using *Cons.IH Fun.prems*(*2*)
by *force*
qed
qed
then have ‹$[\![t]\!]^{\mathcal{M},\beta_1}.\ t \leftarrow ts] \in$ *intrp-rel* $\mathcal{M}\ p \longleftrightarrow [\![t]\!]^{\mathcal{M},\beta_2}.\ t \leftarrow ts] \in$ *intrp-rel* $\mathcal{M}\ p$›
by *argo*
then show *?case*
by *simp*
next
case (*Implies* $\varphi\ \psi$)
then have
‹$\forall\, v \in FV\ \varphi.\ \beta_1\ v = \beta_2\ v$› **and**
‹$\forall\, v \in FV\ \psi.\ \beta_1\ v = \beta_2\ v$›
by *auto*
then have
‹$\mathcal{M}$,$\beta_1 \models \varphi \longleftrightarrow \mathcal{M}$,$\beta_2 \models \varphi$› **and**
‹$\mathcal{M}$,$\beta_1 \models \psi \longleftrightarrow \mathcal{M}$,$\beta_2 \models \psi$›
using *Implies.IH* **by** *auto*
then show *?case*
by *simp*
next
case (*Forall x* φ)
then have ‹$\forall\, a \in dom\ \mathcal{M}.\ (\mathcal{M}$,$\beta_1(x := a) \models \varphi) = (\mathcal{M}$,$\beta_2(x := a) \models \varphi)$›
by *simp*
then show *?case*
by *simp*
qed

lemma *holds-indep-intrp-if*:
fixes
φ :: *form* **and**
$\mathcal{M}\ \mathcal{M}'$ :: ‹*'m intrp*›
assumes
dom-eq: ‹*dom* $\mathcal{M}$ = *dom* $\mathcal{M}'$› **and**
rel-eq: ‹∀ *p. intrp-rel* $\mathcal{M}\ p$ = *intrp-rel* $\mathcal{M}'\ p$› **and**
fn-eq: ‹∀ *f ts.* (*f*, *length ts*) ∈ *functions-form* $\varphi \longrightarrow$ *intrp-fn* $\mathcal{M}\ f\ ts$ = *intrp-fn* $\mathcal{M}'\ f\ ts$›
shows

```
    ‹∀ β.  M,β ⊨ φ ⟷ M',β ⊨ φ›
  using fn-eq
proof (intro allI impI, induction φ)
  case (Atom p ts)

  have all-fn-sym-in: ‹(⋃ t ∈ set ts. functions-term t) ⊆ functions-form (Atom p
ts)› (is ‹?A ⊆ -›)
    by simp

  have eval-tm-eq: ‹⟦t⟧^M,β = ⟦t⟧^M',β›
    if ‹functions-term t ⊆ functions-form (Atom p ts)›
    for t
    using that
  proof (induction t)
    case (Fun f ts')

    have ‹∀ t' ∈ set ts'. functions-term t' ⊆ functions-form (Atom p ts)›
      using Fun.prems
      by auto
    moreover have ‹(f, length [⟦t'⟧^M,β. t' ← ts']) ∈ functions-form (Atom p ts)›
      using Fun.prems
      by fastforce
    ultimately show ?case
      using Fun.IH Atom.prems(1)[rule-format, of f ‹[⟦t'⟧^M,β. t' ← ts']›]
      by (smt (verit, del-insts) eval.simps(2) map-eq-conv)
  qed auto

  have ‹M,β ⊨ Atom p ts ⟷ [⟦t⟧^M,β. t ← ts] ∈ intrp-rel M p›
    by simp
  also have ‹... ⟷ [⟦t⟧^M,β. t ← ts] ∈ intrp-rel M' p›
    by (simp add: rel-eq)
  also have ‹... ⟷ [⟦t⟧^M',β. t ← ts] ∈ intrp-rel M' p›
    using eval-tm-eq all-fn-sym-in
    by (metis (mono-tags, lifting) UN-subset-iff map-eq-conv)
  also have ‹... ⟷ M',β ⊨ Atom p ts›
    by auto
  finally show ?case .
next
  case (Forall x φ)

  have ‹M,β ⊨ (∀ x. φ) ⟷ (∀ a ∈ dom M. M,β(x := a) ⊨ φ)›
    by simp
  also have ‹... = (∀ a ∈ dom M. M',β(x := a) ⊨ φ)›
    using Forall.IH Forall.prems by simp
  also have ‹... = (∀ a ∈ dom M'. M',β(x := a) ⊨ φ)›
    by (simp add: dom-eq)
  also have ‹... = (M',β ⊨ (∀ x. φ))›
    by auto
  finally show ?case .
```

qed *auto*

the above in a more idiomatic form (it is a congruence rule)

corollary *holds-cong*:
assumes
‹*dom* $\mathcal{M}$ = *dom* $\mathcal{M}'$›
‹$\bigwedge p$. *intrp-rel* $\mathcal{M}$ p = *intrp-rel* $\mathcal{M}'$ p›
‹$\bigwedge f$ *ts*. (f, *length ts*) ∈ *functions-form* φ ⟹ *intrp-fn* $\mathcal{M}$ f *ts* = *intrp-fn* $\mathcal{M}'$ f *ts*›
shows ‹$\mathcal{M}$,β ⊨ φ ⟷ $\mathcal{M}'$,β ⊨ φ›
using *assms holds-indep-intrp-if* **by** *blast*

abbreviation (*input*) ‹*termsubst* $\mathcal{M}$ β σ v ≡ $[\![\sigma\ v]\!]^{\mathcal{M},\beta}$›

lemma *subst-lemma-terms*: ‹$[\![t \cdot \sigma]\!]^{\mathcal{M},\beta}$ = $[\![t]\!]^{\mathcal{M},termsubst\ \mathcal{M}\ \beta\ \sigma}$›
proof (*induction t*)
case (*Var v*)
then show *?case*
by *auto*
next
case (*Fun f ts*)

have ‹$[\![Fun\ f\ ts \cdot \sigma]\!]^{\mathcal{M},\beta}$ = $[\![Fun\ f\ [t \cdot \sigma.\ t \leftarrow ts]]\!]^{\mathcal{M},\beta}$›
by *auto*
also have ‹... = *intrp-fn* $\mathcal{M}$ f [$[\![t]\!]^{\mathcal{M},\beta}$. $t \leftarrow [t \cdot \sigma.\ t \leftarrow ts]$]›
by *auto*
also have ‹... = *intrp-fn* $\mathcal{M}$ f [$[\![t \cdot \sigma]\!]^{\mathcal{M},\beta}$. $t \leftarrow ts$]›
unfolding *map-map*
by (*meson comp-apply*)
also have ‹... = *intrp-fn* $\mathcal{M}$ f [$[\![t]\!]^{\mathcal{M},termsubst\ \mathcal{M}\ \beta\ \sigma}$. $t \leftarrow ts$]›
using *Fun.IH*
by (*smt* (*verit*, *best*) *map-eq-conv*)
also have ‹... = $[\![Fun\ f\ ts]\!]^{\mathcal{M},termsubst\ \mathcal{M}\ \beta\ \sigma}$›
by *auto*
finally show *?case* **.**
qed

lemma *eval-indep-β-if*:
assumes ‹∀ $v \in FVT\ t.\ \beta\ v = \beta'\ v$›
shows ‹$[\![t]\!]^{\mathcal{M},\beta}$ = $[\![t]\!]^{\mathcal{M},\beta'}$›
using *assms*
proof (*induction t*)
case (*Var v*)
then show *?case*
by *auto*
next
case (*Fun f ts*)
then show *?case*

by (*smt* (*verit*, *ccfv-SIG*) *eval.simps*(2) *map-eq-conv term.set-intros*(4))
qed

lemma *concat-map*: ‹[*f t*. *t* ← [*g t*. *t* ← *ts*]] = [*f* (*g t*). *t* ← *ts*]› **by** *simp*

lemma *swap-subst-eval*: ‹$\mathcal{M}$,β ⊨ (φ ·$_{fm}$ σ) ⟷ $\mathcal{M}$,(λ*v*. *termsubst* $\mathcal{M}$ β σ *v*) ⊨ φ›
proof (*induction* φ *arbitrary*: σ β)
case (*Atom p ts*)
have ‹$\mathcal{M}$,β ⊨ (*Atom p ts* ·$_{fm}$ σ) ⟷ $\mathcal{M}$,β ⊨ (*Atom p* [*t* · σ. *t* ← *ts*])›
by *auto*
also have ‹... ⟷ [[*t*]$^{\mathcal{M},\beta}$. *t* ← [*t* · σ. *t* ← *ts*]] ∈ *intrp-rel* $\mathcal{M}$ *p*›
by *auto*
also have ‹... ⟷ [[*t* · σ]$^{\mathcal{M},\beta}$. *t* ← *ts*] ∈ *intrp-rel* $\mathcal{M}$ *p*›
using *concat-map*[*of* λ*t*. [*t*]$^{\mathcal{M},\beta}$ λ*t*. *t* · σ] **by** *presburger*
also have ‹... ⟷ [[*t*]$^{\mathcal{M},termsubst\ \mathcal{M}\ \beta\ \sigma}$. *t* ← *ts*] ∈ *intrp-rel* $\mathcal{M}$ *p*›
using *subst-lemma-terms*[*of* - σ $\mathcal{M}$ β] **by** *auto*
finally show *?case*
by *simp*
next
case (*Forall x* φ)
define σ' **where** σ' = σ(*x* := *Var x*)
show *?case*
proof (*cases* ‹∃ *y*. *y* ∈ *FV* (∀ *x*. φ) ∧ *x* ∈ *FVT* (σ' *y*)›)
case *False*
then have ‹(∀ *x*. φ) ·$_{fm}$ σ = ∀ *x*. (φ ·$_{fm}$ σ')›
using *formsubst-def-switch* σ'*-def* **by** *fastforce*
then have ‹$\mathcal{M}$,β ⊨ ((∀ *x*. φ) ·$_{fm}$ σ) = (∀ *a* ∈ *dom* $\mathcal{M}$. $\mathcal{M}$,β(*x* := *a*) ⊨ (φ ·$_{fm}$ σ'))›
by *auto*
also have ‹... = (∀ *a* ∈ *dom* $\mathcal{M}$. $\mathcal{M}$,(λ*v*. [σ' *v*]$^{\mathcal{M},\beta(x := a)}$) ⊨ φ)›
using *Forall* **by** *blast*
also have ‹... = (∀ *a* ∈ *dom* $\mathcal{M}$. $\mathcal{M}$,(λ*v*. [σ' *v*]$^{\mathcal{M},\beta}$)(*x* := *a*) ⊨ φ)›
proof
assume *forward*: ‹∀ *a* ∈ *dom* $\mathcal{M}$. $\mathcal{M}$,(λ*v*. [σ' *v*]$^{\mathcal{M},\beta(x := a)}$) ⊨ φ›
show ‹∀ *a* ∈ *dom* $\mathcal{M}$. $\mathcal{M}$,(λ*v*. [σ' *v*]$^{\mathcal{M},\beta}$)(*x* := *a*) ⊨ φ›
proof
fix *a*
assume ‹*a* ∈ *dom* $\mathcal{M}$›
then have ‹$\mathcal{M}$,(λ*v*. [σ' *v*]$^{\mathcal{M},\beta(x := a)}$) ⊨ φ›
using *forward* **by** *blast*
moreover have ‹∀ *v*∈*FV* φ. (λ*v*. [σ' *v*]$^{\mathcal{M},\beta(x := a)}$) *v* = ((λ*v*. [$\sigma'$ *v*]$^{\mathcal{M},\beta}$)(*x* := *a*)) *v*›
proof
fix *v*
assume ‹*v* ∈ *FV* φ›
then have ‹*v* ≠ *x* ⟹ [σ' *v*]$^{\mathcal{M},\beta(x := a)}$ = ((λ*v*. [σ' *v*]$^{\mathcal{M},\beta}$)(*x* := *a*)) *v*›

by (*metis* (*mono-tags*, *lifting*) *DiffI FV.simps*(4) *False eval-indep-β-if fun-upd-other singletonD*)
moreover have ‹$v = x \Longrightarrow [\![\sigma'\ v]\!]^{\mathcal{M},\beta(x := a)} = ((\lambda v.\ [\![\sigma'\ v]\!]^{\mathcal{M},\beta})(x := a))\ v$›
using σ'*-def* **by** *auto*
ultimately show ‹$[\![\sigma'\ v]\!]^{\mathcal{M},\beta(x := a)} = ((\lambda v.\ [\![\sigma'\ v]\!]^{\mathcal{M},\beta})(x := a))\ v$›
by *simp*
qed
ultimately show ‹$\mathcal{M}$,$(\lambda v.\ [\![\sigma'\ v]\!]^{\mathcal{M},\beta})(x := a) \models \varphi$›
using *holds-indep-β-if* **by** *fast*
qed
next
assume *backward*: ‹$\forall\, a \in dom\ \mathcal{M}.\ \mathcal{M}$,$(\lambda v.\ [\![\sigma'\ v]\!]^{\mathcal{M},\beta})(x := a) \models \varphi$›
show ‹$\forall\, a \in dom\ \mathcal{M}.\ \mathcal{M}$,$(\lambda v.\ [\![\sigma'\ v]\!]^{\mathcal{M},\beta(x := a)}) \models \varphi$›
proof
fix a
assume ‹$a \in dom\ \mathcal{M}$›
then have ‹$\mathcal{M}$,$(\lambda v.\ [\![\sigma'\ v]\!]^{\mathcal{M},\beta})(x := a) \models \varphi$›
using *backward* **by** *blast*
moreover have ‹$\forall\, v{\in}FV\ \varphi.\ (\lambda v.\ [\![\sigma'\ v]\!]^{\mathcal{M},\beta(x := a)})\ v = ((\lambda v.\ [\![\sigma'\ v]\!]^{\mathcal{M},\beta})(x := a))\ v$›
proof
fix v
assume ‹$v \in FV\ \varphi$›
then have ‹$v \neq x \Longrightarrow [\![\sigma'\ v]\!]^{\mathcal{M},\beta(x := a)} = ((\lambda v.\ [\![\sigma'\ v]\!]^{\mathcal{M},\beta})(x := a))\ v$›
by (*metis* (*mono-tags*, *lifting*) *DiffI FV.simps*(4) *False eval-indep-β-if fun-upd-other singletonD*)
moreover have ‹$v = x \Longrightarrow [\![\sigma'\ v]\!]^{\mathcal{M},\beta(x := a)} = ((\lambda v.\ [\![\sigma'\ v]\!]^{\mathcal{M},\beta})(x := a))\ v$›
using σ'*-def* **by** *auto*
ultimately show ‹$[\![\sigma'\ v]\!]^{\mathcal{M},\beta(x := a)} = ((\lambda v.\ [\![\sigma'\ v]\!]^{\mathcal{M},\beta})(x := a))\ v$›
by *simp*
qed
ultimately show ‹$\mathcal{M}$,$(\lambda v.\ [\![\sigma'\ v]\!]^{\mathcal{M},\beta(x := a)}) \models \varphi$›
using *holds-indep-β-if* **by** *fast*
qed
qed
also have ‹$\ldots = (\mathcal{M}$,$(\lambda v.\ [\![\sigma\ v]\!]^{\mathcal{M},\beta}) \models (\forall\ x.\ \varphi))$›
by (*smt* (*verit*, *ccfv-SIG*) σ'*-def fun-upd-apply holds.simps*(4) *holds-indep-β-if*)
finally show *?thesis* **.**
next
case *True*
then have *x-ex*: ‹$\exists\, y.\ y \in FV\ \varphi - \{x\} \wedge x \in FVT\ (\sigma'\ y)$›
by *simp*
then have *x-in*: ‹$x \in FV\ (\varphi \cdot_{fm} \sigma')$›
using *formsubst-fv*
by *auto*
define z **where** ‹$z = variant\ (FV\ (\varphi \cdot_{fm} \sigma'))$›

then have ‹$z \neq x$›
using *x-in variant-form* **by** *auto*
have ‹$(\forall\ x.\ \varphi) \cdot_{fm} \sigma = \forall\ z.\ (\varphi \cdot_{fm} \sigma(x := Var\ z))$›
using *z-def True formsubst-def-switch* σ'*-def* **by** (*smt* (*verit*, *best*) *formsubst.simps*(*4*))
then have ‹$\mathcal{M},\beta \models ((\forall\ x.\ \varphi) \cdot_{fm} \sigma) = (\forall\ a \in dom\ \mathcal{M}.\ \mathcal{M},\beta(z := a) \models (\varphi \cdot_{fm} \sigma(x := Var\ z)))$›
by *auto*
also have ‹$... = (\forall\ a \in dom\ \mathcal{M}.\ \mathcal{M},(\lambda v.\ [\![(\sigma(x := Var\ z))\ v]\!]^{\mathcal{M},\beta(z := a)}) \models \varphi)$›
using *Forall* **by** *blast*
also have ‹$... = (\forall\ a \in dom\ \mathcal{M}.\ \mathcal{M},(\lambda v.\ [\![(\sigma(x := Var\ z))\ v]\!]^{\mathcal{M},\beta})(x := a) \models \varphi)$›
proof
assume *forward*: ‹$\forall\ a{\in}dom\ \mathcal{M}.\ \mathcal{M},(\lambda v.\ [\![(\sigma(x := Var\ z))\ v]\!]^{\mathcal{M},\beta(z := a)}) \models \varphi$›
show ‹$\forall\ a{\in}dom\ \mathcal{M}.\ \mathcal{M},(\lambda v.\ [\![(\sigma(x := Var\ z))\ v]\!]^{\mathcal{M},\beta})(x := a) \models \varphi$›
proof
fix *a*
assume ‹$a \in dom\ \mathcal{M}$›
then have ‹$\mathcal{M},(\lambda v.\ [\![(\sigma(x := Var\ z))\ v]\!]^{\mathcal{M},\beta(z := a)}) \models \varphi$›
using *forward* **by** *blast*
moreover have ‹$\forall\ v{\in}FV\ \varphi.\ (\lambda v.\ [\![(\sigma(x := Var\ z))\ v]\!]^{\mathcal{M},\beta(z := a)})\ v = ((\lambda v.\ [\![(\sigma(x := Var\ z))\ v]\!]^{\mathcal{M},\beta})(x := a))\ v$›
proof
fix *v*
assume *v-in*: ‹$v \in FV\ \varphi$›
then have ‹$v \neq x \Longrightarrow z \notin FVT\ (\sigma\ v)$›
using *z-def variant-form* **by** (*smt* (*verit*, *ccfv-threshold*) σ'*-def eval-term.simps*(*1*) *formsubst-fv fun-upd-other mem-Collect-eq*)
then have ‹$v \neq x \Longrightarrow [\![\sigma\ v]\!]^{\mathcal{M},\beta(z := a)} = [\![\sigma\ v]\!]^{\mathcal{M},\beta}$›
by (*simp add*: *eval-indep-*β*-if*)
then have ‹$v \neq x \Longrightarrow (\lambda v.\ [\![(\sigma(x := Var\ z))\ v]\!]^{\mathcal{M},\beta(z := a)})\ v = ((\lambda v.\ [\![(\sigma(x := Var\ z))\ v]\!]^{\mathcal{M},\beta})(x := a))\ v$›
by *auto*
moreover have ‹$v = x \Longrightarrow (\lambda v.\ [\![(\sigma(x := Var\ z))\ v]\!]^{\mathcal{M},\beta(z := a)})\ v = ((\lambda v.\ [\![(\sigma(x := Var\ z))\ v]\!]^{\mathcal{M},\beta})(x := a))\ v$›
by *auto*
ultimately show ‹$[\![(\sigma(x := Var\ z))\ v]\!]^{\mathcal{M},\beta(z := a)} = ((\lambda v.\ [\![(\sigma(x := Var\ z))\ v]\!]^{\mathcal{M},\beta})(x := a))\ v$›
by *auto*
qed
ultimately show ‹$\mathcal{M},(\lambda v.\ [\![(\sigma(x := Var\ z))\ v]\!]^{\mathcal{M},\beta})(x := a) \models \varphi$›
using *holds-indep-*β*-if* **by** *fast*

```
    qed
  next
    assume backward: ‹∀ a∈dom M. M,(λv. ⟦(σ(x := Var z)) v⟧^{M,β})(x := a)
⊨ φ›
    show ‹∀ a∈dom M. M,(λv. ⟦(σ(x := Var z)) v⟧^{M,β(z := a)}) ⊨ φ›
    proof
      fix a
      assume ‹a ∈ dom M›
      then have ‹M,(λv. ⟦(σ(x := Var z)) v⟧^{M,β})(x := a) ⊨ φ›
        using backward by auto
      moreover have ‹∀ v∈FV φ. (λv. ⟦(σ(x := Var z)) v⟧^{M,β(z := a)}) v =
        ((λv. ⟦(σ(x := Var z)) v⟧^{M,β})(x := a)) v›
      proof
        fix v
        assume v-in: ‹v ∈ FV φ›
        then have ‹v ≠ x ⟹ z ∉ FVT (σ v)›
       using z-def variant-form by (smt (verit, ccfv-threshold) σ'-def eval-term.simps(1)

            formsubst-fv fun-upd-other mem-Collect-eq)
        then have ‹v ≠ x ⟹ ⟦σ v⟧^{M,β(z := a)} = ⟦σ v⟧^{M,β}›
          by (simp add: eval-indep-β-if)
        then have ‹v ≠ x ⟹
        (λv. ⟦(σ(x := Var z)) v⟧^{M,β(z := a)}) v = ((λv. ⟦(σ(x := Var z)) v⟧^{M,β})(x
:= a)) v›
          by auto
        moreover have ‹v = x ⟹
        (λv. ⟦(σ(x := Var z)) v⟧^{M,β(z := a)}) v = ((λv. ⟦(σ(x := Var z)) v⟧^{M,β})(x
:= a)) v›
          by auto
        ultimately show
          ‹⟦(σ(x := Var z)) v⟧^{M,β(z := a)} = ((λv. ⟦(σ(x := Var z)) v⟧^{M,β})(x :=
a)) v›
          by auto
      qed
      ultimately show ‹M,(λv. ⟦(σ(x := Var z)) v⟧^{M,β(z := a)}) ⊨ φ›
        using holds-indep-β-if by fast
    qed
  qed
  also have ‹... = (∀ a ∈ dom M. M,(λv. ⟦σ v⟧^{M,β})(x := a) ⊨ φ)›
    by (smt (verit, ccfv-SIG) fun-upd-apply holds-indep-β-if)
  also have ‹... = (M,(λv. ⟦σ v⟧^{M,β}) ⊨ (∀ x. φ))›
    by auto
  finally show ?thesis .
 qed
qed auto

definition satisfies :: ‹'m intrp ⇒ form set ⇒ bool› where
 ‹satisfies M S ≡ (∀ β φ. is-valuation M β ⟶ φ ∈ S ⟶ M,β ⊨ φ)›
```

lemma *satisfies-iff-satisfies-sing*: ‹*satisfies M S* ⟷ (∀ φ∈*S*. *satisfies M* {φ})›
by (*auto simp*: *satisfies-def*)

end

theory *Ground-FOL-Compactness*
imports
FOL-Semantics
begin

fun *qfree* :: ‹*form* ⇒ *bool*› **where**
‹*qfree* ⊥ = *True*›
| ‹*qfree* (*Atom p ts*) = *True*›
| ‹*qfree* (φ ⟶ ψ) = (*qfree* φ ∧ *qfree* ψ)›
| ‹*qfree* (∀ *x*. φ) = *False*›

lemma *qfree-iff-BV-empty*: *qfree* φ ⟷ *BV* φ = {}
by (*induction* φ) *auto*

lemma *qfree-no-quantif*: ‹*qfree r* ⟹ ¬(∃ *x p*. *r* = ∀ *x*. *p*) ∧ ¬(∃ *x p*. *r* = ∃ *x*. *p*)›
using *qfree.simps*(*3*) *qfree.simps*(*4*) **by** *blast*

lemma *qfree-formsubst*: ‹*qfree* φ ≡ *qfree* (φ ·$_{fm}$ σ)›
proof (*induction* φ)
case (*Forall x* φ)
then show *?case*
using *formsubst-def-switch* **by** (*metis* (*no-types*, *lifting*) *formsubst.simps*(*4*) *qfree-no-quantif*)
qed *simp+*

fun *form-to-formula* :: *form* ⇒ (*nat*×*nterm list*) *formula* **where**
‹*form-to-formula* ⊥ = ⊥›
| ‹*form-to-formula* (*Atom p ts*) = *formula.Atom* (*p*,*ts*)›
| ‹*form-to-formula* (φ ⟶ ψ) = *Imp* (*form-to-formula* φ) (*form-to-formula* ψ)›
| ‹*form-to-formula* (∀ *x*. φ) = ⊥›

fun *pholds* :: ‹(*form* ⇒ *bool*) ⇒ *form* ⇒ *bool*› (‹- ⊨$_p$ -› [*30*, *80*] *80*) **where**
‹*I* ⊨$_p$ ⊥ ⟷ *False*›
| ‹*I* ⊨$_p$ *Atom p ts* ⟷ *I* (*Atom p ts*)›
| ‹*I* ⊨$_p$ φ ⟶ ψ ⟷ ((*I* ⊨$_p$ φ) ⟶ (*I* ⊨$_p$ ψ))›
| ‹*I* ⊨$_p$ (∀ *x*. φ) ⟷ *I* (∀ *x*. φ)›

definition *psatisfiable* :: *form set* ⇒ *bool* **where**

‹*psatisfiable* $S \equiv \exists I.\ \forall \varphi{\in}S.\ I \models_p \varphi$›

abbreviation *psatisfies* **where** ‹*psatisfies* $I\ \Phi \equiv \forall \varphi{\in}\Phi.$ *pholds* $I\ \varphi$›

definition *val-to-prop-val* :: (*form* ⇒ *bool*) ⇒ ((*nat* × *nterm list*) ⇒ *bool*) **where**
‹*val-to-prop-val* $I = (\lambda x.\ I\ (Atom\ (fst\ x)\ (snd\ x)))$›

lemma *pholds-Not*: ‹$I \models_p Not\ \varphi \longleftrightarrow \neg\ (I \models_p \varphi)$›
by *simp*

lemma *pentails-equiv*: ‹*qfree* $\varphi \Longrightarrow (I \models_p \varphi \equiv (val\text{-}to\text{-}prop\text{-}val\ I) \models (form\text{-}to\text{-}formula\ \varphi))$›
proof (*induction* φ)
case *Bot*
then show *?case*
unfolding *val-to-prop-val-def* **by** *simp*
next
case (*Atom x1 x2*)
then show *?case*
unfolding *val-to-prop-val-def* **by** *simp*
next
case (*Implies* $\varphi 1\ \varphi 2$)
then have ‹*qfree* $\varphi 1$› **and** ‹*qfree* $\varphi 2$› **by** *simp+*
then have ‹$I \models_p \varphi 1 = val\text{-}to\text{-}prop\text{-}val\ I \models form\text{-}to\text{-}formula\ \varphi 1$› **and**
‹$I \models_p \varphi 2 = val\text{-}to\text{-}prop\text{-}val\ I \models form\text{-}to\text{-}formula\ \varphi 2$›
using *Implies*(*1*) *Implies*(*2*) **by** *simp+*
then show *?case* **by** *simp*
next
case (*Forall x1* φ)
then have *False* **by** *simp*
then show *?case* **by** *simp*
qed

lemma *pentails-equiv-set*:
assumes *all-qfree*: ‹$\forall \varphi{\in}S.$ *qfree* φ›
shows ‹*psatisfiable* $S \equiv$ *sat* (*form-to-formula* ‘ S)›
proof −
{
assume *psat-s*: ‹*psatisfiable* S›
then obtain I **where** *I-is*: ‹$\forall \varphi{\in}S.\ I \models_p \varphi$›
unfolding *psatisfiable-def* **by** *blast*
define $\mathcal{A}$ **where** ‹$\mathcal{A} = val\text{-}to\text{-}prop\text{-}val\ I$›
then have ‹$\forall \varphi{\in}S.\ \mathcal{A} \models (form\text{-}to\text{-}formula\ \varphi)$›
using *pentails-equiv all-qfree I-is* **by** *blast*
then have ‹*sat* (*form-to-formula* ‘ S)›
unfolding *sat-def* **by** *blast*
}
moreover {
assume ‹*sat* (*form-to-formula* ‘ S)›

then obtain $\mathcal{A}$ **where** *a-is*: ‹$\forall\,\varphi{\in}S.\ \mathcal{A} \models$ *form-to-formula* φ›
by (*meson image-eqI sat-def*)
define I **where** *i-is*: ‹$I = (\lambda x.\ \mathcal{A}$ (*pred* x, *args* x))›
then have ‹$\mathcal{A}$ = *val-to-prop-val* I›
unfolding *val-to-prop-val-def* **by** *simp*
then have ‹$\forall\,\varphi{\in}S.\ I \models_p \varphi$›
using *pentails-equiv all-qfree a-is* **by** *blast*
then have ‹*psatisfiable* S›
unfolding *psatisfiable-def* **by** *auto*
}
ultimately show ‹*psatisfiable* $S \equiv$ *sat* (*form-to-formula* ‘ S)›
by *argo*
qed

definition *finsat* :: *form set* ⇒ *bool* **where**
‹*finsat* $S \equiv \forall\, T{\subseteq}S.$ *finite* $T \longrightarrow$ *psatisfiable* T›

lemma *finsat-fin-sat-eq*:
assumes *all-qfree*: ‹$\forall\,\varphi{\in}S.$ *qfree* φ›
shows ‹*finsat* $S \longleftrightarrow$ *fin-sat* (*form-to-formula* ‘ S)›
proof
assume *finsat-s*: ‹*finsat* S›
then show ‹*fin-sat* (*form-to-formula* ‘ S)›
unfolding *fin-sat-def finsat-def*
by (*metis* (*no-types*, *opaque-lifting*) *assms finite-subset-image pentails-equiv-set subset-eq*)
next
assume *fin-sat-s*: ‹*fin-sat* (*form-to-formula* ‘ S)›
show ‹*finsat* S›
unfolding *finsat-def*
by (*meson assms compactness fin-sat-s pentails-equiv-set psatisfiable-def subsetD*)
qed

lemma *psatisfiable-mono*: ‹*psatisfiable* $S \implies T \subseteq S \implies$ *psatisfiable* T›
unfolding *psatisfiable-def* **by** *blast*

lemma *finsat-mono*: ‹*finsat* $S \implies T \subseteq S \implies$ *finsat* T›
unfolding *finsat-def* **by** *blast*

lemma *finsat-satisfiable*: ‹*psatisfiable* $S \implies$ *finsat* S›
unfolding *psatisfiable-def finsat-def* **by** *blast*

lemma *prop-compactness*: ‹$(\forall\,\varphi \in S.$ *qfree* $\varphi) \implies$ *finsat* S = *psatisfiable* S›
by (*simp add*: *compactness finsat-fin-sat-eq finsat-satisfiable pentails-equiv-set*)

as above, more in the style of HOL Light

lemma *compact-prop*:
assumes ‹$\bigwedge B.$ ⟦*finite* B; $B \subseteq A$⟧ $\implies \exists\, I.$ *psatisfies* I B› **and** ‹$\bigwedge\varphi.\ \varphi \in A \longrightarrow$

qfree φ›
shows ‹$\exists\, I.$ *psatisfies* $I\ A$›
by (*metis assms finsat-def prop-compactness psatisfiable-def*)

Three results required for the FOL uniformity theorem

lemma *compact-prop-alt*:
assumes ‹$\bigwedge I.\ \exists \varphi{\in}A.\ I \models_p \varphi$› ‹$\bigwedge \varphi.\ \varphi \in A \longrightarrow$ *qfree* φ›
obtains B **where** ‹*finite* B› ‹$B \subseteq A$› ‹$\bigwedge I.\ \exists \varphi{\in}B.\ I \models_p \varphi$›
proof −
have ‹$\bigwedge \varphi.\ \varphi \in$ *FOL-Syntax.Not* ‘ $A \longrightarrow$ *qfree* φ›
using *assms* **by** *force*
moreover
have ‹$\nexists I.$ *psatisfies* I (*FOL-Syntax.Not* ‘ A)›
by (*simp add*: *assms*)
ultimately obtain B **where** B: ‹*finite* B› ‹$B \subseteq$ (*FOL-Syntax.Not* ‘ A)› ‹$\bigwedge I.\ \exists r{\in}B.\ \neg\ (I \models_p r)$›
using *compact-prop* [*of* ‹*FOL-Syntax.Not* ‘ A›] **by** *fastforce*
show *thesis*
proof
show ‹*finite* (*Not* −‘ B)›
using *form.inject* **by** (*metis* ‹*finite* B› *finite-vimageI injI*)
show ‹*Not* −‘ $B \subseteq A$›
using B **by** *auto*
show ‹$\exists \varphi \in$ *Not* −‘ $B.\ I \models_p \varphi$› **for** I
using B **by** *force*
qed
qed

lemma *finite-disj-lemma*:
assumes ‹*finite* A›
shows ‹$\exists \Phi.$ *set* $\Phi \subseteq A \wedge (\forall I.\ I \models_p$ *foldr* $(\vee)\ \Phi\ \bot \longleftrightarrow (\exists \varphi{\in}A.\ I \models_p \varphi))$›
using *assms*
proof (*induction* A)
case *empty*
then show *?case*
by *auto*
next
case (*insert* $\varphi\ A$)
then obtain Φ **where** ‹*set* $\Phi \subseteq A$› ‹$\forall I.\ I \models_p$ *foldr* $(\vee)\ \Phi\ \bot \longleftrightarrow (\exists \varphi{\in}A.\ I \models_p \varphi)$›
by *blast*
then show *?case*
by (*force simp*: *intro*!: *exI* [**where** $x{=}\varphi\#\Phi$])
qed

lemma *compact-disj*:
assumes ‹$\bigwedge I.\ \exists \varphi{\in}A.\ I \models_p \varphi$› ‹$\bigwedge \varphi.\ \varphi \in A \longrightarrow$ *qfree* φ›
obtains Φ **where** ‹*set* $\Phi \subseteq A$› ‹$\bigwedge I.\ I \models_p$ *foldr* $(\vee)\ \Phi\ \bot$›
by (*smt* (*verit*, *best*) *assms compact-prop-alt finite-disj-lemma order-trans*)

end

theory *Prenex-Normal-Form*
imports
Ground-FOL-Compactness
begin

inductive *is-prenex* :: *form* ⇒ *bool* **where**
‹*qfree* φ ⟹ *is-prenex* φ›
| ‹*is-prenex* φ ⟹ *is-prenex* (∀ *x*. φ)›
| ‹*is-prenex* φ ⟹ *is-prenex* (∃ *x*. φ)›

inductive-simps *is-prenex-simps* [*simp*]:
is-prenex Bot
is-prenex (*Atom p ts*)
is-prenex (φ ⟶ ψ)
is-prenex (∀ *x*. φ)

lemma *prenex-formsubst1*: ‹*is-prenex* φ ⟹ *is-prenex* (φ ·$_{fm}$ σ)›
proof (*induction* φ *arbitrary*: σ *rule*: *is-prenex.induct*)
case *1*
then show *?case* **using** *is-prenex.intros*(*1*) *qfree-formsubst*
by *blast*
next
case (*2* φ *x*)
then show *?case*
using *formsubst-def-switch* **by** (*metis* (*no-types*, *lifting*) *formsubst.simps*(*4*)
is-prenex.intros(*2*))
next
case (*3* φ *x*)
then show *?case*
using *formsubst-def-switch is-prenex.intros*(*3*)
by (*smt* (*verit*, *del-insts*) *formsubst.simps*(*1*) *formsubst.simps*(*3*) *formsubst.simps*(*4*))
qed

lemma *prenex-formsubst2*: ‹*is-prenex* (φ ·$_{fm}$ σ) ⟹ *is-prenex* φ›
proof (*induction* ‹φ ·$_{fm}$ σ› *arbitrary*: φ σ *rule*: *is-prenex.induct*)
case *1*
then show *?case*
using *is-prenex.intros*(*1*) *qfree-formsubst* **by** *auto*
next
case (*2* ψ *x* φ)
then obtain *y* φ' **where** *phi-is*: ‹φ = ∀ *y*. φ'›
using *formsubst-structure-all* **by** *metis*
then have ‹∃ σ'. ψ = φ'·$_{fm}$ σ'›
using *2*(*3*) **by** (*metis* (*no-types*, *lifting*) *form.sel*(*6*) *formsubst.simps*(*4*))

```
  then obtain σ' where ‹ψ = φ'·fm σ'›
    by blast
  then have ‹is-prenex φ'›
    using 2(2) by blast
  then show ?case
    using phi-is by (simp add: is-prenex.intros(2))
next
  case (3 ψ x φ)
  then obtain y φ' where phi-is: ‹φ = ∃ y. φ'›
    using formsubst-structure-ex by metis
  then have ‹∃ σ'. ψ = φ'·fm σ'›
    using 3(3) by (smt (verit, ccfv-threshold) form.inject(2) form.inject(3) form-
subst.simps(3)
        formsubst.simps(4))
  then obtain σ' where ‹ψ = φ'·fm σ'›
    by blast
  then have ‹is-prenex φ'›
    using 3(2) by blast
  then show ?case
    using phi-is by (simp add: is-prenex.intros(3))
qed

lemma prenex-formsubst: ‹is-prenex (φ ·fm σ) ≡ is-prenex φ›
  using prenex-formsubst1 prenex-formsubst2 by (smt (verit, ccfv-threshold))

lemma prenex-imp: ‹is-prenex (φ ⟶ ψ) ⟹
  qfree (φ ⟶ ψ) ∨ (ψ = ⊥ ∧ (∃ x φ'. is-prenex φ' ∧ φ = (∀ x. φ' ⟶ ⊥)))›
  by (metis form.distinct(11) form.inject(2) is-prenex.cases)

inductive universal :: form ⇒ bool where
  ‹qfree φ ⟹ universal φ›
| ‹universal φ ⟹ universal (∀ x. φ)›

inductive-simps universal-simps [simp]:
  universal Bot
  universal (Atom p ts)
  universal (φ ⟶ ψ)
  universal (∀ x. φ)

fun size :: form ⇒ nat where
  ‹size ⊥ = 1›
| ‹size (Atom p ts) = 1›
| ‹size (φ ⟶ ψ) = size φ + size ψ›
| ‹size (∀ x. φ) = 1 + size φ›

lemma wf-size: ‹wfP (λφ ψ. size φ < size ψ)›
  by (simp add: wfP-if-convertible-to-nat)

lemma size-indep-subst: ‹size (φ ·fm σ) = size φ›
```

proof (*induction* φ *arbitrary*: σ)
case (*Forall* x φ)
have ‹$\exists z\ \sigma'.(\forall x.\ \varphi) \cdot_{fm} \sigma = \forall z.\ (\varphi \cdot_{fm} \sigma')$›
by (*meson formsubst.simps*(*4*))
then obtain $z\ \sigma'$ **where** ‹$(\forall x.\ \varphi) \cdot_{fm} \sigma = \forall z.\ (\varphi \cdot_{fm} \sigma')$›
by *blast*
then have ‹*size* $((\forall x.\ \varphi) \cdot_{fm} \sigma) =$ *size* $(\forall z.\ (\varphi \cdot_{fm} \sigma'))$›
by *argo*
also have ‹... = *size* $(\forall x.\ \varphi)$›
using *Forall* **by** *auto*
finally show *?case* **.**
qed *auto*

lemma *prenex-distinct*: ‹$(\forall x.\ \varphi) \neq (\exists y.\ \psi)$›
by *auto*

lemma *uniq-all-x*: *Uniq* $(\lambda x.\ \exists p.\ r = \forall x.\ p)$
using *Uniq-def* **by** *blast*

lemma *uniq-all-p*: ‹*Uniq* $((\lambda p.\ r = \forall\,(THE\ x.\ \exists p.\ r = \forall x.\ p).\ p))$›
using *uniq-all-x Uniq-def*
by (*smt* (*verit*, *ccfv-threshold*) *form.inject*(*3*))

lemma *uniq-ex-x*: *Uniq* $(\lambda x.\ \exists p.\ r = \exists x.\ p)$
using *Uniq-def* **by** *blast*

lemma *uniq-ex-p*: ‹*Uniq* $((\lambda p.\ r = \exists\,(THE\ x.\ \exists p.\ r = \exists x.\ p).\ p))$›
using *uniq-ex-x Uniq-def*
by (*smt* (*verit*, *best*) *form.inject*(*2*) *form.inject*(*3*))

definition *ppat* :: (*nat* ⇒ *form* ⇒ *form*) ⇒ (*nat* ⇒ *form* ⇒ *form*) ⇒ (*form* ⇒ *form*) ⇒ *form* ⇒ *form* **where**
‹*ppat A B C r* = (*if* $(\exists x\ p.\ r = \forall x.\ p)$ *then*
A (*THE x*. $\exists p.\ r = \forall x.\ p$) (*THE p*. $r = \forall\,(THE\ x.\ \exists p.\ r = \forall x.\ p).\ p$)
else (*if* $\exists x\ p.\ r = \exists x.\ p$ *then*
B (*THE x*. $\exists p.\ r = \exists x.\ p$) (*THE p*. $r = \exists\,(THE\ x.\ \exists p.\ r = \exists x.\ p).\ p$)
else C r))›

lemma *ppat-simpA*: ‹$\forall x\ p.$ *ppat A B C* $(\forall x.\ p) =$ *A x p*›
unfolding *ppat-def* **by** *simp*

lemma *ppat-simpB*: ‹$\forall x\ p.$ *ppat A B C* $(\exists x.\ p) =$ *B x p*›
unfolding *ppat-def* **by** *simp*

lemma *ppat-last*: ‹$(\forall r.\ \neg(\exists x\ p.\ r = \forall x.\ p) \wedge \neg(\exists x\ p.\ r = \exists x.\ p)) \Longrightarrow$ *ppat A B C r* = *C r*›
by *blast*

lemma *ppat-last-qfree*: ‹*qfree r* ⟹ *ppat A B C r* = *C r*›
using *qfree-no-quantif ppat-last* **by** (*simp add*: *ppat-def*)

lemma *ppat-to-ex-qfree*:
‹(∃ *f*. (∀ *x p q*. *f p* (∀ *x*. *q*) = ((*A* :: *form* ⇒ *nat* ⇒ *form* ⇒ *form*) *p*) *x q*) ∧
(∀ *x p q*. *f p* (∃ *x*. *q*) = (*B p*) *x q*) ∧
(∀ *p q*. *qfree q* ⟶ *f p q* = (*C p*) *q*))›
proof
define *f* **where** ‹*f* = (λ*p q*. *ppat* (*A p*) (*B p*) (*C p*) *q*)›
have *A-eq*: ‹(∀ *x p q*. *ppat* (*A p*) (*B p*) (*C p*) (∀ *x*. *q*) = (*A p*) *x q*)› **and**
B-eq: ‹(∀ *x p q*. *ppat* (*A p*) (*B p*) (*C p*) (∃ *x*. *q*) = (*B p*) *x q*)›
unfolding *ppat-def* **by** *simp*+
have *C-eq*: ‹(∀ *p q*. *qfree q* ⟶ *ppat* (*A p*) (*B p*) (*C p*) *q* = (*C p*) *q*)›
using *ppat-last-qfree* **by** *blast*
show ‹(∀ *x p q*. *f p* (∀ *x*. *q*) = *A p x q*) ∧ (∀ *x p q*. *f p* (∃ *x*. *q*) = *B p x q*) ∧ (∀ *p q*. *qfree q* ⟶ *f p q* = (*C p*) *q*)›
using *A-eq B-eq C-eq* **unfolding** *f-def* **by** *blast*
qed

lemma *size-rec*:
‹∀ *f g x*. (∀ (*z*::*form*). *size z* < *size x* ⟶ (*f z* = *g z*)) ⟶ (*H f x* = *H g x*) ⟹ (∃ *f*. ∀ *x*. *f x* = *H f x*)›
using *wfrec* [*of measure size H*] **by** (*metis cut-apply in-measure wf-measure*)

abbreviation *prenex-right-forall* :: (*form* ⇒ *form* ⇒ *form*) ⇒ *form* ⇒ *nat* ⇒ *form* ⇒ *form* **where**
‹*prenex-right-forall* ≡
(λ*p φ x ψ*. (*let y* = *variant*(*FV φ* ∪ *FV* (∀ *x*. *ψ*)) *in* (∀ *y*. *p φ* (*ψ* ·$_{fm}$ (*subst x* (*Var y*))))))›

abbreviation *prenex-right-exists* :: (*form* ⇒ *form* ⇒ *form*) ⇒ *form* ⇒ *nat* ⇒ *form* ⇒ *form* **where**
‹*prenex-right-exists* ≡
(λ*p φ x ψ*. (*let y* = *variant*(*FV φ* ∪ *FV* (∃ *x*. *ψ*)) *in* (∃ *y*. *p φ* (*ψ* ·$_{fm}$ (*subst x* (*Var y*))))))›

lemma *prenex-right-ex*:
‹∃ *prenex-right*. (∀ *φ x ψ*. *prenex-right φ* (∀ *x*. *ψ*) = *prenex-right-forall prenex-right φ x ψ*)
∧ (∀ *φ x ψ*. *prenex-right φ* (∃ *x*. *ψ*) = *prenex-right-exists prenex-right φ x ψ*)
∧ (∀ *φ ψ*. *qfree ψ* ⟶ *prenex-right φ ψ* = (*φ* ⟶ *ψ*))›
proof −
have ‹∀ *φ*. ∃ *prenex-right-only*. ∀ *ψ*. *prenex-right-only ψ* = *ppat*
(λ*x ψ*. (*let y* = *variant*(*FV φ* ∪ *FV* (∀ *x*. *ψ*)) *in* (∀ *y*. *prenex-right-only* (*ψ* ·$_{fm}$ (*subst x* (*Var y*))))))
(λ*x ψ*. (*let y* = *variant*(*FV φ* ∪ *FV* (∃ *x*. *ψ*)) *in* (∃ *y*. *prenex-right-only* (*ψ* ·$_{fm}$

```
(subst x (Var y))))))
    (λψ. (φ ⟶ ψ)) ψ›
  proof
    fix φ
    define A where ‹A = (λg x ψ. (let y = variant(FV φ ∪ FV (∀ x. ψ)) in (∀ y.
g (ψ ·fm (subst x (Var y))))))›
    define B where ‹B = (λp x ψ. (let y = variant(FV φ ∪ FV (∃ x. ψ)) in (∃ y.
p (ψ ·fm (subst x (Var y))))))›
    show ‹∃ prenex-right-only. ∀ ψ. prenex-right-only ψ =
      ppat (A prenex-right-only) (B prenex-right-only) (λψ. (φ ⟶ ψ)) ψ›
    proof (rule size-rec, (rule allI)+, (rule impI))
      fix prenex-right-only g:: form ⇒ form and ψ
      assume IH: ‹∀ z. size z < size ψ ⟶ prenex-right-only z = g z›
      show ‹ppat (A prenex-right-only) (B prenex-right-only) (λψ. (φ ⟶ ψ)) ψ =
        ppat (A g) (B g) (λψ. (φ ⟶ ψ)) ψ›
      proof (cases ∃ x ψ'. ψ = ∀ x. ψ')
        case True
        then obtain x ψ' where psi-is: ψ = ∀ x. ψ'
          by blast
        then have smaller: ‹size (ψ' ·fm σ) < size ψ› for σ
          using size-indep-subst by simp
        have ‹ppat (A prenex-right-only) (B prenex-right-only) (λψ. (φ ⟶ ψ)) ψ
=
          A prenex-right-only x ψ'›
          unfolding ppat-def by (simp add: psi-is)
        also have ‹... = A g x ψ'›
          unfolding A-def using IH smaller by presburger
        also have ‹... = ppat (A g) (B g) (λψ. (φ ⟶ ψ)) ψ›
          unfolding ppat-def by (simp add: psi-is)
        finally show ?thesis .
      next
        case False
        assume falseAll: ‹¬(∃ x ψ'. ψ = ∀  x. ψ')›
        then show ?thesis
        proof (cases ∃ x ψ'. ψ = ∃ x. ψ')
          case True
          then obtain x ψ' where psi-is: ψ = ∃ x. ψ'
            by blast
          then have smaller: ‹size (ψ' ·fm σ) < size ψ› for σ
            using size-indep-subst by simp
          have ‹ppat (A prenex-right-only) (B prenex-right-only) (λψ. (φ ⟶ ψ)) ψ
=
            B prenex-right-only x ψ'›
            unfolding ppat-def by (simp add: psi-is)
          also have ‹... = B g x ψ'›
            unfolding B-def using IH smaller by presburger
          also have ‹... = ppat (A g) (B g) (λψ. (φ ⟶ ψ)) ψ›
            unfolding ppat-def by (simp add: psi-is)
          finally show ?thesis .
```

```
      next
        case False
        then show ?thesis
          using falseAll ppat-last unfolding ppat-def by argo
      qed
    qed
  qed
  then have ‹∃ prenex-right. ∀ φ ψ. prenex-right φ ψ = ppat
              (prenex-right-forall prenex-right φ)
              (prenex-right-exists prenex-right φ)
              ((⟶) φ) ψ›
    using choice[of λφ p. ∀ ψ. p ψ =
          ppat (λx ψ. let y = variant (FV φ ∪ FV (∀ x. ψ)) in ∀ y. p (ψ ·fm subst
x (Var y)))
            (λx ψ. let y = variant (FV φ ∪ FV (∃ x. ψ)) in (∃ y. p (ψ ·fm subst x
(Var y))))
            ((⟶) φ) ψ] by blast
  then obtain prenex-right where prenex-right-is: ‹∀ φ ψ. prenex-right φ ψ =
      ppat (prenex-right-forall prenex-right φ) (prenex-right-exists prenex-right φ)
((⟶) φ) ψ›
    by blast

  have ‹∀ φ x ψ. prenex-right φ (∀ x. ψ) = prenex-right-forall prenex-right φ x ψ›
    using prenex-right-is unfolding ppat-def by simp
 moreover have ‹∀ φ x ψ. prenex-right φ (∃ x. ψ) = prenex-right-exists prenex-right
φ x ψ›
    using prenex-right-is unfolding ppat-def by simp
  moreover have ‹∀ φ ψ. qfree ψ ⟶ prenex-right φ ψ = (φ ⟶ ψ)›
    using prenex-right-is by (metis (no-types, lifting) ppat-last-qfree)
  ultimately show ?thesis
    by blast
qed


consts prenex-right :: form ⇒ form ⇒ form
specification (prenex-right) ‹
  (∀ φ x ψ. prenex-right φ (∀ x. ψ) = prenex-right-forall prenex-right φ x ψ) ∧
  (∀ φ x ψ. prenex-right φ (∃ x. ψ) = prenex-right-exists prenex-right φ x ψ) ∧
  (∀ φ ψ. qfree ψ ⟶ prenex-right φ ψ = (φ ⟶ ψ))›
  using prenex-right-ex by blast

lemma prenex-right-qfree-case: ‹qfree ψ ⟹ prenex-right φ ψ = (φ ⟶ ψ)›
proof −
  assume qfree-psi: qfree ψ
  have ‹((∀ φ x ψ. p φ (∀ x. ψ) = prenex-right-forall p φ x ψ) ∧
  (∀ φ x ψ. p φ (∃ x. ψ) = prenex-right-exists p φ x ψ) ∧
  (∀ φ ψ. qfree ψ ⟶ p φ ψ = (φ ⟶ ψ))) ⟹ (∀ φ ψ. qfree ψ ⟶ p φ ψ = (φ
⟶ ψ))› (is ?P p ⟹ ?Q p) for p
```

by *argo*
then have ‹$(\forall\, \varphi\ \psi.\ qfree\ \psi \longrightarrow prenex\text{-}right\ \varphi\ \psi = (\varphi \longrightarrow \psi))$›
using *someI2-ex*[*of ?P ?Q*] *prenex-right-def prenex-right-ex* **by** *presburger*
then show *?thesis*
using *qfree-psi* **by** *blast*
qed

lemma *prenex-right-all-case*: ‹*prenex-right* φ $(\forall\, x.\ \psi)$ = *prenex-right-forall prenex-right* φ *x* ψ›
proof −
have *all-cases-imp-all-case*: ‹$((\forall\, \varphi\ x\ \psi.\ p\ \varphi\ (\forall\, x.\ \psi) = prenex\text{-}right\text{-}forall\ p\ \varphi\ x\ \psi) \wedge$
$(\forall\, \varphi\ x\ \psi.\ p\ \varphi\ (\exists\, x.\ \psi) = prenex\text{-}right\text{-}exists\ p\ \varphi\ x\ \psi) \wedge$
$(\forall\, \varphi\ \psi.\ qfree\ \psi \longrightarrow p\ \varphi\ \psi = (\varphi \longrightarrow \psi))) \Longrightarrow$
$(p\ \varphi\ (\forall\, x.\ \psi) = prenex\text{-}right\text{-}forall\ p\ \varphi\ x\ \psi)$› (**is** *?P p* $\Longrightarrow$ *?Q p*) **for** *p*
by *meson*
then have ‹*prenex-right* φ $(\forall\, x.\ \psi)$ = *prenex-right-forall prenex-right* φ *x* ψ›
using *someI2-ex*[*of ?P ?Q*] *prenex-right-def prenex-right-ex* **by** *presburger*
then show *?thesis* **.**
qed

lemma *prenex-right-exist-case*: ‹*prenex-right* φ $(\exists\, x.\ \psi)$ = *prenex-right-exists prenex-right* φ *x* ψ›
proof −
have *ex-cases-imp-ex-case*: ‹$((\forall\, \varphi\ x\ \psi.\ p\ \varphi\ (\forall\, x.\ \psi) = prenex\text{-}right\text{-}forall\ p\ \varphi\ x\ \psi) \wedge$
$(\forall\, \varphi\ x\ \psi.\ p\ \varphi\ (\exists\, x.\ \psi) = prenex\text{-}right\text{-}exists\ p\ \varphi\ x\ \psi) \wedge$
$(\forall\, \varphi\ \psi.\ qfree\ \psi \longrightarrow p\ \varphi\ \psi = (\varphi \longrightarrow \psi))) \Longrightarrow$
$(p\ \varphi\ (\exists\, x.\ \psi) = prenex\text{-}right\text{-}exists\ p\ \varphi\ x\ \psi)$› (**is** *?P p* $\Longrightarrow$ *?Q p*) **for** *p*
by *meson*
then have ‹*prenex-right* φ $(\exists\, x.\ \psi)$ = *prenex-right-exists prenex-right* φ *x* ψ›
using *someI2-ex*[*of ?P ?Q*] *prenex-right-def prenex-right-ex* **by** *presburger*
then show *?thesis* **.**
qed

lemma *prenex-right-exists-shape-case*:
‹$\exists\, x2\ \sigma.\ prenex\text{-}right\ \varphi\ (\exists\, x.\ \psi) = \exists\, x2.\ prenex\text{-}right\ \varphi\ (\psi\ \cdot_{fm}\ \sigma)$›
proof −
have *all-cases-imp-all-case*: ‹$((\forall\, \varphi\ x\ \psi.\ p\ \varphi\ (\forall\, x.\ \psi) = prenex\text{-}right\text{-}forall\ p\ \varphi\ x\ \psi) \wedge$
$(\forall\, \varphi\ x\ \psi.\ p\ \varphi\ (\exists\, x.\ \psi) = prenex\text{-}right\text{-}exists\ p\ \varphi\ x\ \psi) \wedge$
$(\forall\, \varphi\ \psi.\ qfree\ \psi \longrightarrow p\ \varphi\ \psi = (\varphi \longrightarrow \psi))) \Longrightarrow$
$(\exists\, x2\ \sigma.\ p\ \varphi\ (\exists\, x.\ \psi) = \exists\, x2.\ p\ \varphi\ (\psi\ \cdot_{fm}\ \sigma))$› (**is** *?P p* $\Longrightarrow$ *?Q p*) **for** *p*
by *meson*
then have ‹$\exists\, x2\ \sigma.\ prenex\text{-}right\ \varphi\ (\exists\, x.\ \psi) = \exists\, x2.\ prenex\text{-}right\ \varphi\ (\psi\ \cdot_{fm}\ \sigma)$›
using *someI2-ex*[*of ?P ?Q*] *prenex-right-def prenex-right-ex* **by** *presburger*
then show *?thesis* **.**
qed

abbreviation *prenex-left-forall* :: (*form* ⇒ *form* ⇒ *form*) ⇒ *form* ⇒ *nat* ⇒ *form* ⇒ *form* **where**
‹*prenex-left-forall* ≡
(λ*p* φ *x* ψ. (*let* *y* = *variant*(*FV* (∀ *x.* φ) ∪ *FV* ψ) *in* (∃ *y.* *p* (φ ·$_{fm}$ (*subst* *x* (*Var* *y*))) ψ)))›

abbreviation *prenex-left-exists* :: (*form* ⇒ *form* ⇒ *form*) ⇒ *form* ⇒ *nat* ⇒ *form* ⇒ *form* **where**
‹*prenex-left-exists* ≡
(λ*p* φ *x* ψ. (*let* *y* = *variant*(*FV* (∃ *x.* φ) ∪ *FV* ψ) *in* (∀ *y.* *p* (φ ·$_{fm}$ (*subst* *x* (*Var* *y*))) ψ)))›

lemma *prenex-left-ex*:
‹∃ *prenex-left*. (∀ φ *x* ψ. *prenex-left* (∀ *x.* φ) ψ = *prenex-left-forall* *prenex-left* φ *x* ψ)
∧ (∀ φ *x* ψ. *prenex-left* (∃ *x.* φ) ψ = *prenex-left-exists* *prenex-left* φ *x* ψ)
∧ (∀ φ ψ. *qfree* φ ⟶ *prenex-left* φ ψ = *prenex-right* φ ψ)›
proof −
have ‹∀ ψ. ∃ *prenex-left-only*. ∀ φ. *prenex-left-only* φ = *ppat*
(λ*x* φ. (*let* *y* = *variant*(*FV* (∀ *x.* φ) ∪ *FV* ψ) *in* (∃ *y.* *prenex-left-only* (φ ·$_{fm}$ (*subst* *x* (*Var* *y*))))))
(λ*x* φ. (*let* *y* = *variant*(*FV* (∃ *x.* φ) ∪ *FV* ψ) *in* (∀ *y.* *prenex-left-only* (φ ·$_{fm}$ (*subst* *x* (*Var* *y*))))))
(λφ. *prenex-right* φ ψ) φ›
proof
fix ψ
define *A* **where** ‹*A* = (λ*g* *x* φ. (*let* *y* = *variant*(*FV* (∀ *x.* φ) ∪ *FV* ψ) *in* (∃ *y.* *g* (φ ·$_{fm}$ (*subst* *x* (*Var* *y*))))))›
define *B* **where** ‹*B* = (λ*p* *x* φ. (*let* *y* = *variant*(*FV* (∃ *x.* φ) ∪ *FV* ψ) *in* (∀ *y.* *p* (φ ·$_{fm}$ (*subst* *x* (*Var* *y*))))))›
show ‹∃ *prenex-left-only*. ∀ φ. *prenex-left-only* φ =
ppat (*A* *prenex-left-only*) (*B* *prenex-left-only*) (λφ. *prenex-right* φ ψ) φ›
proof (*rule* *size-rec*, (*rule* *allI*)+, (*rule* *impI*))
fix *prenex-left-only* *g*:: *form* ⇒ *form* **and** φ
assume *IH*: ‹∀ *z*. *size* *z* < *size* φ ⟶ *prenex-left-only* *z* = *g* *z*›
show ‹*ppat* (*A* *prenex-left-only*) (*B* *prenex-left-only*) (λφ. *prenex-right* φ ψ) φ =
ppat (*A* *g*) (*B* *g*) (λφ. *prenex-right* φ ψ) φ›
proof (*cases* ∃ *x* φ′. φ = ∀ *x.* φ′)
case *True*
then obtain *x* φ′ **where** *phi-is*: φ = ∀ *x.* φ′
by *blast*
then have *smaller*: ‹*size* (φ′ ·$_{fm}$ σ) < *size* φ› **for** σ
using *size-indep-subst* **by** *simp*
have ‹*ppat* (*A* *prenex-left-only*) (*B* *prenex-left-only*) (λφ. *prenex-right* φ ψ) φ =
A *prenex-left-only* *x* φ′›
unfolding *ppat-def* **by** (*simp* *add*: *phi-is*)

```
      also have ‹... = A g x φ'›
        unfolding A-def using IH smaller by presburger
      also have ‹... = ppat (A g) (B g) (λφ. prenex-right φ ψ) φ›
        unfolding ppat-def by (simp add: phi-is)
      finally show ?thesis .
    next
      case False
      assume falseAll: ‹¬(∃ x φ'. φ = ∀ x. φ')›
      then show ?thesis
      proof (cases ∃ x φ'. φ = ∃ x. φ')
        case True
        then obtain x φ' where phi-is: φ = ∃ x. φ'
          by blast
        then have smaller: ‹size (φ' ·fm σ) < size φ› for σ
          using size-indep-subst by simp
      have ‹ppat (A prenex-left-only) (B prenex-left-only) (λφ. prenex-right φ ψ)
φ =
        B prenex-left-only x φ'›
        unfolding ppat-def by (simp add: phi-is)
      also have ‹... = B g x φ'›
        unfolding B-def using IH smaller by presburger
      also have ‹... = ppat (A g) (B g) (λφ. prenex-right φ ψ) φ›
        unfolding ppat-def by (simp add: phi-is)
      finally show ?thesis .
      next
        case False
        then show ?thesis
          using falseAll ppat-last unfolding ppat-def by argo
      qed
    qed
  qed
  then have ‹∃ prenex-left-argswap. ∀ ψ φ. prenex-left-argswap ψ φ = ppat
    (λx φ. let y = variant (FV (∀ x. φ) ∪ FV ψ) in (∃ y. prenex-left-argswap ψ (φ
·fm subst x (Var y))))
    (λx φ. let y = variant (FV (∃ x. φ) ∪ FV ψ) in ∀ y. prenex-left-argswap ψ (φ
·fm subst x (Var y)))
    (λφ. prenex-right φ ψ) φ›
    using choice[of λψ p. ∀ φ. p φ =
          ppat (λx φ. let y = variant (FV (∀ x. φ) ∪ FV ψ) in (∃ y. p (φ ·fm
subst x (Var y))))
          (λx φ. let y = variant (FV (∃ x. φ) ∪ FV ψ) in ∀ y. p (φ ·fm subst x
(Var y)))
          (λφ. prenex-right φ ψ) φ] by blast
  then have ‹∃ prenex-left. ∀ φ ψ. prenex-left φ ψ = ppat
    (λx φ. let y = variant (FV (∀ x. φ) ∪ FV ψ) in (∃ y. prenex-left (φ ·fm subst
x (Var y)) ψ))
    (λx φ. let y = variant (FV (∃ x. φ) ∪ FV ψ) in ∀ y. prenex-left (φ ·fm subst
x (Var y)) ψ)
```

```
    (λφ. prenex-right φ ψ) φ›
    by force
  then obtain prenex-left where prenex-left-is: ‹∀ φ ψ. prenex-left φ ψ = ppat
    (λx φ. prenex-left-forall prenex-left φ x ψ)
    (λx φ. prenex-left-exists prenex-left φ x ψ)
    (λφ. prenex-right φ ψ) φ›
    by blast
  have ‹∀ φ x ψ. prenex-left (∀ x. φ) ψ =  prenex-left-forall prenex-left φ x ψ›
    using prenex-left-is unfolding ppat-def by simp
  moreover have ‹∀ φ x ψ. prenex-left (∃ x. φ) ψ = prenex-left-exists prenex-left
φ x ψ›
    using prenex-left-is unfolding ppat-def by simp
  moreover have ‹∀ φ ψ. qfree φ ⟶ prenex-left φ ψ = prenex-right φ ψ›
    using prenex-left-is by (metis (no-types, lifting) ppat-last-qfree)
  ultimately show ?thesis
    by blast
qed

definition prenex-left where ‹prenex-left = (SOME prenex-left.
  (∀ φ x ψ. prenex-left (∀ x. φ) ψ = prenex-left-forall prenex-left φ x ψ) ∧
  (∀ φ x ψ. prenex-left (∃ x. φ) ψ = prenex-left-exists prenex-left φ x ψ) ∧
  (∀ φ ψ. qfree φ ⟶ prenex-left φ ψ = prenex-right φ ψ))›

lemma prenex-left-forall-case: ‹prenex-left (∀ x. φ) ψ = prenex-left-forall prenex-left
φ x ψ›
  unfolding prenex-left-def by (smt (verit, del-insts) prenex-left-ex some-eq-ex)

lemma prenex-left-qfree-case: ‹qfree φ ⟹ prenex-left φ ψ = prenex-right φ ψ›
  unfolding prenex-left-def by (smt (verit, del-insts) prenex-left-ex some-eq-ex)

lemma prenex-left-exists-case: ‹prenex-left (∃ x. φ) ψ = prenex-left-exists prenex-left
φ x ψ›
  unfolding prenex-left-def by (smt (verit, del-insts) prenex-left-ex some-eq-ex)

lemma prenex-left-exists-shape-case:
  ‹∃ x2 σ. prenex-left (∃ x. φ) ψ = ∀ x2. prenex-left (φ ·fm σ) ψ›
  using prenex-left-exists-case by metis


fun prenex where
  ‹prenex ⊥ = ⊥›
| ‹prenex (Atom p ts) = Atom p ts›
| ‹prenex (φ ⟶ ψ) = prenex-left (prenex φ) (prenex ψ)›
| ‹prenex (∀ x. φ) = ∀ x. (prenex φ)›

lemma holds-indep-forall:
  assumes y-notin: ‹y ∉ FV (∀ x. φ)›
  shows ‹(I,β ⊨ (∀ x. φ) ⟷ I,β ⊨ (∀ y. φ ·fm (subst x (Var y))))›
proof (cases ‹y = x›)
```

case *False*
then have *y-notin-phi*: ‹$y \notin FV\ \varphi$› **using** *y-notin* **by** *simp*
have *beta-equiv*: ‹$\forall\, w \in FV\ \varphi.\ (\lambda v.\ termsubst\ I\ (\beta(y := a))\ (subst\ x\ (Var\ y))\ v)\ w = (\beta(x := a))\ w$› **for** *a*
proof
fix *w*
assume *w-in*: ‹$w \in FV\ \varphi$›
have ‹$w = x \Longrightarrow (\lambda v.\ termsubst\ I\ (\beta(y := a))\ (subst\ x\ (Var\ y))\ v)\ w = (\beta(x := a))\ w$›
by *simp*
moreover have ‹$w \neq x \Longrightarrow (\lambda v.\ termsubst\ I\ (\beta(y := a))\ (subst\ x\ (Var\ y))\ v)\ w = (\beta(x := a))\ w$›
using *y-notin-phi* **by** (*metis w-in eval.simps*(*1*) *fun-upd-other subst-def*)
ultimately show ‹$(\lambda v.\ termsubst\ I\ (\beta(y := a))\ (subst\ x\ (Var\ y))\ v)\ w = (\beta(x := a))\ w$›
by *argo*
qed
have ‹$I,\beta \models (\forall x.\ \varphi) \equiv (\forall\, a \in dom\ I.\ I,\beta(x := a) \models \varphi)$›
by *simp*
also have ‹$... \equiv (\forall\, a \in dom\ I.\ I,(\lambda v.\ termsubst\ I\ (\beta(y := a))\ (subst\ x\ (Var\ y))\ v) \models \varphi)$›
using *holds-indep-β-if*[*OF beta-equiv*] **by** *presburger*
also have ‹$... \equiv (\forall\, a \in dom\ I.\ I,\beta(y := a) \models (\varphi \cdot_{fm} (subst\ x\ (Var\ y))))$›
using *swap-subst-eval*[*of I - φ subst x* (*Var y*)] **by** *presburger*
also have ‹$... \equiv (I,\beta \models (\forall\, y.\ \varphi \cdot_{fm} (subst\ x\ (Var\ y))))$›
by *simp*
finally show *?thesis*
by *argo*
qed *auto*

lemma *forall-imp-commute*:
assumes *y-notin*: ‹$y \notin FV\ \varphi$›
shows ‹$((I :: 'a\ intrp), \beta \models (\varphi \longrightarrow (\forall\, y.\ \psi)) \longleftrightarrow I, \beta \models (\forall\, y.\ \varphi \longrightarrow \psi))$›
proof −
have ‹$((I, \beta \models \varphi) \longrightarrow (\forall\, a \in dom\ I.\ I,\beta(y := a) \models \psi)) \longleftrightarrow$
$(\forall\, a \in dom\ I.\ (I,\beta(y := a) \models \varphi \longrightarrow I,\beta(y := a) \models \psi))$›
by (*smt* (*verit*, *del-insts*) *fun-upd-other holds-indep-β-if assms*)
then show *?thesis*
by *simp*
qed

lemma *forall-imp-exists*:
assumes *y-notin*: ‹$y \notin FV\ \psi$›
shows ‹$((I :: 'a\ intrp), \beta \models ((\forall\, y.\varphi) \longrightarrow \psi) \longleftrightarrow I, \beta \models (\exists\, y.\ (\varphi \longrightarrow \psi)))$›
proof −
have ‹$((\forall\, a \in dom\ I.\ I,\beta(y := a) \models \varphi) \longrightarrow (I, \beta \models \psi)) \longleftrightarrow (\exists\, a \in dom\ I.\ (I,\beta(y := a) \models \varphi \longrightarrow I,\beta \models \psi))$›
using *empty-iff list.set*(*1*)
by (*smt* (*verit*, *best*) *equals0I intrp-is-struct struct-def*)

also have ‹... $\longleftrightarrow$ ($\exists\, a \in dom\ I.\ (I,\beta(y := a) \models \varphi \longrightarrow I,\beta(y := a) \models \psi)$)›
using *holds-indep-β-if* **by** (*smt* (*verit*, *del-insts*) *fun-upd-other y-notin*)
finally show *?thesis*
by *simp*
qed

lemma *exists-imp-forall*:
assumes *y-notin*: ‹$y \notin FV\ \psi$›
shows ‹($I,\ \beta \models ((\exists\, y.\varphi) \longrightarrow\ \psi)\ \longleftrightarrow\ I,\ \beta \models (\forall\, y.\ (\varphi \longrightarrow \psi))$)›
proof −
have ‹($\exists\, a \in dom\ I.\ I,\beta(y := a) \models \varphi) \longrightarrow (I,\ \beta \models \psi) \equiv$
($\forall\, a \in dom\ I.\ (I,\beta(y := a) \models \varphi \longrightarrow I,\beta \models \psi)$)›
using *empty-iff list.set(1)* **by** (*smt* (*verit*, *ccfv-threshold*))
also have ‹... $\equiv$ ($\forall\, a \in dom\ I.\ (I,\beta(y := a) \models \varphi \longrightarrow I,\beta(y := a) \models \psi)$)›
using *holds-indep-β-if* **by** (*smt* (*verit*, *del-insts*) *fun-upd-other y-notin*)
finally show *?thesis*
by *simp*
qed

lemma *exists-imp-commute*:
assumes *y-notin*: ‹$y \notin FV\ \varphi$›
shows ‹(($I :: 'a\ intrp$), $\beta \models (\varphi \longrightarrow (\exists\, y.\ \psi)) \longleftrightarrow I,\ \beta \models (\exists\, y.\ \varphi \longrightarrow \psi)$)›
proof −
have ‹(($I,\ \beta \models \varphi) \longrightarrow (\exists\, a \in dom\ I.\ I,\beta(y := a) \models \psi)) \longleftrightarrow$
($\exists\, a \in dom\ I.\ (I,\ \beta \models \varphi) \longrightarrow (I,\beta(y := a) \models \psi)$)›
by (*smt* (*verit*) *equals0I intrp-is-struct struct-def*)
also have ‹... $\longleftrightarrow$ ($\exists\, a \in dom\ I.\ (I,\beta(y := a) \models \varphi \longrightarrow I,\beta(y := a) \models \psi)$)›
using *y-notin* **by** (*smt* (*verit*, *ccfv-threshold*) *fun-upd-other holds-indep-β-if*)
finally show *?thesis*
using *holds-exists* **by** *simp*
qed

lemma *holds-indep-exists*:
‹$y \notin FV\ (\exists\, x.\ \varphi) \Longrightarrow (I,\beta \models (\exists\, x.\ \varphi)\ \longleftrightarrow\ I,\beta \models (\exists\, y.\ \varphi \cdot_{fm} (subst\ x\ (Var\ y))))$›
by (*metis FV.simps*(*1*,*3*) *formsubst.simps*(*1*,*3*) *holds.simps*(*3*) *holds-indep-forall sup-bot.right-neutral*)

lemma *prenex-right-forall-is*:
assumes ‹$dom\ I \neq \{\}$›
shows ‹(($I,\ \beta \models \varphi \longrightarrow (\forall\, x.\ \psi))\ \longleftrightarrow$
($I,\ \beta \models (\forall\, (variant\ (FV\ \varphi \cup FV\ (\forall\, x.\ \psi))).$
$(\varphi \longrightarrow (\psi \cdot_{fm} (subst\ x\ (Var\ (variant\ (FV\ \varphi \cup FV\ (\forall\, x.\ \psi))))))))))$›
(**is** *?lhs* = *?rhs*)
proof −

define y **where** ‹$y = variant\ (FV\ \varphi \cup FV\ (\forall\, x.\ \psi))$›
then have *y-notin1*: ‹$y \notin FV\ \varphi$› **and** *y-notin2*: ‹$y \notin FV\ (\forall\, x.\ \psi)$›
using *variant-finite finite-FV* **by** (*meson UnCI finite-UnI*)+
have ‹*?lhs* $\longleftrightarrow$ $I, \beta \models (\varphi \longrightarrow (\forall\, y.\ \psi \cdot_{fm} (subst\ x\ (Var\ y))))$›
using *holds-indep-forall y-notin2*
by (*smt* (*verit*, *ccfv-SIG*) *holds.simps*(*3*))
also have ‹... $\longleftrightarrow$ $I, \beta \models (\forall\, y.\ \varphi \longrightarrow (\psi \cdot_{fm} (subst\ x\ (Var\ y))))$›
using *forall-imp-commute*[*OF y-notin1*, *of I* β $\psi \cdot_{fm} (subst\ x\ (Var\ y))$] **.**
finally show *?thesis*
unfolding *y-def* **.**
qed

lemma *prenex-right-exists-is*:
assumes ‹*dom I* $\neq \{\}$›
shows ‹$((I, \beta \models \varphi \longrightarrow (\exists\, x.\ \psi)) \longleftrightarrow$
$(I, \beta \models (\exists\, (variant\ (FV\ \varphi \cup FV\ (\exists\, x.\ \psi))).$
$(\varphi \longrightarrow (\psi \cdot_{fm} (subst\ x\ (Var\ (variant\ (FV\ \varphi \cup FV\ (\exists\, x.\ \psi))))))))))$›
(**is** *?lhs* = *?rhs*)
proof −
define y **where** ‹$y = variant\ (FV\ \varphi \cup FV\ (\exists\, x.\ \psi))$›
then have *y-notin1*: ‹$y \notin FV\ \varphi$› **and** *y-notin2*: ‹$y \notin FV\ (\exists\, x.\ \psi)$›
using *variant-finite finite-FV* **by** (*meson UnCI finite-UnI*)+
have ‹*?lhs* $\longleftrightarrow$ $I, \beta \models (\varphi \longrightarrow (\exists\, y.\ \psi \cdot_{fm} (subst\ x\ (Var\ y))))$›
using *holds-indep-exists y-notin2 holds-exists* **by** (*smt* (*verit*) *holds.simps*(*3*))
also have ‹... $\longleftrightarrow$ $I, \beta \models (\exists\, y.\ \varphi \longrightarrow (\psi \cdot_{fm} (subst\ x\ (Var\ y))))$›
using *exists-imp-commute*[*OF y-notin1*, *of I* β $\psi \cdot_{fm} (subst\ x\ (Var\ y))$] **.**
finally show *?thesis*
unfolding *y-def* **.**
qed

lemma *prenex-left-forall-is*:
assumes ‹*dom I* $\neq \{\}$›
shows ‹$(I, \beta \models ((\forall\, x.\ \varphi) \longrightarrow \psi)) \equiv (I, \beta \models (\exists\, (variant\ (FV\ (\forall\, x.\ \varphi) \cup FV\ \psi)).$
$((\varphi \cdot_{fm} (subst\ x\ (Var\ (variant\ (FV\ (\forall\, x.\ \varphi) \cup FV\ \psi))))) \longrightarrow \psi)))$›
using *forall-imp-exists holds-indep-forall holds.simps*(*3*)
by (*smt* (*verit*, *del-insts*) *FV.simps*(*3*) *UnI2 sup.commute variant-form*)

lemma *prenex-left-exists-is*:
assumes ‹*dom I* $\neq \{\}$›
shows ‹$(I, \beta \models ((\exists\, x.\ \varphi) \longrightarrow \psi)) \equiv (I, \beta \models (\forall\, (variant\ (FV\ (\exists\, x.\ \varphi) \cup FV\ \psi)).$
$((\varphi \cdot_{fm} (subst\ x\ (Var\ (variant\ (FV\ (\exists\, x.\ \varphi) \cup FV\ \psi))))) \longrightarrow \psi)))$›
using *exists-imp-forall holds-indep-exists holds.simps*(*3*)
by (*smt* (*verit*, *ccfv-SIG*) *FV.simps*(*3*) *UnCI finite-FV variant-finite*)

lemma *prenex-right-forall-FV*: ‹$FV\ (\varphi \longrightarrow (\forall\, x.\ \psi)) =$
$FV\ (\forall\, (variant\ (FV\ \varphi \cup FV\ (\forall\, x.\ \psi))).\ (\varphi \longrightarrow (\psi \cdot_{fm} (subst\ x\ (Var\ (variant$
$(FV\ \varphi \cup FV\ (\forall\, x.\ \psi))))))))$›
using *formsubst-rename*

by (*metis Diff-empty Diff-insert0 FV.simps(3) FV.simps(4) Un-Diff finite-FV variant-finite*)

lemma *prenex-right-exists-FV*: ‹*FV* (*φ* ⟶ (∃ *x*. *ψ*)) =
FV (∀ (*variant* (*FV φ* ∪ *FV* (∃ *x*. *ψ*))). (*φ* ⟶ (*ψ* ·$_{fm}$ (*subst x* (*Var* (*variant* (*FV φ* ∪ *FV* (∃ *x*. *ψ*)))))))›
using *formsubst-rename prenex-right-forall-FV* **by** *force*

lemma *prenex-left-forall-FV*: ‹*FV* ((∀ *x*. *φ*) ⟶ *ψ*) =
FV (∃ (*variant* (*FV* (∀ *x*. *φ*) ∪ *FV ψ*)). ((*φ* ·$_{fm}$ (*subst x* (*Var* (*variant* (*FV* (∀ *x*. *φ*) ∪ *FV ψ*))))) ⟶ *ψ*))›
using *formsubst-rename*
by (*metis Diff-idemp Diff-insert-absorb FV.simps(3) FV.simps(4) Un-Diff finite-FV variant-finite FV-exists*)

lemma *prenex-left-exists-FV*: ‹*FV* ((∃ *x*. *φ*) ⟶ *ψ*) =
FV (∀ (*variant* (*FV* (∃ *x*. *φ*) ∪ *FV ψ*)). ((*φ* ·$_{fm}$ (*subst x* (*Var* (*variant* (*FV* (∃ *x*. *φ*) ∪ *FV ψ*))))) ⟶ *ψ*))›
using *formsubst-rename FV-exists prenex-left-forall-FV* **by** *auto*

lemma *prenex-right-forall-language*: ‹*language* {*φ* ⟶ (∀ *x*. *ψ*)} =
language {∀ (*variant* (*FV φ* ∪ *FV* (∀ *x*. *ψ*))). (*φ* ⟶ (*ψ* ·$_{fm}$ (*subst x* (*Var* (*variant* (*FV φ* ∪ *FV* (∀ *x*. *ψ*)))))))}›
using *lang-singleton formsubst-functions-form formsubst-predicates formsubst-language-rename*
by *auto*

lemma *prenex-right-exists-language*: ‹*language* {*φ* ⟶ (∃ *x*. *ψ*)} =
language {∃ (*variant* (*FV φ* ∪ *FV* (∃ *x*. *ψ*))). (*φ* ⟶ (*ψ* ·$_{fm}$ (*subst x* (*Var* (*variant* (*FV φ* ∪ *FV* (∃ *x*. *ψ*)))))))}›
using *lang-singleton formsubst-functions-form formsubst-predicates formsubst-language-rename*
by *auto*

lemma *prenex-left-forall-language*: ‹*language* {(∀ *x*. *φ*) ⟶ *ψ*} =
language {∃ (*variant* (*FV* (∀ *x*. *φ*) ∪ *FV ψ*)). ((*φ* ·$_{fm}$ (*subst x* (*Var* (*variant* (*FV* (∀ *x*. *φ*) ∪ *FV ψ*))))) ⟶ *ψ*)}›
using *lang-singleton formsubst-functions-form formsubst-predicates formsubst-language-rename*
by *auto*

lemma *prenex-left-exists-language*: ‹*language* {(∃ *x*. *φ*) ⟶ *ψ*} =
language {∀ (*variant* (*FV* (∃ *x*. *φ*) ∪ *FV ψ*)). ((*φ* ·$_{fm}$ (*subst x* (*Var* (*variant* (*FV* (∃ *x*. *φ*) ∪ *FV ψ*))))) ⟶ *ψ*)}›
using *lang-singleton formsubst-functions-form formsubst-predicates formsubst-language-rename*
by *auto*

lemma *prenex-props-forall*: ‹*P* ∧ *FV φ* = *FV ψ* ∧ *language* {*φ*} = *language* {*ψ*}
∧
(∀ (*I* :: *'a intrp*) *β*. *dom I* ≠ {} ⟶ (*I*,*β* ⊨ *φ* ⟷ *I*,*β* ⊨ *ψ*)) ⟹

$P \wedge FV\ (\forall\, x.\ \varphi) = FV\ (\forall\, x.\ \psi) \wedge language\ \{(\forall\, x.\ \varphi)\} = language\ \{(\forall\, x.\ \psi)\} \wedge$
$(\forall\, (I :: 'a\ intrp)\ \beta.\ dom\ I \neq \{\} \longrightarrow (I,\beta \models (\forall\, x.\ \varphi) \longleftrightarrow I,\beta \models (\forall\, x.\ \psi)))$
›
using *lang-singleton* **by** *simp*

lemma *prenex-props-exists*: ‹$P \wedge FV\ \varphi = FV\ \psi \wedge language\ \{\varphi\} = language\ \{\psi\}$
$\wedge$
$(\forall\, (I :: 'a\ intrp)\ \beta.\ dom\ I \neq \{\} \longrightarrow (I,\beta \models \varphi \longleftrightarrow I,\beta \models \psi)) \Longrightarrow$
$P \wedge FV\ (\exists\, x.\ \varphi) = FV\ (\exists\, x.\ \psi) \wedge language\ \{(\exists\, x.\ \varphi)\} = language\ \{(\exists\, x.\ \psi)\} \wedge$
$(\forall\, (I :: 'a\ intrp)\ \beta.\ dom\ I \neq \{\} \longrightarrow (I,\beta \models (\exists\, x.\ \varphi) \longleftrightarrow I,\beta \models (\exists\, x.\ \psi)))$
›
using *lang-singleton* **by** *simp*

lemma *prenex-right-props-imp0*:
assumes ‹*qfree* φ›
shows ‹*is-prenex* $\psi \Longrightarrow$ *is-prenex* (*prenex-right* φ ψ)›
proof (*induction* ψ *rule*: *measure-induct-rule* [*of size*])
case (*less* ψ)
show *?case*
proof (*cases rule*: *is-prenex.cases*[*OF* ‹*is-prenex* ψ›])
case (*1* ξ)
then show *?thesis*
by (*simp add*: *assms prenex-right-qfree-case*)
next
case (*2* ξ *x*)
then have ‹*prenex-right* φ ψ = *prenex-right-forall prenex-right* φ *x* ξ›
using *prenex-right-all-case* **by** *blast*
then show ‹*is-prenex* (*prenex-right* φ ψ)›
using *less 2* **by** (*auto simp*: *Let-def prenex-formsubst1 size-indep-subst*)
next
case (*3* ξ *x*)
then have ‹$\exists\, y\ \sigma.$ *prenex-right* φ $\psi = \exists\, y.$ *prenex-right* φ ($\xi \cdot_{fm} \sigma$)›
using *prenex-right-exists-shape-case* **by** *presburger*
then obtain *y* σ **where** *pr-is*: ‹*prenex-right* φ $\psi = \exists\, y.$ *prenex-right* φ ($\xi \cdot_{fm}$ σ)›
by *blast*
have *size-xp*: ‹*size* ($\xi \cdot_{fm} \sigma$) < *size* ψ›
using *3*(*1*) *size-indep-subst* **by** *auto*
have ‹*is-prenex* ($\xi \cdot_{fm} \sigma$)›
using *3*(*2*) *prenex-formsubst1* **by** *blast*
then have ‹*is-prenex* (*prenex-right* φ ($\xi \cdot_{fm} \sigma$))›
using *less size-xp* **by** *blast*
then show *?thesis*
using *is-prenex.intros*(*3*) *pr-is* **by** *presburger*
qed
qed

lemma *prenex-right-props-imp*:
assumes ‹*qfree* φ›

shows ‹*is-prenex* $\psi \Longrightarrow$
is-prenex (*prenex-right* φ ψ) $\wedge$
FV (*prenex-right* φ ψ) = *FV* ($\varphi \longrightarrow \psi$) $\wedge$
language {*prenex-right* φ ψ} = *language* {($\varphi \longrightarrow \psi$)} $\wedge$
($\forall$ (*I* :: *'a intrp*) β. *dom I* $\neq$ {} $\longrightarrow$ ((*I*,β $\models$ (*prenex-right* φ ψ)) $\longleftrightarrow$ (*I*,β $\models$ ($\varphi \longrightarrow \psi$))))›
(**is** ‹*is-prenex* $\psi \Longrightarrow$ *?P* ψ›)
proof (*induction* ψ *rule*: *measure-induct-rule* [*of size*])
case (*less* ψ)
show *?case*
proof (*cases rule*: *is-prenex.cases*[*OF* ‹*is-prenex* ψ›])
case (*1* ξ)
then show *?thesis*
using *prenex-right-qfree-case* ‹*qfree* φ› *is-prenex.intros*(*1*) *qfree.simps*(*3*) **by** *presburger*
next
case (*2* ξ *x*)
have *pr-is1*:‹*prenex-right* φ ψ = *prenex-right-forall prenex-right* φ *x* ξ›
using *2 prenex-right-all-case* **by** *blast*
define *y* **where** ‹*y* = *variant* (*FV* φ $\cup$ *FV* ($\forall$ *x*. ξ))›
then have *pr-is2*: ‹*prenex-right* φ ψ = $\forall$ *y*. *prenex-right* φ (ξ $\cdot_{fm}$ *subst x* (*Var y*))›
using ‹*qfree* φ› *2*(*2*) *pr-is1* **unfolding** *y-def* **by** *meson*
have ‹*is-prenex* (ξ $\cdot_{fm}$ *subst x* (*Var y*))›
using *prenex-formsubst1 2*(*2*) **by** *presburger*
then have *p-xps*: ‹*?P* (ξ $\cdot_{fm}$ *subst x* (*Var y*))›
using *less 2*(*1*) *less-Suc-eq plus-1-eq-Suc size.simps*(*4*) *size-indep-subst* **by** *presburger*
have ‹*is-prenex* (*prenex-right* φ ψ)›
using *prenex-right-props-imp0* ‹*is-prenex* ψ› ‹*qfree* φ› **by** *blast*
moreover have ‹*FV* (*prenex-right* φ ψ) = *FV* ($\varphi \longrightarrow \psi$)›
using *prenex-right-forall-FV*[*of* φ *x* ξ] **by** (*metis 2*(*1*) *FV.simps*(*4*) *p-xps pr-is1 y-def*)
moreover have ‹*language* {*prenex-right* φ ψ} = *language* {$\varphi \longrightarrow \psi$}›
using *prenex-right-forall-language*
by (*metis 2*(*1*) *functions-form.simps*(*4*) *lang-singleton p-xps pr-is1 predicates-form.simps*(*4*) *y-def*)
moreover have ‹($\forall$ (*I* :: *'a intrp*) β. *dom I* $\neq$ {} $\longrightarrow$ *I*,β $\models$ *prenex-right* φ ψ = *I*,β $\models$ $\varphi \longrightarrow \psi$)›
by (*metis 2*(*1*) *holds.simps*(*4*) *p-xps pr-is2 prenex-right-forall-is y-def*)
ultimately show *?thesis*
by *blast*
next
case (*3* ξ *x*)
have *pr-is1*:‹*prenex-right* φ ψ = *prenex-right-exists prenex-right* φ *x* ξ›
using *3 prenex-right-exist-case* **by** *blast*
define *y* **where** ‹*y* = *variant* (*FV* φ $\cup$ *FV* ($\exists$ *x*. ξ))›
then have *pr-is2*: ‹*prenex-right* φ ψ = $\exists$ *y*. *prenex-right* φ (ξ $\cdot_{fm}$ *subst x* (*Var y*))›

using ‹*qfree* φ› *3(2) pr-is1* **unfolding** *y-def* **by** *meson*
have ‹*is-prenex* (ξ $\cdot_{fm}$ *subst x* (*Var y*))›
using *prenex-formsubst1 3(2)* **by** *presburger*
then have *p-xps*: ‹*?P* (ξ $\cdot_{fm}$ *subst x* (*Var y*))›
using *less 3(1) less-Suc-eq plus-1-eq-Suc size.simps size-indep-subst* **by** *simp*
have ‹*is-prenex* (*prenex-right* φ ψ)›
using *prenex-right-props-imp0* ‹*is-prenex* ψ› ‹*qfree* φ› **by** *blast*
moreover have ‹*FV* (*prenex-right* φ ψ) = *FV* (φ $\longrightarrow$ ψ)›
using *prenex-right-exists-FV*[*of* φ *x* ξ] **by** (*metis 3(1) FV.simps*(*4*) *FV-exists p-xps*
pr-is1 y-def)
moreover have ‹*language* {*prenex-right* φ ψ} = *language* {φ $\longrightarrow$ ψ}›
using *prenex-right-forall-language* **by** (*smt* (*verit*) *3(1) p-xps pr-is1*
prenex-props-exists prenex-right-exists-language y-def)
moreover have ‹($\forall$ (*I* :: *'a intrp*) β. *dom I* $\neq$ {} $\longrightarrow$ *I*,β $\models$ *prenex-right* φ ψ = *I*,β $\models$ φ $\longrightarrow$ ψ)›
by (*smt* (*verit*, *best*) *3(1) p-xps pr-is2 prenex-props-exists prenex-right-exists-is y-def*)
ultimately show *?thesis*
by *blast*
qed
qed

lemma *prenex-right-props*:
‹*qfree* φ $\wedge$ *is-prenex* ψ $\Longrightarrow$
is-prenex (*prenex-right* φ ψ) $\wedge$
FV (*prenex-right* φ ψ) = *FV* (φ $\longrightarrow$ ψ) $\wedge$
language {*prenex-right* φ ψ} = *language* {(φ $\longrightarrow$ ψ)} $\wedge$
($\forall$ (*I* :: *'a intrp*) β. *dom I* $\neq$ {} $\longrightarrow$ ((*I*,β $\models$ (*prenex-right* φ ψ)) $\longleftrightarrow$ (*I*,β $\models$ (φ $\longrightarrow$ ψ))))›
using *prenex-right-props-imp* **by** *meson*

lemma *prenex-left-props-imp0*:
assumes ‹*is-prenex* ψ›
shows ‹*is-prenex* φ $\Longrightarrow$ *is-prenex* (*prenex-left* φ ψ)›
proof (*induction* φ *rule*: *measure-induct-rule* [*of size*])
case (*less* φ)
show *?case*
proof (*cases rule*: *is-prenex.cases*[*OF* ‹*is-prenex* φ›])
case (*1* ξ)
then show *?thesis*
using ‹*is-prenex* φ› *prenex-right-props prenex-left-qfree-case* ‹*is-prenex* ψ› **by** *presburger*
next
case (*2* ξ *x*)
then have ‹*prenex-left* φ ψ = *prenex-left-forall prenex-left* ξ *x* ψ›
using *prenex-left-forall-case* **by** *blast*
then show ‹*is-prenex* (*prenex-left* φ ψ)›

using *less 2* **by** (*metis is-prenex.intros(3) lessI plus-1-eq-Suc prenex-formsubst1 size.simps(4) size-indep-subst*)
next
case (*3 ξ x*)
then have ‹∃ *y σ*. *prenex-left φ ψ* = ∀ *y*. *prenex-left* (*ξ* ·$_{fm}$ *σ*) *ψ*›
using *prenex-left-exists-shape-case* **by** *presburger*
then obtain *y σ* **where** *pr-is*: ‹*prenex-left φ ψ* = ∀ *y*. *prenex-left* (*ξ* ·$_{fm}$ *σ*) *ψ*›
by *blast*
have *size-xp*: ‹*size* (*ξ* ·$_{fm}$ *σ*) < *size φ*›
using *3(1) size-indep-subst* **by** *auto*
have ‹*is-prenex* (*ξ* ·$_{fm}$ *σ*)›
using *3(2) prenex-formsubst1* **by** *blast*
then have ‹*is-prenex* (*prenex-left* (*ξ* ·$_{fm}$ *σ*) *ψ*)›
using *less size-xp* **by** *blast*
then show *?thesis*
using *is-prenex.intros pr-is* **by** *presburger*
qed
qed

lemma *prenex-left-props-imp*:
assumes ‹*is-prenex ψ*›
shows ‹*is-prenex φ* ⟹
is-prenex (*prenex-left φ ψ*) ∧
FV (*prenex-left φ ψ*) = *FV* (*φ* ⟶ *ψ*) ∧
(*language* {(*prenex-left φ ψ*)} = *language* {(*φ* ⟶ *ψ*)}) ∧
(∀ (*I* :: 'a *intrp*) *β*. *dom I* ≠ {} ⟶ (*I*,*β* ⊨ *prenex-left φ ψ* ⟷ *I*,*β* ⊨ *φ* ⟶ *ψ*))›
(**is** ‹*is-prenex φ* ⟹ *?P φ*›)
proof (*induction φ rule*: *measure-induct-rule* [*of size*])
case (*less ξ*)
show *?case*
proof (*cases rule*: *is-prenex.cases*[*OF* ‹*is-prenex ξ*›])
case (*1 ξ'*)
then show *?thesis*
using *prenex-right-qfree-case* ‹*is-prenex ψ*›
by (*simp add*: *prenex-left-qfree-case prenex-right-props*)
next
case (*2 ξ' x*)
have *pr-is1*:‹*prenex-left ξ ψ* = *prenex-left-forall prenex-left ξ' x ψ*›
using *2 prenex-left-forall-case* **by** *blast*
define *y* **where** ‹*y* = *variant* (*FV* (∀ *x*. *ξ'*) ∪ *FV ψ*)›
then have *pr-is2*: ‹*prenex-left ξ ψ* = ∃ *y*. *prenex-left* (*ξ'* ·$_{fm}$ *subst x* (*Var y*)) *ψ*›
using ‹*is-prenex ψ*› *2(2) pr-is1* **unfolding** *y-def* **by** *meson*
have ‹*is-prenex* (*ξ'* ·$_{fm}$ *subst x* (*Var y*))›
using *prenex-formsubst1 2(2)* **by** *presburger*
then have *p-xps*: ‹*?P* (*ξ'* ·$_{fm}$ *subst x* (*Var y*))›
using *less 2(1) less-Suc-eq plus-1-eq-Suc size.simps(4) size-indep-subst* **by** *presburger*

have ‹*is-prenex* (*prenex-left* ξ ψ)›
using *prenex-left-props-imp0* ‹*is-prenex* ξ› ‹*is-prenex* ψ› **by** *blast*
moreover have ‹*FV* (*prenex-left* ξ ψ) = *FV* ($\xi \longrightarrow \psi$)›
using *prenex-left-forall-FV*[*of x* ξ' ψ] **by** (*metis 2*(*1*) *FV-exists p-xps pr-is1 y-def*)
moreover have ‹*language* {*prenex-left* ξ ψ} = *language* {$\xi \longrightarrow \psi$}›
using *prenex-left-forall-language*
by (*smt* (*verit*, *ccfv-threshold*) *2*(*1*) *p-xps pr-is1 prenex-props-exists y-def*)
moreover have ‹($\bigwedge$(*I* :: *'a intrp*) β. *dom I* $\neq$ {} $\Longrightarrow$ *I*,β $\models$ *prenex-left* ξ ψ = *I*,β $\models$ $\xi \longrightarrow \psi$)›
by (*metis 2*(*1*) *holds-exists p-xps pr-is2 prenex-left-forall-is y-def*)
ultimately show ‹*?P* ξ›
by *blast*
next
case (*3* ξ' *x*)
have *pr-is1*:‹*prenex-left* ξ ψ = *prenex-left-exists prenex-left* ξ' *x* ψ›
using *3 prenex-left-exists-case* **by** *blast*
define *y* **where** ‹*y* = *variant* (*FV* ($\exists$ *x*. ξ') $\cup$ *FV* ψ)›
then have *pr-is2*: ‹*prenex-left* ξ ψ = $\forall$ *y*. *prenex-left* ($\xi' \cdot_{fm}$ *subst x* (*Var y*)) ψ›
using ‹*is-prenex* ψ› *3*(*2*) *pr-is1* **unfolding** *y-def* **by** *meson*
have ‹*is-prenex* ($\xi' \cdot_{fm}$ *subst x* (*Var y*))›
using *prenex-formsubst1 3*(*2*) **by** *presburger*
then have *p-xps*: ‹*?P* ($\xi' \cdot_{fm}$ *subst x* (*Var y*))›
using *less 3*(*1*) *less-Suc-eq plus-1-eq-Suc size.simps size-indep-subst* **by** *simp*
have ‹*is-prenex* (*prenex-left* ξ ψ)›
using *prenex-left-props-imp0* ‹*is-prenex* ξ› ‹*is-prenex* ψ› **by** *blast*
moreover have ‹*FV* (*prenex-left* ξ ψ) = *FV* ($\xi \longrightarrow \psi$)›
using *prenex-left-exists-FV*[*of x* ξ' ψ] **by** (*metis 3*(*1*) *FV.simps*(*4*) *p-xps pr-is1 y-def*)
moreover have ‹*language* {*prenex-left* ξ ψ} = *language* {$\xi \longrightarrow \psi$}›
using *prenex-left-exists-language*[*of x* ξ' ψ]
by (*smt* (*verit*) *3*(*1*) *p-xps pr-is2 prenex-props-forall y-def*)
moreover have ‹($\forall$ (*I* :: *'a intrp*) β. *dom I* $\neq$ {} $\longrightarrow$
I,β $\models$ *prenex-left* ξ ψ = *I*,β $\models$ $\xi \longrightarrow \psi$)›
by (*metis 3*(*1*) *holds.simps*(*4*) *p-xps pr-is2 prenex-left-exists-is y-def*)
ultimately show ‹*?P* ξ›
by *blast*
qed
qed

lemma *prenex-left-props*:
‹*is-prenex* φ $\wedge$ *is-prenex* ψ $\Longrightarrow$
is-prenex (*prenex-left* φ ψ) $\wedge$
FV (*prenex-left* φ ψ) = *FV* ($\varphi \longrightarrow \psi$) $\wedge$
(*language* {(*prenex-left* φ ψ)} = *language* {($\varphi \longrightarrow \psi$)}) $\wedge$
($\forall$ (*I* :: *'a intrp*) β. *dom I* $\neq$ {} $\longrightarrow$ (*I*,β $\models$ *prenex-left* φ ψ $\longleftrightarrow$ *I*,β $\models$ $\varphi \longrightarrow \psi$))›
using *prenex-left-props-imp* **by** *meson*

theorem *prenex-props*: ‹*is-prenex* (*prenex* φ) $\wedge$ (*FV* (*prenex* φ) = *FV* φ) $\wedge$
(*language* {*prenex* φ} = *language* {φ}) $\wedge$
($\forall$ (*I* :: 'a *intrp*) β. *dom I* $\neq$ {} $\longrightarrow$ (*I*, β $\models$ (*prenex* φ)) $\longleftrightarrow$ (*I*, β $\models$ φ))›
proof (*induction* φ *rule*: *form.induct*)
case *Bot*
then show *?case*
by (*metis is-prenex.simps prenex.simps*(*1*) *qfree.simps*(*1*))
next
case (*Atom p ts*)
then show *?case*
using *is-prenex.intros*(*1*) *prenex.simps*(*2*) *qfree.simps*(*2*) **by** *presburger*
next
case (*Implies* φ ψ)
have ‹*is-prenex* (*prenex* (φ $\longrightarrow$ ψ))›
using *Implies prenex-left-props prenex.simps*(*3*) **by** *presburger*
moreover have ‹*FV* (*prenex* (φ $\longrightarrow$ ψ)) = *FV* (φ $\longrightarrow$ ψ)›
using *Implies prenex-left-props prenex.simps*(*3*) *FV.simps*(*3*) **by** *presburger*
moreover have ‹*language* {*prenex* (φ $\longrightarrow$ ψ)} = *language* {φ $\longrightarrow$ ψ}›
using *Implies prenex-left-props prenex.simps*(*3*) *lang-singleton*
functions-form.simps(*3*) *predicates-form.simps*(*3*) **by** (*metis prod.inject*)
moreover have ‹$\forall$ (*I*::'a *intrp*) β. *FOL-Semantics.dom I* $\neq$ {} $\longrightarrow$
I,β $\models$ *prenex* (φ $\longrightarrow$ ψ) = *I*,β $\models$ φ $\longrightarrow$ ψ›
using *Implies prenex-left-props holds.simps*(*3*) *prenex.simps*(*3*) **by** *metis*
ultimately show *?case* **by** *blast*
next
case (*Forall x* φ)
have ‹*is-prenex* (*prenex* ($\forall$ *x*. φ))›
using *Forall* **using** *is-prenex.intros*(*2*) *prenex.simps*(*4*) **by** *presburger*
moreover have *fv-indep-prenex*: ‹*FV* (*prenex* ($\forall$ *x*. φ)) = *FV* ($\forall$ *x*. φ)›
using *Forall FV.simps*(*4*) *prenex.simps*(*4*) **by** *presburger*
moreover have ‹*language* {*prenex* ($\forall$ *x*. φ)} = *language* {$\forall$ *x*. φ}›
using *Forall prenex.simps*(*4*) *functions-form.simps*(*4*) *predicates-form.simps*(*4*)
unfolding *language-def functions-forms-def predicates-def* **by** *simp*
moreover have ‹($\forall$ (*I* :: 'a *intrp*) β. *dom I* $\neq$ {} $\longrightarrow$ *I*,β $\models$ *prenex* ($\forall$ *x*. φ) =
I,β $\models$ ($\forall$ *x*. φ))›
using *Forall holds.simps*(*4*) **by** *simp*
ultimately show *?case* **by** *blast*
qed

corollary *is-prenex-prenex* [*simp*]: ‹*is-prenex* (*prenex* φ)›
and *FV-prenex* [*simp*]: ‹*FV* (*prenex* φ) = *FV* φ›
and *language-prenex* [*simp*]: ‹*language* {*prenex* φ} = *language* {φ}›
by (*auto simp*: *prenex-props*)

corollary *prenex-holds* [*simp*]: ‹*dom I* $\neq$ {} $\Longrightarrow$ (*I*,β $\models$ (*prenex* φ)) $\longleftrightarrow$ (*I*,β $\models$
φ)›
by (*simp add*: *prenex-props*)

lemma *prenex-satisfies* [*simp*]:
assumes *dom M* ≠ {}
shows *satisfies M* {*prenex φ*} ⟷ *satisfies M* {*φ*}
using *assms prenex-holds* **by** (*fastforce simp*: *satisfies-def*)

end

theory *Bumping*
imports
FOL-Semantics
HOL−Library.Nat-Bijection
begin

abbreviation *numpair* **where**
‹*numpair m n* ≡ *prod-encode* (*m*,*n*)›

abbreviation *numfst* **where**
‹*numfst k* ≡ *fst* (*prod-decode k*)›

abbreviation *numsnd* **where**
‹*numsnd k* ≡ *snd* (*prod-decode k*)›

definition *bump-intrp* :: *'m intrp* ⇒ *'m intrp* **where**
‹*bump-intrp* $\mathcal{M}$ = *Abs-intrp* ((*dom* $\mathcal{M}$), (λ*k zs*. (*intrp-fn* $\mathcal{M}$) (*numsnd k*) *zs*), (*intrp-rel* $\mathcal{M}$))›

lemma *bump-dom* [*simp*]: ‹*dom* (*bump-intrp* $\mathcal{M}$) = *dom* $\mathcal{M}$›
proof −
have *is-struct*: ‹*struct* (*dom* $\mathcal{M}$)›
by (*simp add*: *intrp-is-struct*)
then show *?thesis* **unfolding** *bump-intrp-def* **using** *dom-Abs-is-fst* **by** *blast*
qed

lemma *bump-intrp-fn* [*simp*]: ‹*intrp-fn* (*bump-intrp* $\mathcal{M}$) (*numpair 0 f*) *ts* = *intrp-fn* $\mathcal{M}$ *f ts*›
proof −
have *is-struct*: ‹*struct* (*dom* $\mathcal{M}$)›
by (*smt* (*verit*, *best*) *intrp-is-struct struct-def*)
then show *?thesis* **unfolding** *bump-intrp-def* **by** *simp*
qed

lemma *bump-intrp-rel* [*simp*]: ‹*intrp-rel* (*bump-intrp* $\mathcal{M}$) *n* = *intrp-rel* $\mathcal{M}$ *n*›
unfolding *bump-intrp-def*
by (*smt* (*verit*) *intrp-is-struct intrp-rel-Abs-is-snd-snd struct-def*)

fun *bump-nterm* :: *nterm* ⇒ *nterm* **where**
  ‹*bump-nterm* (*Var x*) = *Var x*›
| ‹*bump-nterm* (*Fun f ts*) = *Fun* (*numpair 0 f*) (*map bump-nterm ts*)›

fun *bump-form* :: *form* ⇒ *form* **where**
  ‹*bump-form* ⊥ = ⊥›
| ‹*bump-form* (*Atom p ts*) = *Atom p* (*map bump-nterm ts*)›
| ‹*bump-form* (*φ* ⟶ *ψ*) = (*bump-form φ*) ⟶ (*bump-form ψ*)›
| ‹*bump-form* (∀ *x*. *φ*) = ∀ *x*. (*bump-form φ*)›

lemma *bumpterm*: ‹⟦*t*⟧^*M*,*β* = ⟦*bump-nterm t*⟧^*bump-intrp M*,*β*›
proof (*induct t*)
  case (*Var x*)
  then show *?case*
    by *simp*
next
  case (*Fun f ts*)
  then have ‹*intrp-fn M f* [⟦*t*⟧^*M*,*β*. *t* ← *ts*] =
    intrp-fn M f [⟦*bump-nterm t*⟧^*bump-intrp M*,*β*. *t* ← *ts*]›
    by (*metis* (*no-types*, *lifting*) *map-eq-conv*)
  also have ‹... =
    intrp-fn (*bump-intrp M*) (*numpair 0 f*) [⟦*bump-nterm t*⟧^*bump-intrp M*,*β*. *t* ← *ts*]›
    by (*simp add*: *bump-intrp-fn*)
  also have ‹... =
   intrp-fn (*bump-intrp M*) (*numpair 0 f*) [⟦*t*⟧^*bump-intrp M*,*β*. *t* ← (*map bump-nterm ts*)]›
    using *map-eq-conv* **by** *fastforce*
  ultimately show *?case* **by** *auto*
qed

lemma *bump-intrp-rel-holds*: ‹(*map* (λ*t*. ⟦*t*⟧^*M*,*β*) *ts* ∈ *intrp-rel M n*) =
  (*map* ((λ*t*. ⟦*t*⟧^*bump-intrp M*,*β*) ∘ *bump-nterm*) *ts* ∈ *intrp-rel* (*bump-intrp M*) *n*)›
proof −
  have ‹(λ*t*. ⟦*t*⟧^*M*,*β*) = (λ*t*. ⟦*t*⟧^*bump-intrp M*,*β*) ∘ *bump-nterm*›
    using *bumpterm* **by** *fastforce*
  then have ‹*map* (λ*t*. ⟦*t*⟧^*M*,*β*) *ts* = *map* ((λ*t*. ⟦*t*⟧^*bump-intrp M*,*β*) ∘ *bump-nterm*) *ts*›
    by *simp*
  then show *?thesis*
    by (*metis bump-intrp-rel*)
qed

lemma *bumpform*: ‹*M*, *β* ⊨ *φ* = (*bump-intrp M*), *β* ⊨ (*bump-form φ*)›
proof (*induct φ arbitrary*: *β*)
  case *Bot*
  then show *?case*

```
    unfolding bump-intrp-def by auto
next
  case (Atom x1 x2)
  then show ?case
    using bump-intrp-rel-holds by auto
next
  case (Implies φ1 φ2)
  then show ?case
    unfolding bump-intrp-def by auto
next
  case (Forall x1 φ)
  have ‹(∀ a ∈ dom M. (bump-intrp M),β(x1 := a) ⊨ bump-form φ) =
    (∀ a ∈ dom M. M,β(x1 := a) ⊨ φ)›
    using Forall by presburger
  then show ?case
    by simp
qed

lemma functions-form-bumpform: ‹(f, m) ∈ functions-form (bump-form φ) ⟹
  ∃ k. (f = numpair 0 k) ∧ (k, m) ∈ functions-form φ›
proof (induct φ)
  case (Atom p ts)
  then have ‹∃ t∈set ts. (f, m) ∈ functions-term (bump-nterm t)› by simp
  then obtain t where t-in: ‹t ∈ set ts› and fm-in-t: ‹(f, m) ∈ functions-term
(bump-nterm t)›
    by blast
  have ‹∃ k. f = numpair 0 k ∧ (k, m) ∈ functions-term t›
    using fm-in-t
  proof (induction t)
    case (Var x)
    then show ?case by auto
  next
    case (Fun g us)
    have t-in-disj: ‹functions-term (bump-nterm (Fun g us)) =
      {((numpair 0 g), length us)} ∪ (⋃ u ∈ set us. functions-term (bump-nterm
u))›
      by simp
    then show ?case
    proof (cases (f, m) = ((numpair 0 g), length us))
      case True
      then show ?thesis by auto
    next
      case False
      then have ‹(f, m) ∈ (⋃ u ∈ set us. functions-term (bump-nterm u))›
        using t-in-disj
        using Fun.prems by blast
      then show ?thesis
        using Fun(1) by fastforce
    qed
```

qed
then have ‹∃ *k*. *f* = *numpair* 0 *k* ∧ (∃ *x*∈*set ts*. (*k*, *m*) ∈ *functions-term x*)›
using *t-in* **by** *blast*
then show *?case* **using** *Atom* **by** *simp*
qed *auto*

lemma *bumpform-interpretation*: ‹*is-interpretation* (*language* {φ}) M ⟹
is-interpretation (*language* {(*bump-form* φ)}) (*bump-intrp* M)›
unfolding *is-interpretation-def language-def*
by (*metis bump-dom bump-intrp-fn fst-conv functions-form-bumpform lang-singleton language-def*)

fun *unbump-nterm* :: *nterm* ⇒ *nterm* **where**
‹*unbump-nterm* (*Var x*) = *Var x*›
| ‹*unbump-nterm* (*Fun f ts*) = *Fun* (*numsnd f*) (*map unbump-nterm ts*)›

fun *unbump-form* :: *form* ⇒ *form* **where**
‹*unbump-form* ⊥ = ⊥›
| ‹*unbump-form* (*Atom p ts*) = *Atom p* (*map unbump-nterm ts*)›
| ‹*unbump-form* (φ ⟶ ψ) = (*unbump-form* φ) ⟶ (*unbump-form* ψ)›
| ‹*unbump-form* (∀ *x*. φ) = ∀ *x*. (*unbump-form* φ)›

lemma *unbumpterm* [*simp*]: *unbump-nterm* (*bump-nterm t*) = *t*
by (*induct t*) (*simp add*: *list.map-ident-strong*)+

lemma *unbumpform* [*simp*]: ‹*unbump-form* (*bump-form* φ) = φ›
by (*induct* φ) (*simp add*: *list.map-ident-strong*)+

definition *unbump-intrp* :: *'m intrp* ⇒ *'m intrp* **where**
‹*unbump-intrp* M = *Abs-intrp* (*dom* M, (λ*k zs*. (*intrp-fn* M) (*numpair* 0 *k*) *zs*), (*intrp-rel* M))›

lemma *unbump-term-intrp*: ‹⟦*bump-nterm t*⟧^{M,β} = ⟦*t*⟧^{*unbump-intrp* M,β}›
proof (*induct t*)
case (*Fun f ts*)
then show *?case*
unfolding *unbump-intrp-def*
by (*smt* (*verit*, *best*) *bump-nterm.simps*(2) *concat-map eval.simps*(2) *intrp-fn-Abs-is-fst-snd*
intrp-is-struct list.map-cong0 struct-def)
qed *simp*

lemma *unbump-holds*: ‹(M,β ⊨ *bump-form* φ) = (*unbump-intrp* M,β ⊨ φ)›
proof (*induct* φ *arbitrary*: β)

case *Bot*
then show *?case*
by *auto*
next
case (*Atom p ts*)
then show *?case*
unfolding *unbump-intrp-def* **using** *bump-intrp-def bumpform dom-Abs-is-fst functions-form-bumpform*
holds-indep-intrp-if intrp-fn-Abs-is-fst-snd intrp-is-struct intrp-rel-Abs-is-snd-snd struct-def
by (*smt* (*verit*, *ccfv-SIG*) *prod-encode-inverse snd-conv*)
next
case (*Implies φ1 φ2*)
then show *?case*
by *auto*
next
case (*Forall x φ*)
then show *?case*
by (*smt* (*verit*, *best*) *bump-form.simps*(*4*) *dom-Abs-is-fst holds.simps*(*4*) *intrp-is-struct*
struct-def unbump-intrp-def)
qed

abbreviation *numlist* **where**
‹*numlist ns* ≡ *list-encode ns*›

fun *num-of-term* :: *nterm* ⇒ *nat* **where**
‹*num-of-term* (*Var x*) = *numpair 0 x*›
| ‹*num-of-term* (*Fun f ts*) = *numpair 1* (*numpair f* (*numlist* (*map num-of-term ts*)))›

lemma *term-induct2*:
(⋀*x y*. *P* (*Var x*) (*Var y*))
⟹ (⋀*x g us*. *P* (*Var x*) (*Fun g us*))
⟹ (⋀*f ts y*. *P* (*Fun f ts*) (*Var y*))
⟹ (⋀*f ts g us*. (⋀*p q*. *p* ∈ *set ts* ⟹ *q* ∈ *set us* ⟹ *P p q*) ⟹ *P* (*Fun f ts*) (*Fun g us*))
⟹ *P t1 t2*
proof (*induction t2 arbitrary*: *t1*)
case (*Var y*)
then show *?case* **by** (*metis is-FunE is-VarE*)
next
case (*Fun g us*)
then have ‹*p* ∈ *set ts* ⟹ *q* ∈ *set us* ⟹ *P p q*› **for** *ts p q*
by *blast*
then show *?case*
using *Fun* **by** (*metis is-FunE is-VarE*)
qed

lemma *num-of-term-inj*: ‹*num-of-term s = num-of-term t ⟷ s = t*›
proof (*induction s t rule*: *term-induct2*)
case (*4 f ts g us*)
have ‹(*Fun f ts = Fun g us*) ⟹ *num-of-term* (*Fun f ts*) = *num-of-term* (*Fun g us*)›
by *auto*
moreover {
assume ‹*num-of-term* (*Fun f ts*) = *num-of-term* (*Fun g us*)›
then have ‹*numpair f* (*numlist* (*map num-of-term ts*)) = *numpair g* (*numlist* (*map num-of-term us*))›
by *auto*
then have *fun-eq*: ‹*f = g*› **and** *nl-eq*: ‹*numlist* (*map num-of-term ts*) = (*numlist* (*map num-of-term us*))›
by *auto*
then have *map num-of-term ts = map num-of-term us*
using *list-encode-eq* **by** *blast*
then have *args-eq*: ‹*ts = us*›
using *4* **by** (*metis list.inj-map-strong*)
have ‹*Fun f ts = Fun g us*›
using *fun-eq args-eq* **by** *simp*
}
ultimately show *?case* **by** *auto*
qed *auto*

fun *num-of-form* :: *form* ⇒ *nat* **where**
‹*num-of-form* ⊥ = *numpair 0 0*›
| ‹*num-of-form* (*Atom p ts*) = *numpair 1* (*numpair p* (*numlist* (*map num-of-term ts*)))›
| ‹*num-of-form* (*φ* ⟶ *ψ*) = *numpair 2* (*numpair* (*num-of-form φ*) (*num-of-form ψ*))›
| ‹*num-of-form* (∀ *x*. *φ*) = *numpair 3* (*numpair x* (*num-of-form φ*))›

lemma *numlist-num-of-term*: ‹*numlist* (*map num-of-term ts*) = (*numlist* (*map num-of-term us*)) ≡ *ts = us*›
by (*smt* (*verit*) *list.inj-map-strong list-encode-eq num-of-term-inj*)

lemma *num-of-form-inj*: ‹*num-of-form φ = num-of-form ψ ⟷ φ = ψ*›
proof
show ‹*num-of-form φ = num-of-form ψ ⟹ φ = ψ*›
proof (*induct φ arbitrary*: *ψ rule*: *num-of-form.induct*)
case *1*
then show *?case*
using *num-of-form.elims num-of-form.simps*(*1*) *zero-neq-numeral zero-neq-one*
by (*metis prod.sel*(*1*) *prod-encode-inverse*)
next
case (*2 p ts*)
then show *?case*

```
    proof (cases ψ)
      case Bot
      then show ?thesis
        using 2 num-of-term-inj by fastforce
    next
      case (Atom q us)
      then show ?thesis
        using 2 by (simp add: numlist-num-of-term)
    next
      case (Implies ψ1 ψ2)
      then show ?thesis
        using 2 by simp
    next
      case (Forall y ψ1)
      then have ‹∃ k. num-of-form ψ = numpair 3 k›
        by auto
      moreover have ‹∃ k'. num-of-form (Atom p ts) = numpair 1 k'›
        by auto
      ultimately show ?thesis using 2 by force
    qed
  next
    case (3 φ1 φ2)
    then show ?case
     by (smt (verit, best) One-nat-def Pair-inject Suc-eq-numeral form.distinct(11)

        form.distinct(7) form.sel(3) form.sel(4) nat.simps(3) num-of-form.elims
numeral-3-eq-3
        numerals(2) prod-encode-eq)
  next
    case (4 x φ1)
    then show ?case
    by (smt (verit, ccfv-SIG) One-nat-def Zero-neq-Suc num-of-form.elims num-of-form.simps(4)
        numeral-3-eq-3 numerals(2) old.nat.inject old.prod.inject prod-encode-eq)
  qed
qed auto

consts term-of-num :: nat ⇒ nterm
specification (term-of-num) ‹∀ n. term-of-num n = (THE t. num-of-term t = n)›
  using num-of-term-inj by force


lemma term-of-num-of-term [simp]: ‹term-of-num(num-of-term t) = t›
  using num-of-term-inj HOL.nitpick-choice-spec by auto

consts form-of-num :: nat ⇒ form
specification (form-of-num) ‹∀ n. form-of-num n = (THE φ. num-of-form φ =
n)›
  using num-of-form-inj by force
```

lemma *form-of-num-of-form* [*simp*]: ‹*form-of-num* (*num-of-form* φ) = φ›
using *num-of-form-inj HOL.nitpick-choice-spec* **by** *auto*

end

theory *Skolem-Normal-Form*
imports
Prenex-Normal-Form
Bumping
begin

lemma *witness-imp-holds-exists*:
assumes ‹*is-interpretation* (*functions-term t*, *preds*) (*I* :: (*nat*, *nat*) *term intrp*)›
and
‹*is-valuation I* β› **and**
‹*I*, β $\models$ (φ $\cdot_{fm}$ (*subst x t*))›
shows ‹*I*, β $\models$ ($\exists$ *x*. φ)›
proof −
have ‹(λ*v*. $[\![$*subst x t v*$]\!]^{I,\beta}$) = β(*x* := $[\![t]\!]^{I,\beta}$)›
proof −
have $\forall$ *n*. $[\![$*subst x t n*$]\!]^{I,\beta}$ = (β(*x* := $[\![t]\!]^{I,\beta}$)) *n*
by (*simp add*: *subst-def*)
then show *?thesis*
by *blast*
qed
moreover have ‹$[\![t]\!]^{I,\beta}$ $\in$ *dom I*›
using *assms*(*1*)
proof (*induct t*)
case (*Var x*)
then show *?case*
using *assms*(*2*) **by** (*auto simp*: *is-valuation-def*)
next
case (*Fun f ts*)
then have ‹*u* $\in$ *set ts* $\Longrightarrow$ $[\![u]\!]^{I,\beta}$ $\in$ *dom I*› **for** *u*
by (*smt* (*verit*) *UN-I Un-iff fst-conv funas-term.simps*(*2*) *is-interpretation-def list.set-map*)
then show *?case*
using *eval.simps*(*2*) *fst-conv imageE length-map list.set-map list-all-set assms*(*2*)
unfolding *is-valuation-def*
by (*smt* (*verit*, *best*) *Fun.prems Un-insert-left funas-term.simps*(*2*) *insert-iff is-interpretation-def subsetI*)
qed
ultimately have ‹$\exists$ *a* $\in$ *dom I*. *I*, β(*x* := *a*) $\models$ φ›
using *assms*(*3*) *swap-subst-eval*[*of I* β φ *subst x t*] **by** *auto*
then show *?thesis*
using *holds-exists* **by** *blast*
qed

definition *skolem1* :: *nat* ⇒ *nat* ⇒ *form* ⇒ *form* **where**
‹*skolem1 f x φ* = *φ* ·$_{fm}$ (*subst x* (*Fun f* (*map Var* (*sorted-list-of-set* (*FV* (∃ *x*. *φ*))))))›

lemma *fvt-var-x-simp*:
‹*FVT* (*Var x* · *subst x* (*Fun f* (*map Var* (*sorted-list-of-set* (*FV φ* − {*x*}))))) = *FV φ* − {*x*}›
proof −
have *remove-list*:
‹*set* (*map Var* (*sorted-list-of-set* (*FV φ* − {*x*}))) = *Var* ‘ (*FV φ* − {*x*})›
using *set-map set-sorted-list-of-set* **using** *finite-FV* **by** *auto*
have ‹*FVT* (*Var x* · *subst x* (*Fun f* (*map Var* (*sorted-list-of-set* (*FV φ* − {*x*}))))) =
FVT (*Fun f* (*map Var* (*sorted-list-of-set* (*FV φ* − {*x*}))))›
by *simp*
also have ‹... = ⋃ (*FVT* ‘ *set* (*map Var* (*sorted-list-of-set* (*FV φ* − {*x*}))))›
using *term.set*(*4*) **by** *metis*
also have ‹... = ⋃ (*FVT* ‘ *Var* ‘ (*FV φ* − {*x*}))›
using *remove-list* **by** *auto*
also have ‹... = *FV φ* − {*x*}›
by *force*
finally show *?thesis* **.**
qed

lemma *holds-indep-intrp-if2*:
fixes *I I′* :: *'a intrp*
shows
‹⟦*I*,*β* ⊨ *φ*; *dom I* = *dom I′*; ⋀*p*. *intrp-rel I p* = *intrp-rel I′ p*;
⋀*f zs*. (*f*, *length zs*) ∈ *functions-form φ* ⟹ *intrp-fn I f zs* = *intrp-fn I′ f zs*⟧ ⟹
I′,*β* ⊨ *φ*›
using *holds-indep-intrp-if* **by** *blast*

lemma *fun-upds-prop*: ‹*length xs* = *length zs* ⟹ ∀ *z*∈*set zs*. *P z* ⟹ ∀ *v*. *P* (*g v*)
⟹ ∀ *v*. *P* ((*foldr* (λ*kv f*. *fun-upd f* (*fst kv*) (*snd kv*)) (*zip xs zs*) *g*) *v*)›
proof (*induction zs arbitrary*: *xs g*)
case *Nil*
then show *?case*
by *simp*
next
case (*Cons a zs*)
obtain *x xsp* **where** *xs-is*: ‹*xs* = *x* # *xsp*›
using *Cons*(*2*) **by** (*metis length-Suc-conv*)
with *Cons* **show** *?case*
by *auto*
qed

lemma ‹{*z*. ∃ *y*. *y* ∈ *FV* *φ* ∧ *z* ∈ *functions-term* (*Var y* · *subst x t*)} = *functions-term t* ∨
{*z*. ∃ *y*. *y* ∈ *FV* *φ* ∧ *z* ∈ *functions-term* (*Var y* · *subst x t*)} = {}›
proof −
have ‹*y* ≠ *x* ⟹ *functions-term* (*Var y* · *subst x t*) = {}› **for** *y*
by (*simp add*: *subst-def*)
moreover have ‹*y* = *x* ⟹ *functions-term* (*Var y* · *subst x t*) = *functions-term t*› **for** *y*
by *simp*
ultimately show *?thesis*
by *blast*
qed

lemma *func-form-subst*: ‹*x* ∈ *FV* *φ* ⟹ (*f*, *length ts*) ∈ *functions-form* (*φ* ·*fm* *subst x* (*Fun f ts*))›
proof (*induction φ rule*: *functions-form.induct*)
case *1*
then show *?case* **by** *simp*
next
case (*2 p ts*)
then show *?case*
by (*metis* (*no-types*, *lifting*) *UnCI eval-term.simps*(*1*) *formsubst-functions-form*
funas-term.simps(*2*) *mem-Collect-eq singletonI subst-simps*(*1*))
next
case (*3 φ ψ*)
then show *?case*
by *auto*
next
case (*4 y φ*)
then show *?case*
by (*metis* (*no-types*, *lifting*) *UnI2 Un-commute eval-term.simps*(*1*) *formsubst-functions-form*
funas-term.simps(*2*) *mem-Collect-eq singletonI subst-simps*(*1*))
qed

lemma *holds-formsubst*:
M,*β* ⊨ (*p* ·*fm* *i*) ⟷ *M*,(λ*t*. ⟦*t*⟧$^{M,\beta}$) ∘ *i* ⊨ *p*
by (*simp add*: *holds-indep-β-if swap-subst-eval*)

lemma *holds-formsubst1*:
M,*β* ⊨ (*p* ·*fm* *Var*(*x*:=*t*)) ⟷ *M*,*β*(*x* := ⟦*t*⟧$^{M,\beta}$) ⊨ *p*
by (*simp add*: *holds-indep-β-if swap-subst-eval*)

lemma *holds-formsubst2*:

$M,\beta \models (p \cdot_{fm} subst\ x\ t) \longleftrightarrow M,\beta(x := [\![t]\!]^{M,\beta}) \models p$
by (*simp add*: *holds-formsubst1 subst-def*)

lemma *size-nonzero* [*simp*]: *size fm* > *0*
by (*induction fm*) *auto*

lemma
assumes *prenex-ex-phi*: ‹*is-prenex* (∃ *x*. φ)›
and *notin-ff*: ‹¬ (*f*, *card* (*FV* (∃ *x*. φ))) ∈ *functions-form* (∃ *x*. φ)›
shows *holds-skolem1a*: *is-prenex* (*skolem1 f x* φ) (**is** *?A*)
and *holds-skolem1b*: *FV* (*skolem1 f x* φ) = *FV* (∃ *x*. φ) (**is** *?B*)
and *holds-skolem1c*:
Prenex-Normal-Form.size (*skolem1 f x* φ) < *Prenex-Normal-Form.size* (∃ *x*. φ) (**is** *?C*)
and *holds-skolem1d*: *predicates-form* (*skolem1 f x* φ) = *predicates-form* (∃ *x*. φ) (**is** *?D*)
and *holds-skolem1e*: *functions-form* (∃ *x*. φ) ⊆ *functions-form* (*skolem1 f x* φ) (**is** *?E*)
and *holds-skolem1f*: *functions-form* (*skolem1 f x* φ)
⊆ (*f*, *card* (*FV* (∃ *x*. φ))) ▷ *functions-form* (∃ *x*. φ) (**is** *?F*)
proof −
show *?A*
by (*metis form.inject*(*2*) *form.inject*(*3*) *prenex-ex-phi prenex-formsubst prenex-imp*
qfree-no-quantif skolem1-def)
show *?B*
proof
show ‹*FV* (*skolem1 f x* φ) ⊆ *FV* (∃ *x*. φ)›
proof
fix *z*
assume ‹*z* ∈ *FV* (*skolem1 f x* φ)›
then obtain *y* **where** *y-in*: ‹*y* ∈ *FV* φ› **and**
z-in: ‹*z* ∈ *FVT* (*Var y* · *subst x* (*Fun f* (*map Var* (*sorted-list-of-set* (*FV* φ − {*x*}))))›
unfolding *skolem1-def* **using** *formsubst-fv FV-exists* **by** *auto*
then have *neq-x*: ‹*y* ≠ *x* ⟹ *z* ∈ *FV* φ − {*x*}›
by (*simp add*: *subst-def*)
then show ‹*z* ∈ *FV* (∃ *x*. φ)›
using *fvt-var-x-simp z-in* **by** *force*
qed
next
show ‹*FV* (∃ *x*. φ) ⊆ *FV* (*skolem1 f x* φ)›
proof
fix *z*
assume *z-in*: ‹*z* ∈ *FV* (∃ *x*. φ)›
then have ‹*FVT* (*Var z* · *subst x* (*Fun f* (*map Var* (*sorted-list-of-set* (*FV* (∃ *x*. φ)))))) = {*z*}›
by (*simp add*: *subst-def*)

```
      then show ‹z ∈ FV (skolem1 f x φ)›
        unfolding skolem1-def using z-in formsubst-fv by auto
    qed
  qed
  show ?C
    by (simp add: size-indep-subst skolem1-def)
  show ?D
    by (simp add: formsubst-predicates skolem1-def)
  show ?E
    by (simp add: formsubst-functions-form skolem1-def)
  show ?F
  proof
    fix g
    assume g-in: ‹g ∈ functions-form (skolem1 f x φ)›
    then have ‹g ∈ functions-form φ ∪ {g. ∃ y. y ∈ FV φ ∧
      g ∈ functions-term (Var y · subst x (Fun f (map Var (sorted-list-of-set (FV
((∃ x. φ)))))))}›
      unfolding skolem1-def using formsubst-functions-form
      by blast
    moreover have ‹{g. ∃ y ∈ FV φ.
      g ∈ functions-term (Var y · subst x (Fun f (map Var (sorted-list-of-set (FV
((∃ x. φ)))))))}
                  ⊆ (f, card (FV (∃ x. φ))) ▷ functions-form φ›
    proof
      fix h
       assume ‹h ∈ {g. ∃ y ∈ FV φ. g ∈ functions-term (Var y · subst x (Fun f
(map Var
      (sorted-list-of-set (FV ((∃ x. φ)))))))}›
      then obtain y where y-in: ‹y ∈ FV φ› and h-in:
       ‹h ∈ functions-term (Var y · subst x (Fun f (map Var (sorted-list-of-set (FV
((∃ x. φ)))))))›
        by blast
      then have y-neq-x-case: ‹y ≠ x ⟹ h ∈ functions-form φ›
        by (simp add: subst-def)
     have ‹functions-term (Var x · subst x (Fun f (map Var (sorted-list-of-set (FV
((∃ x. φ)))))))  =
        functions-term (Fun f (map Var (sorted-list-of-set (FV ((∃ x. φ))))))›
        by (simp add: subst-def)
      have ‹functions-term (Fun f (map Var (sorted-list-of-set (FV ((∃ x. φ))))))
=
        {(f, card (FV (∃ x. φ)))}›
        by auto
      then have y-eq-x-case: ‹y = x ⟹ h = (f, card (FV (∃ x. φ)))›
        using y-in h-in by auto
      show ‹h ∈ (f, card (FV (∃ x. φ))) ▷ functions-form φ›
        using y-neq-x-case y-eq-x-case by blast
    qed
    ultimately show ‹g ∈ (f, card (FV (∃ x. φ))) ▷ functions-form (∃ x. φ)›
      by auto
```

qed
qed

definition *define-fn* ≡ λ*FN f n h*. λ*g zs*. *if g=f* ∧ *length zs* = *n then h zs else FN g zs*

lemma *holds-skolem1g*:
assumes *prenex-ex-phi*: ‹*is-prenex* (∃ *x*. φ)›
and *notin-ff*: ‹¬ (*f*, *card* (*FV* (∃ *x*. φ))) ∈ *functions-form* (∃ *x*. φ)›
fixes *I* :: *'a intrp*
assumes *interp-I*: *is-interpretation* (*language* {φ}) *I*
and *nempty-I*: *dom I* ≠ {}
and *valid*: ⋀β. *is-valuation I* β ⟹ *I*,β ⊨ (∃ *x*. φ)
obtains *M* **where** *dom M* = *dom I*
intrp-rel M = *intrp-rel I*
⋀*g zs*. *g* ≠ *f* ∨ *length zs* ≠ *card* (*FV* (∃ *x*. φ)) ⟹ *intrp-fn M g zs* = *intrp-fn I g zs*
is-interpretation (*language* {*skolem1 f x* φ}) *M*
⋀β. *is-valuation M* β ⟹ *M*,β ⊨ *skolem1 f x* φ
proof −
have *ex-a-mod-phi*: ∃ *a*∈*dom I*. *I*,β(*x* := *a*) ⊨ φ
if ∀ *v*. β *v* ∈ *dom I* **for** β
using *that FOL-Semantics.holds-exists is-valuation-def valid* **by** *blast*
define *intrp-f* **where** — Using *fold* instead causes complications
‹*intrp-f* ≡ λ*zs*. *foldr* (λ*kv f*. *fun-upd f* (*fst kv*) (*snd kv*))
(*zip* (*sorted-list-of-set* (*FV* (∃ *x*. φ))) *zs*) (λ*z*. *SOME c*. *c* ∈ *dom I*)›
define *thex* **where** *thex* ≡ λ*zs*. *SOME a*. *a* ∈ *dom I* ∧ (*I*, (*intrp-f zs*)(*x*:=*a*) ⊨ φ)
define *FN* **where** *FN* ≡ *define-fn* (*intrp-fn I*) *f* (*card* (*FV* (∃ *x*. φ))) *thex*

define *M* :: *'a intrp* **where** ‹*M* = *Abs-intrp* (*dom I*, *FN*, *intrp-rel I*)›
show *?thesis*
proof
show *dom-M-I-eq*: ‹*dom M* = *dom I*›
unfolding *M-def* **by** *simp*
show *intrp-rel-eq*: ‹*intrp-rel M* = *intrp-rel I*›
unfolding *M-def* **by** *simp*
show *intrp-fn-eq*: ⋀*g zs*. *g* ≠ *f* ∨ *length zs* ≠ *card* (*FV* (∃ *x*. φ)) ⟹
intrp-fn M g zs = *intrp-fn I g zs*
unfolding *M-def FN-def define-fn-def*
by *fastforce*
have *in-dom-I*: ‹*intrp-fn M f zs* ∈ *dom I*›
if *len-eq*: ‹*length zs* = *card* (*FV* φ − {*x*})› **and** *zs-in*: ‹*set zs* ⊆ *dom M*›
for *zs*
proof −
have *len-eq2*: ‹*length* (*sorted-list-of-set* (*FV* (∃ *x*. φ))) = *length zs*›
using *len-eq* **by** *simp*
have *zs-in2*: ‹∀ *z*∈*set zs*. *z* ∈ *dom I*›

```
      using dom-M-I-eq zs-in by force
    have fn-is-thex: ‹(intrp-fn M) f zs = thex zs›
      using len-eq by (auto simp: M-def FN-def define-fn-def)
    have ‹∀ v. (intrp-f zs) v ∈ dom I›
       using fun-upds-prop[OF len-eq2 zs-in2] nempty-I some-in-eq unfolding
intrp-f-def
      by (metis (mono-tags))
    then show ‹intrp-fn M f zs ∈ dom I›
      using nempty-I ex-a-mod-phi interp-I unfolding is-interpretation-def
      by (metis (mono-tags, lifting) fn-is-thex someI-ex thex-def)
  qed
  show ‹is-interpretation (language {skolem1 f x φ}) M›
    unfolding is-interpretation-def
  proof (intro strip)
    fix g l
    assume §: ‹(g, length l) ∈ fst (language {skolem1 f x φ}) ∧ set l ⊆ dom M›
    then have ‹(g, length l) ∈ fst (language {φ}) ∨ (g, length l) = (f, card (FV
φ − {x}))›
      using holds-skolem1f lang-singleton notin-ff prenex-ex-phi by auto
    with § show ‹intrp-fn M g l ∈ dom M›
    by (metis FV-exists dom-M-I-eq in-dom-I interp-I intrp-fn-eq is-interpretation-def
prod.inject)
  qed
  show M,β ⊨ skolem1 f x φ if is-valuation M β for β
  proof −
    have M,β(x:=thex (map β (sorted-list-of-set(FV(∃ x. φ))))) ⊨ φ
    proof (rule holds-indep-intrp-if2)
      have I, (intrp-f (map β (sorted-list-of-set(FV(∃ x. φ))))(x:=a) ⊨ φ ⟷
I, β(x:=a) ⊨ φ
        for a
      proof (intro holds-indep-β-if strip)
        fix v
        assume v ∈ FV φ
        then have v=x ∨ v ∈ FV (∃ x. φ)
          using FV-exists by blast
        moreover have
          foldr (λkv f. f(fst kv := snd kv)) (zip vs (map β vs)) (λz. SOME c. c ∈
dom I) w = β w
          if w ∈ set vs set vs ⊆ FV (∃ x. φ) for vs w
          using that by (induction vs) auto
        ultimately
        show ((intrp-f (map β (sorted-list-of-set (FV (∃ x. φ))))) (x := a)) v =
(β(x := a)) v
          using finite-FV intrp-f-def by auto
      qed
      then show I,β (x := thex (map β (sorted-list-of-set (FV (∃ x. φ))))) ⊨ φ
        by (metis (mono-tags, lifting) dom-M-I-eq ex-a-mod-phi is-valuation-def
that thex-def
          verit-sko-ex')
```

show *dom I = dom M*
using *dom-M-I-eq* **by** *auto*
show $\bigwedge p$. *intrp-rel I p = intrp-rel M p*
using *intrp-rel-eq* **by** *auto*
show $\bigwedge f\ zs$. $(f,\ length\ zs) \in$ *functions-form* $\varphi \Longrightarrow$ *intrp-fn I f zs = intrp-fn M f zs*
using *functions-form.simps notin-ff intrp-fn-eq*
by (*metis sup-bot.right-neutral*)
qed
moreover have *FN f* (*map* β (*sorted-list-of-set* (*FV* $\varphi - \{x\}$))) =
thex (*map* β (*sorted-list-of-set* (*FV* $\varphi - \{x\}$)))
by (*simp add*: *FN-def define-fn-def*)
ultimately show *?thesis*
by (*simp add*: *holds-formsubst2 skolem1-def M-def o-def*)
qed
qed
qed

lemma *holds-skolem1h*:
assumes *prenex-ex-phi*: ‹*is-prenex* $(\exists\, x.\ \varphi)$› **and** ‹$\neg$ $(f,\ card\ (FV\ (\exists\, x.\ \varphi))) \in$ *functions-form* $(\exists\, x.\ \varphi)$›
assumes *is-intrp*: *is-interpretation* (*language* $\{$*skolem1 f x* $\varphi\}$) *N*
and *nempty-N*: *dom N* $\neq \{\}$
and *is-val*: *is-valuation N* β
and *skol-holds*: *N*,$\beta \models$ *skolem1 f x* φ
shows *N*,$\beta \models (\exists\, x.\ \varphi)$
proof −
have ‹$\exists\, a \in$*dom N*. *N*,$\beta(x := a) \models \varphi$›
proof −
have ‹*N*,$(\lambda v.\ [\![$*subst x* (*Fun f* (*map Var* (*sorted-list-of-set* (*FV* $(\exists\, x.\ \varphi)$)))) $v]\!]^{N,\beta}) \models \varphi$›
by (*metis skol-holds skolem1-def swap-subst-eval*)
then have *holds-eval-f*: ‹*N*,$\beta(x := [\![$*Fun f* (*map Var* (*sorted-list-of-set* (*FV* $(\exists\, x.\ \varphi)$)))$]\!]^{N,\beta}) \models \varphi$›
by (*smt* (*verit*, *best*) *eval.simps*(1) *fun-upd-other fun-upd-same holds-indep-*β*-if subst-def*)
show ‹$\exists\, a \in$*dom N*. *N*,$\beta(x := a) \models \varphi$›
proof (*cases* ‹$x \in$ *FV* φ›)
case *True*
have *eval-to-intrp*: ‹$[\![$*Fun f* (*map Var* (*sorted-list-of-set* (*FV* $(\exists\, x.\ \varphi)$)))$]\!]^{N,\beta}$ =
intrp-fn N f $[[\![t]\!]^{N,\beta}.\ t \leftarrow$ *map Var* (*sorted-list-of-set* (*FV* $(\exists\, x.\ \varphi)$))]›
by *simp*

have ‹$[[\![t]\!]^{N,\beta}.\ t \leftarrow$ *map Var* (*sorted-list-of-set* (*FV* $(\exists\, x.\ \varphi)$))] =
$[\beta\ t.\ t \leftarrow$ (*sorted-list-of-set* (*FV* $(\exists\, x.\ \varphi)$))]›
by *auto*
then have *all-in-dom*: ‹*set* $[[\![t]\!]^{N,\beta}.\ t \leftarrow$ *map Var* (*sorted-list-of-set* (*FV* $(\exists\, x.\ \varphi)$))] $\subseteq$ *dom N*›

using *is-val* **by** (*auto simp*: *is-valuation-def*)
have ‹(*f*, *length* $[\![t]\!]^{N,\beta}$. *t* ← *map Var* (*sorted-list-of-set* (*FV* (∃ *x*. φ)))]) ∈
functions-form (φ ·$_{fm}$ *subst x* (*Fun f* (*map Var* (*sorted-list-of-set* (*FV* (∃ *x*. φ))))))›
using *func-form-subst*[*OF True*] *is-intrp lang-singleton*
unfolding *skolem1-def is-interpretation-def* **by** (*metis length-map*)
then have ‹$[\![$*Fun f* (*map Var* (*sorted-list-of-set* (*FV* (∃ *x*. φ))))$]\!]^{N,\beta}$ ∈ *dom N*›
using *is-intrp eval-to-intrp all-in-dom* **unfolding** *is-interpretation-def skolem1-def*
by (*metis fst-conv lang-singleton*)
then show *?thesis*
using *holds-eval-f* **by** *blast*
next
case *False*
obtain *a* **where** *a-in*: ‹*a* ∈ *dom N*›
using *nempty-N* **by** *blast*
then have ‹*N*,β(*x* := *a*) ⊨ φ›
using *holds-eval-f False* **by** (*metis fun-upd-other holds-indep-β-if*)
then show *?thesis*
using *a-in* **by** *blast*
qed
qed
then show *?thesis*
by *simp*
qed

lemma *holds-skolem1*:
assumes ‹*is-prenex* (∃ *x*. φ)› **and** ‹¬ (*f*, *card* (*FV* (∃ *x*. φ))) ∈ *functions-form* (∃ *x*. φ)›
shows ‹*is-prenex* (*skolem1 f x* φ) ∧
FV (*skolem1 f x* φ) = *FV* (∃ *x*. φ) ∧
size (*skolem1 f x* φ) < *size* (∃ *x*. φ) ∧
predicates-form (*skolem1 f x* φ) = *predicates-form* (∃ *x*. φ) ∧
functions-form (∃ *x*. φ) ⊆ *functions-form* (*skolem1 f x* φ) ∧
functions-form (*skolem1 f x* φ) ⊆ *insert* (*f*, *card* (*FV* (∃ *x*. φ))) (*functions-form* (∃ *x*. φ)) ∧
(∀ (*I* :: 'a *intrp*). *is-interpretation* (*language* {φ}) *I* ∧
¬ (*dom I* = {}) ∧
(∀ β. *is-valuation I* β ⟶ (*I*, β ⊨ (∃ *x*. φ))) ⟶
(∃ (*M* :: 'a *intrp*). *dom M* = *dom I* ∧
intrp-rel M = *intrp-rel I* ∧
(∀ *g zs*. ¬*g*=*f* ∨ ¬(*length zs* = *card* (*FV* (∃ *x*. φ))) ⟶ *intrp-fn M g zs* = *intrp-fn I g zs*) ∧
is-interpretation (*language* {*skolem1 f x* φ}) *M* ∧
(∀ β. *is-valuation M* β ⟶ (*M*, β ⊨ (*skolem1 f x* φ))))) ∧
(∀ (*N* :: 'a *intrp*). *is-interpretation* (*language* {*skolem1 f x* φ}) *N* ∧
¬ (*dom N* = {}) ⟶
(∀ β. *is-valuation N* β ∧ (*N*, β ⊨ (*skolem1 f x* φ)) ⟶ (*N*, β ⊨ (∃ *x*. φ))))

›
by (*smt* (*verit*, *ccfv-SIG*) *assms holds-skolem1a holds-skolem1b holds-skolem1c holds-skolem1d*
holds-skolem1e holds-skolem1f holds-skolem1g holds-skolem1h)

lemma *skolems-ex*: ‹∃ *skolems*. ∀ φ. *skolems* φ = (λk. *ppat* (λx ψ. ∀ x. (*skolems* ψ k))
(λx ψ. *skolems* (*skolem1* (*numpair J k*) x ψ) (*Suc k*)) (λψ. ψ) φ)›
proof (*intro size-rec strip*)
fix *skolems* :: *form* ⇒ *nat* ⇒ *form* **and** *g* φ
assume *IH*: ‹∀ *z*. *size z* < *size* φ ⟶ *skolems z* = *g z*›
show (λk.
ppat (λx ψ. ∀ x. *skolems* ψ k) (λx ψ. *skolems* (*skolem1* (*numpair J k*) x ψ) (*Suc k*))(λψ. ψ) φ) =
(λk. *ppat* (λx ψ. ∀ x. *g* ψ k) (λx ψ. *g* (*skolem1* (*numpair J k*) x ψ) (*Suc k*)) (λψ. ψ) φ)
proof (*cases* ∃ x φ'. φ = ∀ x. φ')
case *True*
then obtain x φ' **where** *phi-is*: φ = ∀ x. φ'
by *blast*
then have *smaller*: ‹*size* (φ' ·$_{fm}$ σ) < *size* φ› **for** σ
using *size-indep-subst* **by** *simp*
have *ppat-to-skol*: ‹(*ppat* (λx ψ. ∀ x. (*skolems* ψ k))
(λx ψ. *skolems* (*skolem1* (*numpair J k*) x ψ) (*Suc k*)) (λψ. ψ) φ) = (∀ x. *skolems* φ' k)› **for** k
unfolding *ppat-def* **by** (*simp add*: *phi-is*)
have *skol-to-g*: ‹(∀ x. *skolems* φ' k) = (∀ x. *g* φ' k)› **for** k
using *IH smaller* **by** (*simp add*: *phi-is*)
have *g-to-ppat*: ‹(∀ x. *g* φ' k) =
ppat (λx ψ. ∀ x. *g* ψ k) (λx ψ. *g* (*skolem1* (*numpair J k*) x ψ) (*Suc k*)) (λψ. ψ) φ› **for** k
unfolding *ppat-def* **using** *phi-is* **by** *simp*
show *?thesis*
using *ppat-to-skol skol-to-g g-to-ppat* **by** *auto*
next
case *False*
assume *falseAll*: ‹¬(∃ x φ'. φ = ∀ x. φ')›
then show *?thesis*
proof (*cases* ∃ x φ'. φ = ∃ x. φ')
case *True*
then obtain x φ' **where** *phi-is*: φ = ∃ x. φ'
by *blast*
then have *smaller*: ‹*size* (φ' ·$_{fm}$ σ) < *size* φ› **for** σ
using *size-indep-subst* **by** *simp*
have *ppat-to-skol*: ‹(*ppat* (λx ψ. ∀ x. (*skolems* ψ k))
(λx ψ. *skolems* (*skolem1* (*numpair J k*) x ψ) (*Suc k*)) (λψ. ψ) φ) =
skolems (*skolem1* (*numpair J k*) x φ') (*Suc k*)› **for** k
unfolding *ppat-def* **using** *phi-is* **by** *simp*

have *skol-to-g*: ‹*skolems* (*skolem1* (*numpair J k*) *x* φ') (*Suc k*) =
g (*skolem1* (*numpair J k*) *x* φ') (*Suc k*)› **for** *k*
using *IH smaller phi-is* **by** (*simp add*: *skolem1-def*)
have *g-to-ppat*: ‹*g* (*skolem1* (*numpair J k*) *x* φ') (*Suc k*) =
ppat ($\lambda x\ \psi.\ \forall\ x.\ g\ \psi\ k$) ($\lambda x\ \psi.\ g$ (*skolem1* (*numpair J k*) *x* ψ) (*Suc k*)) ($\lambda\psi.\ \psi$) φ› **for** *k*
unfolding *ppat-def* **using** *phi-is* **by** *simp*
show *?thesis*
using *ppat-to-skol skol-to-g g-to-ppat* **by** *simp*
next
case *False*
then show *?thesis*
using *falseAll ppat-last* **unfolding** *ppat-def* **by** *auto*
qed
qed
qed

consts *skolems* :: *nat* ⇒ *form* ⇒ *nat* ⇒ *form*
specification (*skolems*)
skolems-eq: ‹$\bigwedge J\ \psi\ k.$ *skolems J* ψ *k*
= *ppat* ($\lambda x\ \varphi'.\ \forall x.$ (*skolems J* φ' *k*)) ($\lambda x\ \varphi'.$ *skolems J* (*skolem1* (*numpair J k*) *x* φ') (*Suc k*)) ($\lambda\varphi.\ \varphi$) ψ›
using *skolems-ex* **by** *meson*

bounding the possible Skolem functions in a given formula

definition *skolems-bounded* ≡ $\lambda p\ J\ k.\ \forall l\ m.$ (*numpair J l*, *m*) ∈ *functions-form p* ⟶ *l* < *k*

lemma *skolems-bounded-mono*: ⟦*skolems-bounded p J k'*; *k'*≤*k*⟧ ⟹ *skolems-bounded p J k*
by (*meson dual-order.strict-trans1 skolems-bounded-def*)

lemma *skolems-bounded-prenex*: *skolems-bounded* φ *K k* ⟹ *skolems-bounded* (*prenex* φ) *K k*
unfolding *skolems-bounded-def*
by (*metis Pair-inject lang-singleton prenex-props*)

Basic properties proved by induction on the number of Skolemisation steps. Harrison's gigantic conjunction broken up for more manageable proofs, at the cost of some repetition

first, the simplest properties

lemma *holds-skolems-induction-A*:
assumes *size p* = *n* **and** *is-prenex p* **and** *skolems-bounded p J k*
shows *universal*(*skolems J p k*) ∧
FV(*skolems J p k*) = *FV p* ∧
predicates-form (*skolems J p k*) = *predicates-form p* ∧
functions-form p ⊆ *functions-form* (*skolems J p k*) ∧
functions-form (*skolems J p k*) ⊆ {(*numpair J l*,*m*) | *l m. k* ≤ *l*} ∪ *functions-form p*

```
  using assms
proof (induction n arbitrary: k p rule: less-induct)
  case (less n)
  show ?case
    using ‹is-prenex p›
  proof cases
    case 1
    then show ?thesis
      by (metis (no-types, lifting) ppat-last-qfree skolems-eq universal.simps UnCI
subsetI)
  next
    case (2 φ x)
   then have smaller: Prenex-Normal-Form.size φ < n and skbo: skolems-bounded
φ J k
      using less.prems by (auto simp add: skolems-bounded-def)
    have skoeq: skolems J p k = (∀ x. skolems J φ k)
      by (metis 2(1) ppat-simpA skolems-eq)
    show ?thesis
      using less.IH [OF smaller refl ‹is-prenex φ›, of k] skoeq
      by (simp add: 2 is-valuation-def lang-singleton skbo)
  next
    case (3 φ x)
    define φ' where φ' ≡ skolem1 (numpair J k) x φ
    have smaller: Prenex-Normal-Form.size φ' < n
          and pair-notin-ff: (numpair J k, card (FV (∃ x. φ))) ∉ functions-form
(∃ x. φ)
      using 3 holds-skolem1c less.prems unfolding φ'-def skolems-bounded-def by
blast+
    have pre: is-prenex φ'
      using 3(1) pair-notin-ff holds-skolem1a ‹is-prenex p› φ'-def by blast
    define φ'' where φ'' ≡ skolems J φ' (Suc k)
    have skos: skolems J (∃ x. φ) k = φ''
      by (metis φ'-def φ''-def ppat-simpB skolems-eq)
    have funsub: functions-form p ⊆ functions-form φ'
                 functions-form φ' ⊆ insert (numpair J k, card (FV (∃ x. φ)))
(functions-form p)
      using 3(1) pair-notin-ff φ'-def holds-skolem1e holds-skolem1f ‹is-prenex p›
by presburger+
    have skbo: skolems-bounded φ' J (Suc k)
      using ‹skolems-bounded p J k› unfolding skolems-bounded-def less-Suc-eq
      using funsub(2) by fastforce
    have FV: FV φ' = FV φ − {x}
      using 3(1) pair-notin-ff holds-skolem1b ‹is-prenex p› φ'-def by auto
    have preq: predicates-form φ' = predicates-form φ
      using φ'-def formsubst-predicates skolem1-def by presburger
    show ?thesis
      using less.IH [OF smaller refl pre, of Suc k] skolems-eq [of J p] FV funsub 3
      by (force simp: preq skbo ppat-simpB simp flip: φ'-def)
  qed
```

qed

the final conjunct of the HOL Light version

lemma *holds-skolems-induction-B*:
fixes $N :: 'a$ *intrp*
assumes *size* $p = n$ **and** *is-prenex* p **and** *skolems-bounded* p J k
and *is-interpretation* (*language* $\{$*skolems* J p $k\}$) N *dom* $N \neq \{\}$
and *is-valuation* N β N,$\beta \models$ *skolems* J p k
shows N,$\beta \models p$
using *assms*
proof (*induction n arbitrary*: N k p β *rule*: *less-induct*)
case (*less n*)
show *?case*
using ‹*is-prenex p*›
proof *cases*
case *1*
with *less* **show** *?thesis*
by (*metis* (*no-types*, *lifting*) *ppat-last-qfree skolems-eq*)
next
case (*2* φ x)
then have *smaller*: *Prenex-Normal-Form.size* $\varphi < n$ **and** *skbo*: *skolems-bounded* φ J k
using *less.prems* **by** (*auto simp add*: *skolems-bounded-def*)
have *skolems* J p k = ($\forall$ x**.** *skolems* J φ k)
by (*metis 2*(*1*) *ppat-simpA skolems-eq*)
then show *?thesis*
using *less.IH* [*OF smaller refl* ‹*is-prenex* φ›, *of k*] *less.prems*
by (*simp add*: *lang-singleton skbo valuation-valmod 2*)
next
case (*3* φ x)
define φ' **where** $\varphi' \equiv$ *skolem1* (*numpair* J k) x φ
have *smaller*: *Prenex-Normal-Form.size* $\varphi' < n$
and *pair-notin-ff*: (*numpair* J k, *card* (*FV* ($\exists$ x**.** φ))) $\notin$ *functions-form* ($\exists$ x**.** φ)
using *3 holds-skolem1c less.prems* **unfolding** φ'*-def skolems-bounded-def* **by** *blast*+
have *pre*: *is-prenex* φ'
using *3*(*1*) *pair-notin-ff holds-skolem1a* ‹*is-prenex p*› φ'*-def* **by** *blast*
define φ'' **where** $\varphi'' \equiv$ *skolems* J φ' (*Suc k*)
have *skos*: *skolems* J ($\exists$ x**.** φ) k = φ''
by (*metis* φ'*-def* φ''*-def ppat-simpB skolems-eq*)
have *functions-form* $\varphi' \subseteq$ *insert* (*numpair J k*, *card* (*FV* ($\exists$ x**.** φ))) (*functions-form p*)
using *3*(*1*) *pair-notin-ff* φ'*-def holds-skolem1f* ‹*is-prenex p*› **by** *presburger*
then have *skbo*: *skolems-bounded* φ' J (*Suc k*)
using ‹*skolems-bounded p J k*› **unfolding** *skolems-bounded-def less-Suc-eq* **by** *fastforce*
have *prex*: *is-prenex* ($\exists$ x**.** φ)
using *3* ‹*is-prenex p*› **by** *blast*

```
    have functions-form (skolem1 (numpair J k) x φ) ⊆ functions-form (skolems
J φ' (Suc k))
      using φ'-def holds-skolems-induction-A pre skbo by blast
    then show ?thesis
      using less.IH [OF smaller refl pre, of Suc k] less.prems skolems-eq [of J p]
      apply (simp add: skbo ppat-simpB 3)
      using holds-skolem1h [of x φ numpair J k] pair-notin-ff ‹is-prenex φ'›
      by (metis prex φ'-def holds-exists interpretation-sublanguage lang-singleton)
  qed
qed
```

the penultimate conjunct of the HOL Light version

```
lemma holds-skolems-induction-C:
  fixes M :: 'a intrp
  assumes size p = n and is-prenex p and skolems-bounded p J k
    and is-interpretation (language {p}) M dom M ≠ {} satisfies M {p}
  shows ∃ M'. dom M' = dom M ∧ intrp-rel M' = intrp-rel M ∧
                (∀ g zs. intrp-fn M' g zs ≠ intrp-fn M g zs
                      ⟶ (∃ l. k ≤ l ∧ g = numpair J l)) ∧
                is-interpretation (language {skolems J p k}) M' ∧
                satisfies M' {skolems J p k}
  using assms
proof (induction n arbitrary: M k p rule: less-induct)
  case (less n)
  show ?case
    using ‹is-prenex p›
  proof cases
    case 1
    with less show ?thesis
      by (metis (no-types, lifting) ppat-last-qfree skolems-eq)
  next
    case (2 φ x)
   then have smaller: Prenex-Normal-Form.size φ < n and skbo: skolems-bounded
φ J k
      using less.prems by (auto simp add: skolems-bounded-def)
    have skoeq: skolems J p k = (∀ x. skolems J φ k)
      by (metis 2(1) ppat-simpA skolems-eq)
    show ?thesis
      using less.IH [OF smaller refl ‹is-prenex φ›, of k M] skoeq less.prems
      apply (simp add: skbo 2 lang-singleton satisfies-def)
      by (metis fun-upd-triv is-valuation-def valuation-valmod)
  next
    case (3 φ x)
    define φ' where φ' ≡ skolem1 (numpair J k) x φ
    have smaller: Prenex-Normal-Form.size φ' < n
          and pair-notin-ff: (numpair J k, card (FV (∃ x. φ))) ∉ functions-form
(∃ x. φ)
      using 3 holds-skolem1c less.prems unfolding φ'-def skolems-bounded-def by
blast+
```

```
    have pre: is-prenex φ'
      using 3(1) pair-notin-ff holds-skolem1a ‹is-prenex p› φ'-def by blast
    define φ'' where φ'' ≡ skolems J φ' (Suc k)
    have skos: skolems J (∃ x. φ) k = φ''
      by (metis φ'-def φ''-def ppat-simpB skolems-eq)
   have functions-form φ' ⊆ insert (numpair J k, card (FV (∃ x. φ))) (functions-form
p)
      using 3(1) pair-notin-ff φ'-def holds-skolem1f ‹is-prenex p› by presburger
    then have skbo: skolems-bounded φ' J (Suc k)
      unfolding skolems-bounded-def less-Suc-eq
     by (meson insert-iff less.prems(3) old.prod.inject prod-encode-eq skolems-bounded-def
subsetD)
    have prex: is-prenex (∃ x. φ)
      using 3(1) ‹is-prenex p› by blast
    have **: ∃ M'. dom M' = dom M ∧ intrp-rel M' = intrp-rel M
          ∧ (∀ g. (∃ zs. intrp-fn M' g zs ≠ intrp-fn M g zs) ⟶ (∃ l≥k. g = numpair
J l))
           ∧ is-interpretation (language {skolems J φ' (Suc k)}) M'
           ∧ satisfies M' {skolems J φ' (Suc k)}
      if is-interpretation (language {(∃ x. φ)}) M
        and dom M ≠ {}
        and M-extend: ⋀β. is-valuation M β ⟹ (∃ a ∈ dom M. M, β(x:=a) ⊨ φ)
      for M :: 'a intrp
    proof −
      have M: is-interpretation (language {φ}) M
        using lang-singleton that(1) by auto
      with that show ?thesis
        using less.IH[OF smaller refl ‹is-prenex φ'› skbo]
        using holds-skolem1g [OF prex pair-notin-ff M] holds-exists
        by (smt (verit) φ'-def nat-le-linear not-less-eq-eq satisfies-def singleton-iff)
    qed
    show ?thesis
     using less.IH [OF smaller refl pre, of Suc k] less.prems skolems-eq [of J p] **
      by (simp add: skbo ppat-simpB 3 φ'-def satisfies-def)
  qed
qed


corollary holds-skolems-prenex-A:
  assumes is-prenex φ skolems-bounded φ K 0
  shows universal(skolems K φ 0) ∧ (FV (skolems K φ 0) = FV φ) ∧
        predicates-form (skolems K φ 0) = predicates-form φ ∧
        functions-form φ ⊆ functions-form (skolems K φ 0) ∧
           functions-form (skolems K φ 0) ⊆ {(numpair K l,m) | l m. True} ∪
(functions-form φ)
  using holds-skolems-induction-A [OF refl assms] by simp

corollary holds-skolems-prenex-B:
  assumes is-prenex φ skolems-bounded φ K 0
```

and *is-interpretation* (*language* {*skolems K φ 0*}) *M dom M* ≠ {}
and *is-valuation M β M*,*β* ⊨ *skolems K φ 0*
shows *M*,*β* ⊨ *φ*
using *holds-skolems-induction-B* [*OF refl assms*] **by** *simp*

corollary *holds-skolems-prenex-C*:
assumes *is-prenex φ skolems-bounded φ K 0*
and *is-interpretation* (*language* {*φ*}) *M dom M* ≠ {} *satisfies M* {*φ*}
shows ∃ *M′*. *dom M′* = *dom M* ∧ *intrp-rel M′* = *intrp-rel M* ∧
(∀ *g zs*. *intrp-fn M′ g zs* ≠ *intrp-fn M g zs* ⟶ (∃ *l*. *g* = *numpair K l*))
∧
is-interpretation (*language* {*skolems K φ 0*}) *M′* ∧
satisfies M′ {*skolems K φ 0*}
using *holds-skolems-induction-C* [*OF refl assms*] **by** *simp*

definition *skopre* **where**
‹*skopre k φ* = *skolems k* (*prenex φ*) *0*›

corollary *skopre-model-A*:
assumes *skolems-bounded φ K 0*
shows *universal*(*skopre K φ*) ∧ (*FV* (*skopre K φ*) = *FV φ* ∧
predicates-form (*skopre K φ*) = *predicates-form φ* ∧
functions-form φ ⊆ *functions-form* (*skopre K φ*) ∧
functions-form (*skopre K φ*) ⊆ {(*numpair K l*,*m*) | *l m*. *True*} ∪ (*functions-form φ*)
using *skolems-bounded-prenex holds-skolems-prenex-A*
by (*metis* (*no-types*, *lifting*) *Pair-inject assms lang-singleton prenex-props skopre-def*)

corollary *skopre-model-B*:
assumes *skolems-bounded φ K 0*
and *is-interpretation* (*language* {*skopre K φ*}) *M dom M* ≠ {}
and *is-valuation M β M*,*β* ⊨ *skopre K φ*
shows *M*,*β* ⊨ *φ*
using *skolems-bounded-prenex holds-skolems-prenex-B*
by (*metis assms prenex-props skopre-def*)

corollary *skopre-model-C*:
assumes *skolems-bounded φ K 0*
and *is-interpretation* (*language* {*φ*}) *M dom M* ≠ {} *satisfies M* {*φ*}
shows ∃ *M′*. *dom M′* = *dom M* ∧ *intrp-rel M′* = *intrp-rel M* ∧
(∀ *g zs*. *intrp-fn M′ g zs* ≠ *intrp-fn M g zs* ⟶ (∃ *l*. *g* = *numpair K l*)) ∧
is-interpretation (*language* {*skopre K φ*}) *M′* ∧
satisfies M′ {*skopre K φ*}
using *holds-skolems-prenex-C skopre-def*
by (*metis assms prenex-props prenex-satisfies skolems-bounded-prenex*)

definition *skolemize* **where**
‹*skolemize* φ = *skopre* (*num-of-form* (*bump-form* φ) + *1*) (*bump-form* φ)›

lemma *no-skolems-bump-nterm*:
shows *i*>*0* $\Longrightarrow$ (*numpair i l*, *m*) $\notin$ *functions-term* (*bump-nterm t*)
proof (*induction t*)
case (*Var x*)
then show *?case*
by *auto*
next
case (*Fun ff ts*) **then show** *?case*
by *induction auto*
qed

lemma *no-skolems-bump-form*: *i*>*0* $\Longrightarrow$ *skolems-bounded* (*bump-form* φ) *i 0*
by (*induction* φ) (*auto simp*: *skolems-bounded-def no-skolems-bump-nterm*)

lemma *universal-skolemize* [*iff*]: *universal* (*skolemize* φ)
and *FV-skolemize* [*simp*]: *FV* (*skolemize* φ) = *FV* (*bump-form* φ)
and *predicates-form-skolemize* [*simp*]: *predicates-form* (*skolemize* φ) = *predicates-form* (*bump-form* φ)
by (*simp-all add*: *skolemize-def no-skolems-bump-form skopre-model-A*)

lemma *functions-bump-form*: *functions-form* (*bump-form* φ) $\subseteq$ *functions-form* (*skolemize* φ)
by (*simp add*: *skolemize-def no-skolems-bump-form skopre-model-A*)

lemma *functions-skolemize*:
functions-form (*skolemize* φ) $\subseteq$ {(*numpair* (*num-of-form* (*bump-form* φ)+*1*) *l*, *m*) |*k l m*. *True*} $\cup$ *functions-form* (*bump-form* φ)
unfolding *skolemize-def*
using *no-skolems-bump-form skopre-model-A* **by** *auto*

lemma *skolemize-imp-holds-bump-form*:
assumes *is-interpretation* (*language* {*skolemize* φ}) *N dom N* $\neq$ {}
and *is-valuation N* β *N*,β $\models$ *skolemize* φ
shows *N*,β $\models$ *bump-form* φ
using *assms no-skolems-bump-form skolemize-def skopre-model-B* **by** *fastforce*

lemma *is-interpretation-skolemize*:
assumes *is-interpretation* (*language* {*bump-form* φ}) *M dom M* $\neq$ {} *satisfies M* {*bump-form* φ}
obtains *M'* **where** *dom M'* = *dom M intrp-rel M'* = *intrp-rel M*
$\bigwedge$*g zs*. *intrp-fn M' g zs* $\neq$ *intrp-fn M g zs* $\Longrightarrow$ $\exists$ *l*. *g* = *numpair* (*num-of-form* (*bump-form* φ) + *1*) *l*
is-interpretation (*language* {*skolemize* φ}) *M' satisfies M'* {*skolemize* φ}

by (*metis add-gr-0 assms no-skolems-bump-form skolemize-def skopre-model-C zero-less-one*)

lemma *functions-form-skolemize*:
assumes ‹(*f*,*m*) ∈ *functions-form* (*skolemize* φ)›
obtains *k* **where** ‹*f* = *numpair* 0 *k*› ‹(*k*,*m*) ∈ *functions-form* φ› | *l* **where** ‹*f* = *numpair* (*num-of-form* (*bump-form* φ) + 1) *l*›
using *functions-skolemize assms functions-form-bumpform* **by** (*fastforce dest*: *that*)

definition *skomod1* **where**
‹*skomod1* φ *M* ≡
if satisfies M {φ}
then (*SOME M'*. *dom M'* = *dom* (*bump-intrp M*) ∧
intrp-rel M' = *intrp-rel* (*bump-intrp M*) ∧
(∀ *g zs*. *intrp-fn M' g zs* ≠ *intrp-fn* (*bump-intrp M*) *g zs* ⟶
(∃ *l*. *g* = *numpair* (*num-of-form* (*bump-form* φ) + 1) *l*)) ∧
is-interpretation (*language* {*skolemize* φ}) *M'* ∧ *satisfies M'* {*skolemize* φ})
else (*Abs-intrp* (*dom M*, (λ*g zs*. (*SOME a*. *a* ∈ *dom M*)), *intrp-rel M*))›

lemma *skomod1-works*:
assumes *M*: ‹*is-interpretation* (*language* {φ}) *M*› ‹*dom M* ≠ {}›
shows ‹*dom* (*skomod1* φ *M*) = *dom* (*bump-intrp M*) ∧
intrp-rel (*skomod1* φ *M*) = *intrp-rel* (*bump-intrp M*) ∧
is-interpretation (*language* {*skolemize* φ}) (*skomod1* φ *M*) ∧
(*satisfies M* {φ} ⟶
(∀ *g zs*. *intrp-fn* (*skomod1* φ *M*) *g zs* ≠ *intrp-fn* (*bump-intrp M*) *g zs* ⟶
(∃ *l*. *g* = *numpair* (*num-of-form* (*bump-form* φ) + 1) *l*)) ∧
satisfies (*skomod1* φ *M*) {*skolemize* φ})›
proof (*cases* ‹*satisfies M* {φ}›)
case *True*
obtain *M'* **where**
dom M' = *dom* (*bump-intrp M*) *intrp-rel M'* = *intrp-rel* (*bump-intrp M*)
⋀*g zs*. *intrp-fn M' g zs* ≠ *intrp-fn* (*bump-intrp M*) *g zs* ⟹ ∃ *l*. *g* = *numpair* (*num-of-form* (*bump-form* φ) + 1) *l*
is-interpretation (*language* {*skolemize* φ}) *M' satisfies M'* {*skolemize* φ}
proof (*rule is-interpretation-skolemize*)
show *is-interpretation* (*language* {*bump-form* φ}) (*bump-intrp M*)
by (*simp add*: *assms*(1) *bumpform-interpretation*)
next
show *dom* (*bump-intrp M*) ≠ {}
by (*simp add*: *assms*(2))
next
show *satisfies* (*bump-intrp M*) {*bump-form* φ}

```
    by (metis True bump-dom bumpform is-valuation-def satisfies-def singleton-iff)
  qed metis
  then show ?thesis
    apply (simp only: skomod1-def True)
    by (smt (verit, del-insts) someI)
next
  case False
  then show ?thesis
   by (simp add: skomod1-def assms bump-intrp-def intrp-is-struct is-interpretation-def
some-in-eq)
qed

definition skomod-FN ≡ λM g zs. if numfst g = 0 then intrp-fn M (numsnd g) zs
                                else intrp-fn (skomod1 (unbump-form (form-of-num
(numfst g − 1))) M) g zs

definition skomod where
  ‹skomod M ≡ Abs-intrp (dom M, skomod-FN M, intrp-rel M)›


lemma skomod-interpretation:
  assumes ‹is-interpretation (language {φ}) M›  ‹dom M ≠ {}›
  shows ‹is-interpretation (language {skolemize φ}) (skomod M)›
proof −
  have stM: struct (dom M)
    by (simp add: intrp-is-struct)
  have indom: intrp-fn M f l ∈ dom M
    if (f, length l) ∈ functions-form φ and set l ⊆ dom M for f l
    using assms that by (auto simp: is-interpretation-def lang-singleton)
  show ?thesis
  proof −
    have intrp-fn (skomod M) f l ∈ dom (skomod M)
      if fl: (f, length l) ∈ functions-form (skolemize φ) and set l ⊆ dom (skomod
M) for f l
    proof −
      consider (0) k where ‹f = numpair 0 k› ‹(k, length l) ∈ functions-form φ›
            | (1) l where ‹f = numpair (num-of-form (bump-form φ) + 1) l›
        using functions-form-skolemize [OF fl] by metis
      then show ?thesis
      proof cases
        case 0
        with that show ?thesis
          by (simp add: stM indom skomod-def skomod-FN-def)
      next
        case (1 l')
        then show ?thesis
          using that skomod1-works [OF assms]
          by (force simp add: stM indom skomod-def skomod-FN-def lang-singleton
is-interpretation-def)
```

```
    qed
  qed
  then show ?thesis
    by (auto simp: lang-singleton is-interpretation-def)
 qed
qed



proposition skomod-works:
 assumes ‹is-interpretation (language {φ}) M›  ‹dom M ≠ {}›
 shows  ‹satisfies M {φ} ⟷ satisfies (skomod M) {skolemize φ}›
proof
 assume φ: satisfies M {φ}
 have Abs-intrp (dom M, skomod-FN M, intrp-rel M), β ⊨ skolemize φ
  if is-valuation (Abs-intrp (dom M, skomod-FN M, intrp-rel M)) β
  for β :: nat ⇒ 'a
 proof −
  have is-valuation (skomod1 φ M) β
    by (metis assms bump-dom dom-Abs-is-fst is-valuation-def skomod1-works
struct.intro that)
  then have skomod1 φ M,β ⊨ skolemize φ
    by (meson φ assms insertCI satisfies-def skomod1-works)
  then show ?thesis
  proof (rule holds-indep-intrp-if2)
    fix f and zs :: 'a list
    assume f: (f, length zs) ∈ functions-form (skolemize φ)
   show intrp-fn (skomod1 φ M) f zs = intrp-fn (Abs-intrp (dom M, skomod-FN
M, intrp-rel M)) f zs
      using functions-form-skolemize [OF f]
    proof cases
      case (1 k)
      with φ skomod1-works [OF assms] show ?thesis
        apply (simp add: skomod-FN-def bump-intrp-def)
        by (metis Zero-neq-Suc prod.inject prod-encode-inverse sndI)
    qed (simp add: skomod-FN-def)
  qed (simp-all add: assms skomod1-works)
 qed
 then show satisfies (skomod M) {skolemize φ}
  by (simp add: satisfies-def skomod-def skomod-FN-def)
next
 assume φ: satisfies (skomod M) {skolemize φ}
 have bump-intrp M,β ⊨ bump-form φ if is-valuation (bump-intrp M) β for β ::
nat ⇒ 'a
 proof −
  have skomod M,β ⊨ skolemize φ
    using φ that by (simp add: satisfies-def is-valuation-def skomod-def)
  with assms that skomod-interpretation [OF assms] skolemize-imp-holds-bump-form
  have skomod M,β ⊨ bump-form φ
```

by (*simp add*: *is-valuation-def skolemize-imp-holds-bump-form skomod-def*)
then show *?thesis*
proof (*rule holds-indep-intrp-if2*)
fix *f* **and** *zs* :: *'a list*
assume *f*: (*f*, *length zs*) ∈ *functions-form* (*bump-form* φ)
show *intrp-fn* (*skomod M*) *f zs* = *intrp-fn* (*bump-intrp M*) *f zs*
using *functions-form-bumpform* [*OF f*]
by (*auto simp*: *skomod-FN-def skomod-def*)
qed (*simp-all add*: *skomod-def*)
qed
then have *satisfies* (*bump-intrp M*) {*bump-form* φ}
by (*auto simp*: *satisfies-def*)
then show *satisfies M* {φ}
by (*metis bump-dom bumpform is-valuation-def satisfies-def singleton-iff*)
qed

proposition *skolemize-satisfiable*:
‹(∃ *M*::*'a intrp*. *dom M* ≠ {} ∧ *is-interpretation* (*language S*) *M* ∧
satisfies M S) ⟷
(∃ *M*::*'a intrp*. *dom M* ≠ {} ∧ *is-interpretation* (*language* (*skolemize* ‘ *S*)) *M*
∧ *satisfies M* (*skolemize* ‘ *S*))› (**is** *?lhs* = *?rhs*)
proof
assume *?lhs*
then obtain *M*::*'a intrp* **where** *dom M* ≠ {}
and *int*: *is-interpretation* (*language S*) *M* **and** *sat*: *satisfies M S*
by *auto*
show *?rhs*
proof (*intro exI conjI*)
show *dom* (*skomod M*) ≠ {}
using ‹*dom M* ≠ {}› **by** (*simp add*: *dom-def skomod-def struct-def*)
have *intrp-fn* (*skomod M*) *f l* ∈ *dom* (*skomod M*)
if *l*: *set l* ⊆ *dom* (*skomod M*)
and φ ∈ *S*
and *f*: (*f*, *length l*) ∈ *functions-form* (*skolemize* φ)
for *f l* φ
proof −
have *is-interpretation* (*language* {φ}) *M*
using ‹φ ∈ *S*› **unfolding** *lang-singleton*
by (*metis Sup-upper functions-forms-def image-iff int interpretation-sublanguage language-def*)
then have *is-interpretation* (*language* {*skolemize* φ}) (*skomod M*)
by (*intro skomod-interpretation* ‹*dom M* ≠ {}›)
then show *?thesis*
by (*simp add*: *f is-interpretation-def l lang-singleton*)
qed
then show *is-interpretation* (*language* (*skolemize* ‘ *S*)) (*skomod M*)
by (*auto simp add*: *is-interpretation-def language-def functions-forms-def*)

```
  next
    have skomod M,β ⊨ skolemize φ
      if is-valuation (skomod M) β and φ ∈ S for β φ
    proof −
      have is-interpretation (language {φ}) M
        using ‹φ ∈ S› unfolding lang-singleton
      by (metis Sup-upper functions-forms-def image-iff int interpretation-sublanguage
language-def)
      then show ?thesis
        using that sat ‹dom M ≠ {}›
        by (metis satisfies-def singleton-iff skomod-works)
    qed
    then show satisfies (skomod M) (skolemize ‘ S)
      by (auto simp add: satisfies-def image-iff)
  qed
next
  assume ?rhs
  then obtain M :: 'a intrp where dom M ≠ {}
    and int: is-interpretation (language (skolemize ‘ S)) M
    and sat: satisfies M (skolemize ‘ S)
    by auto
  show ?lhs
  proof (intro exI conjI)
    show dom (unbump-intrp M) ≠ {}
      using ‹dom M ≠ {}› struct-def by blast
  next
    have functions-forms (bump-form ‘ S) ⊆ functions-forms (skolemize ‘ S)
      using functions-bump-form functions-forms-def by auto
    then have ∗: is-interpretation (language (bump-form ‘ S)) M
      by (metis int interpretation-sublanguage language-def)
    have is-interpretation (language {φ}) (unbump-intrp M)
      if is-interpretation (language {bump-form φ}) M for φ
      using that
    proof (induction φ)
      case (Atom p ts)
      have ∗∗: (f,k) ∈ Union (set (map functions-term l))
          ⟹ (numpair 0 f, k) ∈ Union (set (map functions-term (map bump-nterm
l))) for l k f
      proof (induction l)
        case Nil
        then show ?case by simp
      next
        case (Cons a l)
          have (f,k) ∈ functions-term t ⟹ (numpair 0 f, k) ∈ functions-term
(bump-nterm t) for t
          by (induction t) auto
        with Cons show ?case
          by auto
      qed
```

```
    show ?case
      using Atom ** by (auto simp: is-interpretation-def lang-singleton un-
bump-intrp-def)
  qed (auto simp: is-interpretation-def lang-singleton)
  with * show is-interpretation (language S) (unbump-intrp M)
    unfolding is-interpretation-def lang-singleton
    by (simp add: language-def functions-forms-def) blast
 next
  have unbump-intrp M,β ⊨ φ
    if is-valuation (unbump-intrp M) β and φ ∈ S for β φ
  proof −
    have is-interpretation (language {skolemize φ}) M
      using ‹φ ∈ S› unfolding lang-singleton
    by (metis Sup-upper functions-forms-def image-eqI int interpretation-sublanguage
language-def)
    then show ?thesis
      using that ‹dom M ≠ {}› sat unfolding satisfies-def
      by (metis dom-Abs-is-fst image-eqI intrp-is-struct is-valuation-def
          skolemize-imp-holds-bump-form unbump-holds unbump-intrp-def)
  qed
  then show satisfies (unbump-intrp M) S
    by (auto simp add: satisfies-def image-iff)
 qed
qed


fun specialize :: form ⇒ form where
  ‹specialize ⊥ = ⊥›
| ‹specialize (Atom p ts) = Atom p ts›
| ‹specialize (φ ⟶ ψ) = φ ⟶ ψ›
| ‹specialize (∀ x. φ) = specialize φ›


lemma specialize-satisfies:
  fixes M :: 'a intrp
  assumes ‹dom M ≠ {}›
  shows ‹satisfies M (specialize ` S) ⟷ satisfies M S›
proof −
  have satisfies M {specialize φ} ⟷ satisfies M {φ} for φ
  proof (induction φ)
    case (Forall x1 φ)
    show ?case
    proof
      show satisfies M {specialize (∀ x1. φ)} ⟹ satisfies M {∀ x1. φ}
        using Forall by (auto simp: satisfies-def valuation-valmod)
      show satisfies M {specialize (∀ x1. φ)} if §: satisfies M {∀ x1. φ}
      proof −
        have M,β ⊨ specialize φ if is-valuation M β for β
          using that § Forall unfolding is-valuation-def satisfies-def singleton-iff
```

```
        by (metis fun-upd-triv holds.simps(4))
      then show ?thesis
        by (auto simp: satisfies-def)
    qed
  qed
 qed auto
 then show ?thesis
  by (auto simp add: satisfies-def)
qed


lemma specialize-qfree: ‹universal φ ⟹ qfree (specialize φ)›
 by (induction rule: universal.induct) (auto elim: qfree.elims)

lemma functions-form-specialize [simp]: functions-form(specialize φ) = functions-form
φ
 by (induction φ) auto

lemma predicates-form-form-specialize [simp]: predicates-form(specialize φ) = pred-
icates-form φ
 by (induction φ) auto


lemma specialize-language: ‹language (specialize ` S) = language S›
 by (simp add: language-def functions-forms-def predicates-def)

definition skolem :: form ⇒ form where
 ‹skolem φ = specialize(skolemize φ)›

lemma skolem-qfree: ‹qfree (skolem φ)›
 by (simp add: skolem-def specialize-qfree)

theorem skolem-satisfiable:
    ‹(∃ M::'a intrp. dom M ≠ {} ∧ is-interpretation (language S) M ∧ satisfies M
S)
 ⟷ (∃ M::'a intrp. dom M ≠ {} ∧ is-interpretation (language (skolem ` S)) M
∧
     satisfies M (skolem ` S))›
proof −
 have is-interpretation (language (skolemize ` S)) M ⟷ is-interpretation (language
(skolem ` S)) M
   for M :: 'a intrp
  by (simp add: functions-forms-def is-interpretation-def language-def skolem-def)
 moreover
 have satisfies M (skolemize ` S) ⟷ satisfies M (skolem ` S)
   if dom M ≠ {} for M :: 'a intrp
    by (smt (verit, del-insts) image-iff satisfies-def skolem-def specialize-satisfies
that)
 ultimately show ?thesis
```

by (*metis skolemize-satisfiable*)
qed

end

theory *Canonical-Models*
imports *Skolem-Normal-Form*
begin

inductive-set *terms-set* :: ‹(*nat* × *nat*) *set* ⇒ *nterm set›* **for** *fns* :: ‹(*nat* × *nat*) *set›* **where**
vars: ‹(*Var v*) ∈ *terms-set fns›*
| *fn*: ‹(*Fun f ts*) ∈ *terms-set fns›*
if ‹(*f*, *length ts*) ∈ *fns›* ‹⋀*t*. *t* ∈ *set ts* ⟹ *t* ∈ *terms-set fns›*

lemma *struct-terms-set* [*iff*]: ‹*struct* (*terms-set A*)›
by (*metis empty-iff struct.intro terms-set.vars*)

lemma *stupid-canondom*: ‹*t* ∈ *terms-set* (*fst* ℒ) ⟹ *functions-term t* ⊆ (*fst* ℒ)›
by (*induction t rule*: *terms-set.induct*) *auto*

lemma *finite-subset-instance*: ‹*finite t′* ⟹ *t′* ⊆ {φ ·$_{fm}$ σ |σ φ. *P* σ ∧ φ ∈ *s*} ⟹
(∃*t*. *finite t* ∧ *t* ⊆ *s* ∧ *t′* ⊆ {φ ·$_{fm}$ σ |σ φ. *P* σ ∧ φ ∈ *t*})›
proof (*induction t′ rule*: *finite.induct*)
case *emptyI*
then show *?case* **by** *blast*
next
case (*insertI A a*)
obtain φ*a* **where** *phi-in*: ‹φ*a* ∈ *s›* **and** *phi-ex-subs*: ‹∃σ. *P* σ ∧ *a* = φ*a* ·$_{fm}$ σ›
using *insertI*(*3*) **by** *auto*
obtain Φ **where** *Phi-subs*: ‹Φ ⊆ *s›* **and** ‹*finite* Φ› **and** *Phi-set*: ‹*A* ⊆ {φ ·$_{fm}$ σ
|σ φ. *P* σ ∧ φ ∈ Φ}›
using *insertI*(*3*) *insertI*(*2*) **by** *auto*
then have ‹*finite* (φ*a* ▷ Φ)›
by *auto*
moreover have ‹(φ*a* ▷ Φ) ⊆ *s*›
using *phi-in Phi-subs* **by** *auto*
moreover have ‹*a* ▷ *A* ⊆ {φ ·$_{fm}$ σ |σ φ. *P* σ ∧ φ ∈ (φ*a* ▷ Φ)}›
using *phi-ex-subs Phi-set* **by** *blast*
ultimately show *?case* **by** *blast*
qed

lemma *finite-subset-skolem*: ‹*finite u* ⟹ *u* ⊆ {*skolem* φ |φ. φ ∈ *s*} ⟹
(∃*t*. *finite t* ∧ *t* ⊆ *s* ∧ *u* = {*skolem* φ |φ. φ ∈ *t*})›

proof (*induction u rule*: *finite.induct*)
case *emptyI*
then show *?case* **by** *auto*
next
case (*insertI A a*)
obtain φa **where** *phi-in*: ‹$\varphi a \in s$› **and** *phi-ex-subs*: ‹a = *skolem* φa›
using *insertI*(*3*) **by** *auto*
obtain Φ **where** *Phi-subs*: ‹$\Phi \subseteq s$› **and** ‹*finite* Φ› **and** *Phi-set*: ‹$A = \{$*skolem* φ $|\varphi.\ \varphi \in \Phi\}$›
using *insertI*(*3*) *insertI*(*2*) **by** *auto*
then have ‹*finite* $(\varphi a \triangleright \Phi)$›
by *auto*
moreover have ‹$(\varphi a \triangleright \Phi) \subseteq s$›
using *phi-in Phi-subs* **by** *auto*
moreover have ‹$a \triangleright A = \{$*skolem* φ $|\varphi.\ \varphi \in (\varphi a \triangleright \Phi)\}$›
using *phi-ex-subs Phi-set* **by** *blast*
ultimately show *?case*
by *blast*
qed

lemma *valuation-exists*: ‹$\neg$ (*dom* $M = \{\}$) $\Longrightarrow \exists \beta.$ *is-valuation* M β›
unfolding *dom-def is-valuation-def* **by** *fast*

lemma *holds-itlist-exists*:
‹(M,β $\models$ (*foldr* (λx *p*. $\exists$ *x*. *p*) *xs* φ)) $\longleftrightarrow$
($\exists$ *as*. *length as* = *length xs* $\wedge$ *set as* $\subseteq$ *dom* M $\wedge$
(M,(*foldr* (λu β. β(*fst u* := *snd u*)) (*rev* (*zip xs as*)) β) $\models$ φ))›
proof (*induction xs arbitrary*: β φ)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons x xs*)
show *?case*
by (*force simp add*: *Cons length-Suc-conv simp flip*: *fun-upd-def*)
qed

definition *canonical* :: ‹(*nat* × *nat*) *set* × (*nat* × *nat*) *set* ⇒ *nterm intrp* ⇒ *bool*›
where
‹*canonical* $\mathcal{L}$ $\mathcal{M}$ ≡ (*dom* $\mathcal{M}$ = *terms-set* (*fst* $\mathcal{L}$) ∧ (∀ *f*. *intrp-fn* $\mathcal{M}$ *f* = *Fun f*))›

definition *pintrp-of-intrp* :: ‹'*m intrp* ⇒ (*nat* ⇒ '*m*) ⇒ (*form* ⇒ *bool*)› **where**
‹*pintrp-of-intrp* $\mathcal{M}$ β = ($\lambda\varphi$. $\mathcal{M}$,β $\models$ φ)›

definition

canon-of-prop :: ‹((*nat* × *nat*) *set* × (*nat* × *nat*) *set*) ⇒ (*form* ⇒ *bool*) ⇒ *nterm intrp*› **where**
‹*canon-of-prop* $\mathcal{L}$ *I* ≡ *Abs-intrp* (*terms-set* (*fst* $\mathcal{L}$), *Fun*, (λ*p*. {*ts*. *I* (*Atom p ts*)}))›

lemma *dom-canon-of-prop* [*simp*]: ‹*dom* (*canon-of-prop* $\mathcal{L}$ *I*) = *terms-set* (*fst* $\mathcal{L}$)›
by (*simp add*: *canon-of-prop-def*)

lemma *intrp-fn-canon-of-prop* [*simp*]: ‹*intrp-fn* (*canon-of-prop* $\mathcal{L}$ *I*) = *Fun*›
by (*simp add*: *canon-of-prop-def*)

lemma *intrp-rel-canon-of-prop* [*simp*]: ‹*intrp-rel* (*canon-of-prop* $\mathcal{L}$ *I*) = (λ*p*. {*ts*. *I* (*Atom p ts*)})›
by (*simp add*: *canon-of-prop-def*)

lemma *pholds-pintrp-of-intrp*:
‹*qfree* φ ⟹ (*pintrp-of-intrp* $\mathcal{M}$ β) ⊨$_p$ φ ⟷ $\mathcal{M}$,β ⊨ φ›
unfolding *pintrp-of-intrp-def* **by** (*induction* φ) *simp*+

lemma *intrp-of-canon-of-prop* [*simp*]:
‹*pintrp-of-intrp* (*canon-of-prop* $\mathcal{L}$ *I*) *Var* (*Atom p ts*) = *I* (*Atom p ts*)›
proof −
have §: ‹*terms-set* (*fst* $\mathcal{L}$) ≠ {}›
using *terms-set.vars* **by** *auto*
have ‹∀ *t* ∈ *set ts*. ⟦*t*⟧$^{\textit{Abs-intrp}\ (\textit{terms-set}\ (\textit{fst}\ \mathcal{L}),\ \textit{Fun},\ \lambda p.\ \{ts.\ I\ (\textit{form.Atom}\ p\ ts)\}),\textit{Var}}$ = *t*›
proof (*induction ts*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons t ts*)
have ‹⟦*t*⟧$^{\textit{Abs-intrp}\ (\textit{terms-set}\ (\textit{fst}\ \mathcal{L}),\ \textit{Fun},\ \lambda p.\ \{ts.\ I\ (\textit{form.Atom}\ p\ ts)\}),\textit{Var}}$ = *t*›
proof (*induction t*)
case (*Var x*)
then show *?case*
by *simp*
next
case *Fun*
then show *?case*
by (*simp add*: § *struct-def map-idI*)
qed
with *Cons* **show** *?case*
by *simp*
qed
then show *?thesis*
by (*simp add*: § *struct-def canon-of-prop-def pintrp-of-intrp-def holds-def map-idI*)
qed

lemma *holds-canon-of-prop*:
assumes ‹*qfree* φ› **shows** ‹(*canon-of-prop* $\mathcal{L}$ *I*),*Var* $\models \varphi \longleftrightarrow I \models_p \varphi$›
proof −
have ‹*pintrp-of-intrp* (*canon-of-prop* $\mathcal{L}$ *I*) *Var* $\models_p \varphi \longleftrightarrow I \models_p \varphi$›
using *assms*
by (*induction* φ) *auto*
with *assms* **show** *?thesis*
using *pholds-pintrp-of-intrp* **by** *blast*
qed

lemma *holds-canon-of-prop-general*:
assumes ‹*qfree* φ› **shows** ‹(*canon-of-prop* $\mathcal{L}$ *I*),$\beta \models \varphi \longleftrightarrow I \models_p (\varphi \cdot_{fm} \beta)$›
proof −
have ‹*pintrp-of-intrp* (*canon-of-prop* $\mathcal{L}$ *I*) $\beta \models_p \varphi \longleftrightarrow I \models_p (\varphi \cdot_{fm} \beta)$›
using *assms*
proof (*induction* φ)
case *Atom*
have ‹$[\![t]\!]^{Abs\text{-}intrp\ (terms\text{-}set\ (fst\ \mathcal{L}),\ Fun,\ \lambda p.\ \{ts.\ I\ (form.Atom\ p\ ts)\}),\beta} = t \cdot \beta$›
for *t*
using *term-subst-eval* **by** (*metis empty-iff intrp-fn-Abs-is-fst-snd struct-def terms-set.simps*)
moreover have ‹*struct* (*terms-set* (*fst* $\mathcal{L}$))›
by (*metis empty-iff struct.intro terms-set.vars*)
ultimately show *?case*
by (*simp add*: *canon-of-prop-def pintrp-of-intrp-def Atom*)
qed *auto*
with *assms* **show** *?thesis*
by (*simp add*: *pholds-pintrp-of-intrp*)
qed

lemma *canonical-canon-of-prop*: ‹*canonical* $\mathcal{L}$ (*canon-of-prop* $\mathcal{L}$ *I*)›
unfolding *canonical-def canon-of-prop-def*
by (*metis dom-Abs-is-fst emptyE intrp-fn-Abs-is-fst-snd struct-def terms-set.vars*)

lemma *interpretation-canon-of-prop*: ‹*is-interpretation* $\mathcal{L}$ (*canon-of-prop* $\mathcal{L}$ *I*)›
unfolding *is-interpretation-def canon-of-prop-def*
by (*metis* (*no-types*, *lifting*) *canonical-canon-of-prop canonical-def dom-Abs-is-fst intrp-fn-Abs-is-fst-snd intrp-is-struct subset-code*(*1*) *terms-set.fn*)

lemma *prop-valid-imp-fol-valid*: ‹*qfree* $\varphi \wedge (\forall I.\ I \models_p \varphi) \Longrightarrow (\forall \mathcal{M}\ \beta.\ \mathcal{M},\beta \models \varphi)$›
using *pholds-pintrp-of-intrp* **by** *fast*

lemma *fol-valid-imp-prop-valid*: ‹*qfree* $\varphi \wedge (\forall \mathcal{M}\ \beta.\ canonical\ (language\ \{\varphi\})\ \mathcal{M} \longrightarrow \mathcal{M},\beta \models \varphi)$
$\Longrightarrow \forall I.\ I \models_p \varphi$›
using *canonical-canon-of-prop holds-canon-of-prop* **by** *blast*

lemma *satisfies-psatisfies*: ‹$[\![\varphi \in \Phi;\ \Phi \subseteq \{\varphi.\ qfree\ \varphi\};\ satisfies\ \mathcal{M}\ \Phi;\ is\text{-}valuation\ \mathcal{M}\ \beta]\!] \Longrightarrow$
psatisfies (*pintrp-of-intrp* $\mathcal{M}\ \beta$) Φ›
using *pholds-pintrp-of-intrp satisfies-def* **by** *blast*

lemma *psatisfies-instances*:
assumes *qf*: ‹$\Phi \subseteq \{\varphi.\ qfree\ \varphi\}$›
and *ps*: ‹*psatisfies* I $\{\varphi \cdot_{fm} \beta\ |\varphi\ \beta.\ (\forall x.\ \beta\ x \in terms\text{-}set\ (fst\ \mathcal{L})) \wedge \varphi \in \Phi\}$›
shows ‹*satisfies* (*canon-of-prop* $\mathcal{L}\ I$) Φ›
unfolding *satisfies-def is-valuation-def*
by (*smt* (*verit*, *best*) *dom-canon-of-prop holds-canon-of-prop-general mem-Collect-eq ps qf*
subset-iff)

lemma *satisfies-instances*:
assumes ‹*is-interpretation* (*language* Ξ) $\mathcal{M}$›
shows ‹*satisfies* $\mathcal{M}$ $\{\varphi \cdot_{fm} \sigma\ |\varphi\ \sigma.\ \varphi \in \Phi \wedge (\forall x.\ \sigma\ x \in terms\text{-}set\ (fst\ (language\ \Xi)))\} \longleftrightarrow$
satisfies $\mathcal{M}\ \Phi$›
unfolding *satisfies-def mem-Collect-eq*
proof (*intro iffI strip*)
fix $\beta\ \varphi$
assume $\mathcal{M}$: ‹$\forall \beta\ \varphi.$ *is-valuation* $\mathcal{M}\ \beta \longrightarrow \varphi \in \Phi \longrightarrow \mathcal{M},\beta \models \varphi$› ‹*is-valuation* $\mathcal{M}\ \beta$›
and ‹$\exists \varphi'\ \sigma.\ \varphi = \varphi' \cdot_{fm} \sigma \wedge \varphi' \in \Phi \wedge (\forall x.\ \sigma\ x \in terms\text{-}set\ (fst\ (language\ \Xi)))$›
then obtain $\varphi'\ \sigma$ **where** σ: ‹$\varphi = \varphi' \cdot_{fm} \sigma$› ‹$\varphi' \in \Phi$› ‹$\forall x.\ \sigma\ x \in terms\text{-}set\ (functions\text{-}forms\ \Xi)$›
by (*auto simp add*: *language-def*)
with $\mathcal{M}$ *assms* **have** ‹$\mathcal{M}$,$(\lambda v.\ [\![\sigma\ v]\!]^{\mathcal{M},\beta}) \models \varphi'$›
by (*metis* (*no-types*, *lifting*) *eq-fst-iff interpretation-eval interpretation-sublanguage*
is-valuation-def language-def stupid-canondom)
with *assms* **show** ‹$\mathcal{M}$,$\beta \models \varphi$›
by (*simp add*: σ *swap-subst-eval*)
qed (*metis subst-var terms-set.vars*)

lemma *compact-canon-qfree*:

assumes *qf*: ‹$\Phi \subseteq \{\varphi.\ qfree\ \varphi\}$›
and *int*: ‹$\bigwedge\Psi.\ [\![finite\ \Psi;\ \Psi \subseteq \Phi]\!]$
$\Longrightarrow \exists \mathcal{M}::'a\ intrp.\ is\text{-}interpretation\ (language\ \Xi)\ \mathcal{M} \wedge dom\ \mathcal{M} \neq \{\} \wedge$
satisfies $\mathcal{M}\ \Psi$›
obtains $\mathcal{C}$ **where** ‹*is-interpretation* (*language* Ξ) $\mathcal{C}$› ‹*canonical* (*language* Ξ) $\mathcal{C}$›
‹*satisfies* $\mathcal{C}\ \Phi$›
proof −
define Γ **where** ‹$\Gamma \equiv \lambda X.\ \{\varphi \cdot_{fm} \beta\ |\beta\ \varphi.\ (\forall x.\ \beta\ x \in terms\text{-}set\ (fst\ (language$
$\Xi))) \wedge \varphi \in X\}$›
have ‹*psatisfiable* ($\Gamma\ \Phi$)›
unfolding *psatisfiable-def*
proof (*rule compact-prop*)
fix Θ
assume ‹*finite* Θ› ‹$\Theta \subseteq \Gamma\ \Phi$›
then have ‹$\exists \Psi.$ *finite* $\Psi \wedge \Psi \subseteq \Phi \wedge \Theta \subseteq \Gamma\ \Psi$›
using *finite-subset-instance* Γ*-def* **by** *force*
then obtain Ψ **where** Ψ: ‹*finite* Ψ› ‹$\Psi \subseteq \Phi$› ‹$\Theta \subseteq \Gamma\ \Psi$›
by *auto*
have ‹*psatisfiable* Θ›
proof (*rule psatisfiable-mono*)
obtain $\mathcal{M}$::‹$'a$ *intrp*› **where** $\mathcal{M}$: ‹*is-interpretation* (*language* Ξ) $\mathcal{M}$› ‹*dom* $\mathcal{M}$
$\neq \{\}$›
‹*satisfies* $\mathcal{M}\ \Psi$›
using *int* Ψ **by** *meson*
then obtain β **where** β: ‹*is-valuation* $\mathcal{M}\ \beta$›
by (*meson valuation-exists*)
moreover have ‹$\Gamma\ \Psi \subseteq \{\varphi.\ qfree\ \varphi\}$›
using Γ*-def* Ψ *qf qfree-formsubst* **by** *auto*
moreover have ‹*satisfies* $\mathcal{M}$ ($\Gamma\ \Psi$)›
using $\mathcal{M}$ **unfolding** Γ*-def*
by (*smt* (*verit*, *del-insts*) *mem-Collect-eq satisfies-def satisfies-instances*)
ultimately show ‹*psatisfiable* ($\Gamma\ \Psi$)›
by (*meson psatisfiable-def satisfies-psatisfies*)
qed (*use* Ψ **in** *auto*)
then show ‹$\exists I.$ *psatisfies* $I\ \Theta$›
using *psatisfiable-def* **by** *blast*
qed (*use qf qfree-formsubst* Γ*-def* **in** *force*)
with *qf that* **show** *thesis*
unfolding Γ*-def psatisfiable-def*
by (*smt* (*verit*, *ccfv-threshold*) *canonical-canon-of-prop interpretation-canon-of-prop*

mem-Collect-eq psatisfies-instances)
qed

lemma *interpretation-restrictlanguage*:
‹$\Psi \subseteq \Phi \Longrightarrow$ *is-interpretation* (*language* Φ) $\mathcal{M} \Longrightarrow$ *is-interpretation* (*language* Ψ)
$\mathcal{M}$›
unfolding *is-interpretation-def language-def functions-forms-def predicates-def*

by (*metis Union-mono fstI image-mono in-mono*)

lemma *interpretation-extendlanguage*:
fixes $\mathcal{M}$:: ‹'*a intrp*›
assumes *int*: ‹*is-interpretation* (*language* Ψ) $\mathcal{M}$› **and** ‹*dom* $\mathcal{M} \neq \{\}$›
and ‹*satisfies* $\mathcal{M}$ Ψ›
obtains $\mathcal{N}$ **where** ‹*dom* $\mathcal{N}$ = *dom* $\mathcal{M}$› ‹*intrp-rel* $\mathcal{N}$ = *intrp-rel* $\mathcal{M}$›
‹*is-interpretation* (*language* Φ) $\mathcal{N}$› ‹*satisfies* $\mathcal{N}$ Ψ›
proof
define *m* **where** ‹*m* ≡ (*SOME a*. *a* ∈ *dom* $\mathcal{M}$)›
have *m*: ‹*m* ∈ *dom* $\mathcal{M}$›
by (*simp add*: ‹*dom* $\mathcal{M} \neq \{\}$› *m-def some-in-eq*)
define $\mathcal{N}$ **where** ‹$\mathcal{N}$ ≡ *Abs-intrp*
(*dom* $\mathcal{M}$,
(λ*g zs*. *if* (*g*,*length zs*) ∈ *functions-forms* Ψ *then intrp-fn* $\mathcal{M}$ *g zs*
else m),
intrp-rel $\mathcal{M}$)›
show *eq*: ‹*dom* $\mathcal{N}$ = *dom* $\mathcal{M}$› ‹*intrp-rel* $\mathcal{N}$ = *intrp-rel* $\mathcal{M}$›
by (*simp-all add*: $\mathcal{N}$*-def*)
show ‹*is-interpretation* (*language* Φ) $\mathcal{N}$›
proof −
have §: ‹*fst* (*language* Ψ) = *functions-forms* Ψ›
by (*simp add*: *language-def*)
obtain β **where** ‹*is-valuation* $\mathcal{M}$ (β $\mathcal{M}$)›
by (*meson* ‹*dom* $\mathcal{M} \neq \{\}$› *valuation-exists*)
then have ‹∀ *n*. β $\mathcal{M}$ *n* ∈ *dom* $\mathcal{M}$›
using *is-valuation-def* **by** *blast*
with *eq m int* **show** *?thesis*
unfolding $\mathcal{N}$*-def is-interpretation-def*
by (*smt* (*verit*, *ccfv-SIG*) § *intrp-fn-Abs-is-fst-snd intrp-is-struct*)
qed
show ‹*satisfies* $\mathcal{N}$ Ψ›
unfolding *satisfies-def*
proof (*intro strip*)
fix β φ
assume β: ‹*is-valuation* $\mathcal{N}$ β› **and** ‹φ ∈ Ψ›
then have ‹*is-valuation* $\mathcal{M}$ β›
by (*simp add*: *eq is-valuation-def*)
then have ‹$\mathcal{M}$,β ⊨ φ›
using ‹φ ∈ Ψ› ‹*satisfies* $\mathcal{M}$ Ψ› **by** (*simp add*: *satisfies-def*)
moreover
have ‹($\mathcal{N}$,β ⊨ φ) ⟷ ($\mathcal{M}$,β ⊨ φ)›
proof (*intro holds-cong*)
fix *f* :: *nat* **and** *ts* :: ‹'*a list*›
assume ‹(*f*, *length ts*) ∈ *functions-form* φ›
then show ‹*intrp-fn* $\mathcal{N}$ *f ts* = *intrp-fn* $\mathcal{M}$ *f ts*›
using $\mathcal{N}$*-def* ‹φ ∈ Ψ› *functions-forms-def* **by** *auto*
qed (*auto simp*: *eq*)

ultimately show ‹$\mathcal{N}$,$\beta \models \varphi$› **by** *auto*
qed
qed

theorem *compact-ls*:
assumes ‹$\bigwedge\Psi$. $[\![$*finite* Ψ; $\Psi \subseteq \Phi]\!]$
$\Longrightarrow \exists \mathcal{M}$::*'a intrp*. *is-interpretation* (*language* Φ) $\mathcal{M} \wedge$ *dom* $\mathcal{M} \neq \{\} \wedge$ *satisfies* $\mathcal{M}$ Ψ›
obtains $\mathcal{C}$::‹*nterm intrp*› **where** ‹*is-interpretation* (*language* Φ) $\mathcal{C}$› ‹*dom* $\mathcal{C} \neq \{\}$› ‹*satisfies* $\mathcal{C}$ Φ›
proof −
have ‹$\exists \mathcal{M}$::*'a intrp*. *is-interpretation*(*language* (*skolem* ‘ Φ)) $\mathcal{M} \wedge$
dom $\mathcal{M} \neq \{\} \wedge$ *satisfies* $\mathcal{M}$ Ψ›
if Ψ: ‹*finite* Ψ› ‹$\Psi \subseteq$ *skolem* ‘ Φ› **for** Ψ
by (*smt* (*verit*, *ccfv-threshold*) *assms finite-subset-image interpretation-extendlanguage*
interpretation-restrictlanguage skolem-satisfiable that)
with *compact-canon-qfree skolem-qfree*
obtain $\mathcal{C}$ **where** $\mathcal{C}$: ‹*is-interpretation* (*language* (*skolem* ‘ Φ)) $\mathcal{C}$›
‹*canonical* (*language* (*skolem* ‘ Φ)) $\mathcal{C}$›
‹*satisfies* $\mathcal{C}$ (*skolem* ‘ Φ)›
by *blast*
have ‹*dom* $\mathcal{C} \neq \{\}$›
using *struct-def* **by** *blast*
with *skolem-satisfiable*[*of* Φ] $\mathcal{C}$ *that* **show** *thesis*
by *metis*
qed

lemma *canon*:
assumes ‹*is-interpretation* (*language* Φ) $\mathcal{M}$› ‹*dom* $\mathcal{M} \neq \{\}$› ‹*satisfies* $\mathcal{M}$ Φ›
obtains $\mathcal{C}$::‹*nterm intrp*› **where** ‹*is-interpretation* (*language* Φ) $\mathcal{C}$› ‹*dom* $\mathcal{C} \neq \{\}$› ‹*satisfies* $\mathcal{C}$ Φ›
using *compact-ls assms* **unfolding** *satisfies-def* **by** *blast*

definition *lowmod* :: ‹*nterm intrp* ⇒ *nat intrp*› **where**
‹*lowmod* $\mathcal{M}$ ≡ *Abs-intrp* (*num-of-term* ‘ (*dom* $\mathcal{M}$),
(λ*g ns*. *num-of-term* (*intrp-fn* $\mathcal{M}$ *g* (*map term-of-num ns*))),
(λ*p*. {*ns*. (*map term-of-num ns*) ∈ *intrp-rel* $\mathcal{M}$ *p*}))›

lemma *dom-lowmod* [*simp*]: ‹*dom* (*lowmod* $\mathcal{M}$) = *num-of-term* ‘ (*dom* $\mathcal{M}$)›
by (*metis* (*no-types*, *lifting*) *dom-Abs-is-fst image-is-empty intrp-is-struct lowmod-def struct-def*)

lemma *intrp-fn-lowmod* [*simp*]: ‹*intrp-fn* (*lowmod* $\mathcal{M}$) *f ns* = *num-of-term* (*intrp-fn* $\mathcal{M}$ *f* (*map term-of-num ns*))›
by (*metis dom-lowmod intrp-fn-Abs-is-fst-snd intrp-is-struct lowmod-def*)

lemma *intrp-rel-lowmod* [*simp*]: ‹*intrp-rel* (*lowmod* $\mathcal{M}$) p = {*ns*. (*map term-of-num ns*) ∈ *intrp-rel* $\mathcal{M}$ p}›
by (*metis* (*no-types*, *lifting*) *dom-lowmod intrp-is-struct intrp-rel-Abs-is-snd-snd lowmod-def*)

lemma *is-valuation-lowmod*: ‹*is-valuation* (*lowmod* $\mathcal{C}$) (*num-of-term* ∘ β) ⟷ *is-valuation* $\mathcal{C}$ β›
by (*simp add*: *is-valuation-def image-iff num-of-term-inj*)

lemma *lowmod-dom-empty*: ‹*dom* (*lowmod* $\mathcal{M}$) = {} ⟷ *dom* $\mathcal{M}$ = {}›
by *simp*

lemma *lowmod-termval*:
assumes ‹*is-valuation* (*lowmod* $\mathcal{M}$) β›
shows ‹*eval t* (*lowmod* $\mathcal{M}$) β = *num-of-term* (*eval t* $\mathcal{M}$ (*term-of-num* ∘ β))›
proof (*induction t*)
case (*Var x*)
then show *?case*
using *assms* **unfolding** *is-valuation-def*
by (*metis* (*no-types*, *lifting*) *comp-apply dom-lowmod eval.simps*(*1*) *image-iff term-of-num-of-term*)
next
case (*Fun f args*)
then show *?case*
using *assms* **unfolding** *is-valuation-def comp-apply intrp-fn-lowmod eval.simps*
by (*smt* (*verit*) *concat-map map-eq-conv term-of-num-of-term*)
qed

lemma *lowmod-holds*:
assumes ‹*is-valuation* (*lowmod* $\mathcal{M}$) β›
shows ‹(*lowmod* $\mathcal{M}$),β ⊨ φ ⟷ $\mathcal{M}$,(*term-of-num* ∘ β) ⊨ φ›
using *assms*
proof (*induction* φ *arbitrary*: β)
case (*Atom x1 x2*)
then show *?case*
using *lowmod-termval* [*OF Atom*] **by** (*simp add*: *comp-def*)
next
case (*Forall x1* φ)
then have ‹⋀*a*. *a* ∈ *dom* $\mathcal{M}$ ⟹ *is-valuation* (*lowmod* $\mathcal{M}$) (β(*x1* := *num-of-term a*))›
by (*simp add*: *valuation-valmod*)
with *Forall* **show** *?case*
apply *simp*
by (*smt* (*verit*, *best*) *fun-upd-apply holds-indep-β-if term-of-num-of-term*)
qed *auto*

lemma *lowmod-intrp*: ‹*is-interpretation* $\mathcal{L}$ (*lowmod* $\mathcal{M}$) = *is-interpretation* $\mathcal{L}$ $\mathcal{M}$›
proof
have *inj*: ‹*inj num-of-term*›
by (*meson injI num-of-term-inj*)
show ‹*is-interpretation* $\mathcal{L}$ (*lowmod* $\mathcal{M}$) $\Longrightarrow$ *is-interpretation* $\mathcal{L}$ $\mathcal{M}$›
unfolding *is-interpretation-def dom-lowmod intrp-fn-lowmod inj-image-mem-iff* [*OF inj*]
by (*metis* (*no-types*, *lifting*) *concat-map image-mono image-set length-map map-idI term-of-num-of-term*)
show ‹*is-interpretation* $\mathcal{L}$ $\mathcal{M}$ $\Longrightarrow$ *is-interpretation* $\mathcal{L}$ (*lowmod* $\mathcal{M}$)›
unfolding *is-interpretation-def dom-lowmod intrp-fn-lowmod subset-iff*
by (*smt* (*verit*, *best*) *image-iff length-map list.set-map term-of-num-of-term*)
qed

lemma *loewenheim-skolem*:
assumes ‹*is-interpretation* (*language* Φ) $\mathcal{M}$› ‹*dom* $\mathcal{M} \neq \{\}$›
assumes ‹$\bigwedge\varphi$. $\varphi \in \Phi \Longrightarrow$ *qfree* φ› ‹*satisfies* $\mathcal{M}$ Φ›
obtains $\mathcal{N}$ :: ‹*nat intrp*› **where** ‹*is-interpretation* (*language* Φ) $\mathcal{N}$› ‹*dom* $\mathcal{N} \neq \{\}$› ‹*satisfies* $\mathcal{N}$ Φ›
proof −
obtain $\mathcal{C}$::‹*nterm intrp*› **where** $\mathcal{C}$: ‹*is-interpretation* (*language* Φ) $\mathcal{C}$› ‹*dom* $\mathcal{C} \neq \{\}$› ‹*satisfies* $\mathcal{C}$ Φ›
using *assms canon* **by** *blast*
show *?thesis*
proof
show ‹*is-interpretation* (*language* Φ) (*lowmod* $\mathcal{C}$)›
by (*simp add*: $\mathcal{C}$ *lowmod-intrp*)
show ‹*dom* (*lowmod* $\mathcal{C}$) $\neq \{\}$›
by (*simp add*: $\mathcal{C}$)
show ‹*satisfies* (*lowmod* $\mathcal{C}$) Φ›
using $\mathcal{C}$ **unfolding** *satisfies-def*
by (*smt* (*verit*, *ccfv-SIG*) *comp-apply dom-lowmod image-iff is-valuation-def lowmod-holds term-of-num-of-term*)
qed
qed

theorem *uniformity*:
assumes ‹*qfree* φ›
‹$\bigwedge\mathcal{C}$::*nterm intrp*. $\bigwedge\beta$. ⟦*dom* $\mathcal{C} \neq \{\}$; *is-valuation* $\mathcal{C}$ β⟧ $\Longrightarrow$ $\mathcal{C}$,β $\models$ *foldr Exists xs* φ›
obtains σs **where** ‹$\bigwedge\sigma$ *x*. σ *set* $\sigma s \Longrightarrow \sigma$ *x* ∈ *terms-set* (*fst* (*language* {φ}))›
‹$\bigwedge I$. $I \models_p$ (*foldr* ($\lambda\varphi$ ψ. $\varphi \vee \psi$) (*map* ($\lambda\sigma$. $\varphi \cdot_{fm} \sigma$) σs) $\bot$)›
proof −
define *A* **where** ‹*A* ≡ *formsubst* φ ‘ {σ. ∀ *x*. σ *x* ∈ *terms-set* (*fst* (*language* {φ}))}›

```
  have ‹∃φ′ ∈ A. I ⊨p φ′› for I
  proof −
    have ∗: False if ‹satisfies (canon-of-prop (language {φ}) I) {¬ φ}›
    proof −
      obtain β where β: ‹is-valuation (canon-of-prop (language {φ}) I) β›
        by (metis struct-def valuation-exists intrp-is-struct)
      then have ‹canon-of-prop (language {φ}) I, β ⊨ foldr Exists xs φ›
        using assms struct-def by fastforce
      then obtain as where len: ‹length as = length xs›
            and sub: ‹set as ⊆ terms-set (fst (language {φ}))›
            and sat0: ‹canon-of-prop (language {φ}) I,
                    foldr (λu β. β(fst u := snd u)) (rev (zip xs as)) β ⊨ φ›
        by (force simp add: holds-itlist-exists)
      define F where ‹F ≡ λas::nterm list. λxs::nat list. foldr (λu β. β(fst u :=
snd u)) (rev (zip xs as))›
      have F-Cons: ‹F (a#as) (x#xs) β = F as xs ((β(x := a)))› for a as x xs β
        by (simp add: F-def)
      have sat: ‹canon-of-prop (language {φ}) I, F as xs β ⊨ φ›
        using sat0 by (simp add: F-def)
      have ‹⟦length as = length xs;
            set as ⊆ dom (canon-of-prop (language {φ}) I);
            is-valuation (canon-of-prop (language {φ}) I) β⟧
           ⟹ is-valuation (canon-of-prop (language {φ}) I) (F as xs β)›
        for xs as
      proof (induction xs arbitrary: as β)
        case Nil
        then show ?case
          by (simp add: F-def is-valuation-def)
      next
        case (Cons x xs as β)
        then obtain a as′ where aas′: ‹as = a#as′› ‹length as′ = length xs›
          by (metis length-Suc-conv)
        with Cons show ?case
          by (simp add: is-valuation-def aas′ F-Cons)
      qed
      then have ‹is-valuation (canon-of-prop (language {φ}) I) (F as xs β)›
        by (metis (no-types, lifting) β dom-canon-of-prop len sub)
      then show ?thesis
        using sat satisfies-def that by (force simp: F-def)
    qed
    then show ?thesis
      using psatisfies-instances [of concl: ‹language {φ}› I ‹{¬ φ}›] ‹qfree φ›
      by (force simp: A-def)
  qed
  then obtain Φ where Φ: ‹set Φ ⊆ A› ‹⋀I. I ⊨p foldr (∨) Φ ⊥›
   by (smt (verit, ccfv-threshold) A-def assms(1) compact-disj image-iff qfree-formsubst)
  show thesis
  proof
    define sf where ‹sf ≡ λq. @σ. (∀ i. σ i ∈ terms-set (fst (language {φ}))) ∧ q
```

```
= φ ·fm σ›
    have sf-works: ‹(∀ i. sf a i ∈ terms-set (fst (language {φ}))) ∧ a = φ ·fm (sf
a)›
      if ‹a ∈ A› for a
      using that unfolding A-def sf-def image-iff Bex-def mem-Collect-eq
      by (rule someI-ex)
    show ‹σ i ∈ terms-set (fst (language {φ}))›
      if ‹σ ∈ set (map sf Φ)› for σ i
    proof −
     have ∗: ‹set Θ ⊆ A ⟹ σ ∈ set (map sf Θ) ⟹ σ i ∈ terms-set (fst (language
{φ}))› for Θ
        by (induction Θ) (auto simp: sf-works)
      then show ?thesis
        using Φ that by fastforce
    qed
    show ‹I ⊨p foldr (∨) (map ((·fm) φ) (map sf Φ)) ⊥› for I
      using Φ(2) [of I] Φ(1)
    by (induction Φ) (use sf-works in force)+
  qed
qed

end
```

References

[1] J. Harrison. Formalizing basic first order model theory. In *TPHOLs*, volume 1479 of *Lecture Notes in Computer Science*, pages 153–170. Springer, 1998.