A deep embedding of HOL in HOL: soundness, completeness, consistency

Christoph Benzmüller Daniel Kirchner

August 7, 2026

Abstract

This entry presents a minimal, machine-checked deep embedding of classical higher-order logic (HOL) in Isabelle/HOL, following Benzmüller, Brown and Kohlhase (BKK) [4]. Throughout, HOL means Church's simple theory of types [5]: it is the embedded *object* logic, while Isabelle/HOL serves as the ambient meta-logic. The language of BKK §2 is formalised in a *locally nameless* representation, which makes BKK's convention of identifying α-equivalent terms literally true and eliminates capture-avoiding substitution and renaming machinery altogether. The semantics of BKK §3 is rendered abstractly: applicative structures, Σ-evaluations $(D, @, E)$ subject to BKK's four conditions on the evaluation function, Σ-models $(D, @, E, \upsilon)$, and the class $\mathcal{M}_{\beta\mathfrak{fb}}$ of Σ-Henkin models (BKK's property q is automatic here, since primitive equality is part of the signature). Standard models arise as the full special case, and the familiar recursive denotation over frames enters only as the canonical way of constructing concrete models.

On this basis we prove the natural-deduction calculus NK of BKK §7 sound (Theorem 7.3, including BKK's evaluation-variant argument) and complete in the sense of Henkin, via the abstract-consistency and model-existence method of BKK §6, with the term model realised as a term evaluation. Completeness is established in a form stronger than BKK's Corollary 7.7: it holds at every infinite value carrier, for every signature with infinitely many parameters, and for open formulas. Consistency follows by exhibiting a concrete Σ-standard model over finite domains. The signature includes BKK's optional primitive equality (Remark 7.9) and—going beyond BKK, following Andrews [1]—typed description operators. As an illustration, Cantor's theorem is derived *inside* NK, in surjective and injective form and at every type, with the Cantor sentences stated as in the Stanford Encyclopedia entry on Church's type theory [3].

What is new. To the best of our knowledge, this entry contains the first machine-checked *completeness* proof for a deep embedding of HOL in HOL. The most closely related AFP entry, *Metatheory of* $\mathcal{Q}_0$ by Díaz [6], formalises

Andrews' system $\mathcal{Q}_0$ ([2], Chapter 5) with named variables up to and including soundness and consistency; completeness is not covered there. Harrison's classic self-verification of HOL Light [8] likewise concerns soundness and consistency of HOL in HOL, not completeness. The nearest completeness mechanisations concern weaker logics: Koch and Kirst [9] prove completeness of second-order logic with respect to Henkin semantics in Coq, by translation to mono-sorted first-order logic — second-order logic is a proper fragment of Church's type theory, and the translation route differs from a direct model-existence proof for a higher-order calculus — and From [7] provides a synthetic-completeness framework in Isabelle/HOL, instantiated to propositional, first-order, modal and hybrid logic. Second, the entry demonstrates that BKK's abstract consistency / model-existence technique (BKK §6)—the basis of their uniform completeness proofs for the whole landscape of model classes—can indeed be formalised. Third, the formalisation *extends* BKK by typed description operators, following Andrews [1]: the rule $NK(\iota)$ and the corresponding model condition are carried through soundness, model existence and completeness. Finally, completeness is established in a form stronger than BKK's Corollary 7.7: for open formulas, at every infinite value carrier, and for every signature with infinitely many parameters, with no cardinality link between signature and carrier.

This contribution is part of the LogiKEy project (https://logikey.org).

Contents

1 Syntax: a deep embedding of HOL in HOL **3**

2 Semantics: applicative structures, Henkin models and standard models **14**

3 The natural-deduction calculus NK **32**

4 Soundness **37**

5 Completeness **48**

6 Consistency **83**

7 Example: Cantor's theorem **88**

8 Main results **95**

1 Syntax: a deep embedding of HOL in HOL

BKK's language of classical higher-order logic — HOL, by which we mean Church's simple theory of types throughout (BKK Sections 2.1–2.2) — in a *locally nameless* representation: bound variables are de Bruijn indices, free variables and parameters carry their type. BKK take alphabetic variants to be identical (BKK Section 2.1); locally nameless makes that literally true — α-equivalent terms are *equal*, so capture-avoiding substitution and the entire renaming theory disappear. As in BKK, non-logical constants are *parameters* with names drawn from a type $'p$, and we include BKK's optional primitive equality *Eq* σ (BKK Section 2.1, Remark 7.9), alongside the always-expressible defined Leibniz equality (BKK Section 2.2). Beyond BKK's signature we deliberately carry a family of description operators *Iota* σ of type $(\sigma \Rightarrow \mathrm{o}) \Rightarrow \sigma$, one for each type: BKK have no description operator (they decline to add one, BKK Section 2.3.1, pointing to Andrews 1972 for the semantic issues); we follow that reference instead and equip *Iota* σ with the description axiom for singletons — the rule $NK(\iota)$ of the calculus and the corresponding model condition *gm-descB*. Definitions and lemmas below that carry no BKK reference are locally-nameless infrastructure (opening, closing, freshness, renaming): they have no counterpart in the paper, where identification of alphabetic variants is handled informally (BKK Section 2.1).

1.1 Types and terms

datatype *ty* = *Ind* (‹ι›) | *Bool* (‹o›) | *Fun ty ty* (**infixr** ‹⇒› *65*)

datatype (*pars*: $'p$) *tm* =
  Bnd nat — bound variable (de Bruijn index)
| *Fre nat ty* — free variable: a name and its type
| *Par* $'p$ *ty* — parameter (typed constant)
| *Neg* | *Dis* | *Pi ty* | *Iota ty* | *Eq ty*
  — logical constants $\neg$, $\vee$, $\mathbf{\Pi}_\sigma$, $\boldsymbol{\iota}_\sigma$, $=_\sigma$
| *App* $'p$ *tm* $'p$ *tm* (**infixl** ‹·› *200*)
| *Abs ty* $'p$ *tm* — abstraction, domain type annotated (BKK's $\lambda X_\sigma.\ A$; nameless, so no binder variable)
for *map*: *prn* — We call the map function for the parameter type *prn* for parameter renaming.

BKK write disjunctions in infix and negations in prefix notation (BKK Section 2.2 declares the convention of writing $((\vee A)B)$ as $A \vee B$); we mirror this with input/output abbreviations for the applied connectives.

Abstraction is written $\mathbf{\Lambda}_\sigma\ b$ for BKK's $\lambda X_\sigma.\ A$ (bold Λ, since λ is reserved); being nameless, it displays the domain type but no binder variable.

notation *Abs* (‹**Λ**_ -› [*0*, *200*] *200*)

Decorated atoms: a free variable x of type σ is written ${x^f}_\sigma$ (BKK's X_σ),

a parameter w is written $w^p{}_\sigma$ (BKK's typed constants).

notation *Fre* ($\langle$-^{f}_$\rangle$ [*1000*, *0*] *1000*)
notation *Par* ($\langle$-^{p}_$\rangle$ [*1000*, *0*] *1000*)

abbreviation *NegA* :: *'p tm* ⇒ *'p tm* ($\langle\neg$ -$\rangle$ [*66*] *66*) **where**
$\neg\ A \equiv Neg \cdot A$
abbreviation *DisA* :: *'p tm* ⇒ *'p tm* ⇒ *'p tm* (**infixr** $\langle\vee\rangle$ *61*) **where**
$A \vee B \equiv (Dis \cdot A) \cdot B$

Summary of the term notation and its BKK Section 2 counterparts: application $s \cdot t$ (BKK juxtaposition), abstraction $\mathbf{\Lambda}_\sigma\ b$, free variables $x^f{}_\sigma$ (BKK's X_σ), parameters $w^p{}_\sigma$ (typed constants), and the applied connectives $\neg\ A$ and $A \vee B$. The defined layer adds $A \supset B$, $\mathbf{\Pi}_\sigma\ b$, Leibniz equality $A \doteq_\alpha B$ and applied primitive equality (below); opening is $b\langle u\rangle$ (below); on the semantic side, $(\!|A|\!)_\xi$ is the denotation and $\xi(x_\sigma := d)$ the assignment update (Section 2).

Types and terms (over a countable parameter type) are countable, so the language of sentences can be enumerated — the basis for our countable rendering of BKK's transfinite extension construction (BKK Section 6, Lemma 6.32).

instance *ty* :: *countable* **by** *countable-datatype*
instance *tm* :: (*countable*) *countable* **by** *countable-datatype*

1.2 Opening and free-variable substitution

$opn_k\ u\ t$ replaces the bound index k in t by u; $t(\!|u|\!)$ opens the top binder. Because bound variables are indices, substituting for a free variable (*fsub*) needs no renaming: it passes straight through *Abs*.

primrec *opn* :: *nat* ⇒ *'p tm* ⇒ *'p tm* ⇒ *'p tm* **where**
opn k u (Bnd i) = (if i = k then u else Bnd i)
| *opn k u (n^f_σ) = n^f_σ*
| *opn k u (p^p_σ) = p^p_σ*
| *opn k u Neg = Neg*
| *opn k u Dis = Dis*
| *opn k u (Pi σ) = Pi σ*
| *opn k u (Iota τ) = Iota τ*
| *opn k u (Eq τ) = Eq τ*
| *opn k u (s · t) = (opn k u s) · (opn k u t)*
| *opn k u (Λ_σ b) = Λ_σ (opn (Suc k) u b)*

Opening the outermost binder is by far the most frequent operation, so it gets its own notation: $b\langle u\rangle$ is b with de Bruijn index *0* instantiated to u — BKK's $[u/X]B$ for the bound variable of the enclosing abstraction or quantifier.

abbreviation *opn0* ($\langle$-$\langle$-$\rangle\rangle$ [*1000*, *0*] *1000*) **where** $b\langle u\rangle \equiv opn\ 0\ u\ b$

primrec *fsub* :: *nat* ⇒ *ty* ⇒ *'p tm* ⇒ *'p tm* ⇒ *'p tm* **where**
  fsub x σ u (*Bnd i*) = *Bnd i*
| *fsub x σ u* ($n^f\ \tau$) = (*if n* = *x* ∧ *τ* = *σ then u else* $n^f\ \tau$)
| *fsub x σ u* ($p^p\ \tau$) = $p^p\ \tau$
| *fsub x σ u Neg* = *Neg*
| *fsub x σ u Dis* = *Dis*
| *fsub x σ u* (*Pi τ*) = *Pi τ*
| *fsub x σ u* (*Iota τ*) = *Iota τ*
| *fsub x σ u* (*Eq τ*) = *Eq τ*
| *fsub x σ u* (*s* · *t*) = (*fsub x σ u s*) · (*fsub x σ u t*)
| *fsub x σ u* (**Λ**$_\tau$ *b*) = **Λ**$_\tau$ (*fsub x σ u b*)

Substitution notation: $b[u/x_\sigma]$ substitutes u for the free variable $x^f{}_\sigma$ in b.

syntax *-fsub* :: *logic* ⇒ *logic* ⇒ *logic* ⇒ *logic* ⇒ *logic* (‹-[-′/-]› [*1000*, *0*, *0*, *0*] *1000*)
syntax-consts *-fsub* == *fsub*
translations $b[u/x_\sigma]$ ⇌ *CONST fsub x σ u b*

1.3 Free variables and parameters

primrec *fvs* :: *'p tm* ⇒ *nat set* **where**
  fvs (*Bnd i*) = {}
| *fvs* ($n^f\ \sigma$) = {*n*}
| *fvs* ($p^p\ \sigma$) = {}
| *fvs Neg* = {} | *fvs Dis* = {} | *fvs* (*Pi σ*) = {}
| *fvs* (*Iota σ*) = {} | *fvs* (*Eq σ*) = {}
| *fvs* (*s* · *t*) = *fvs s* ∪ *fvs t*
| *fvs* (**Λ**$_\sigma$ *b*) = *fvs b*

lemma *finite-fvs* [*simp*]: *finite* (*fvs t*) **by** (*induction t*) *auto*
lemma *finite-pars* [*simp*]: *finite* (*pars t*) **by** (*induction t*) *auto*

Renaming parameters. Eigen-parameters (BKK's w_α) must be chosen fresh; to weaken the context or extend a consistent set we move them out of the way with an injective renaming.

lemma *prn-opn*: *prn ϱ* (*opn k u t*) = *opn k* (*prn ϱ u*) (*prn ϱ t*) **by** (*induction t arbitrary*: *k*) *auto*
lemma *pars-prn*: *pars* (*prn ϱ t*) = *ϱ* ‘ *pars t* **by** (*induction t*) *auto*
lemma *prn-prn*: *prn f* (*prn g t*) = *prn* (*λp*. *f* (*g p*)) *t* **by** (*induction t*) *auto*
lemma *prn-id* [*simp*]: *prn* (*λp*. *p*) *t* = *t* **by** (*induction t*) *auto*
lemma *prn-cong*: (⋀*p*. *p* ∈ *pars t* ⟹ *ϱ p* = *p*) ⟹ *prn ϱ t* = *t* **by** (*induction t*) *auto*

The typed free occurrences — the (name, type) pairs a term reads from an assignment. A name can occur at several types, so this is finer than *fvs*.

primrec *occ* :: *'p tm* ⇒ (*nat* × *ty*) *set* **where**
  occ (*Bnd i*) = {}
| *occ* ($n^f\ \sigma$) = {(*n*, *σ*)}

| *occ* ($p^p{}_\sigma$) = {}
| *occ Neg* = {} | *occ Dis* = {} | *occ* (*Pi* σ) = {}
| *occ* (*Iota* σ) = {} | *occ* (*Eq* σ) = {}
| *occ* ($s \cdot t$) = *occ s* ∪ *occ t*
| *occ* ($\mathbf{\Lambda}_\sigma$ *b*) = *occ b*

lemma *fvs-eq-fst-occ*: *fvs t* = *fst* ‘ *occ t* **by** (*induction t*) (*auto simp*: *image-Un*)
lemma *occ-opn*: *occ* (*opn k u t*) ⊆ *occ t* ∪ *occ u* **by** (*induction t arbitrary*: *k*) *auto*

1.4 A fresh free variable exists

A deterministic fresh name for a finite set (used for the abstraction case of the denotation).

definition *fresh* :: *nat set* ⇒ *nat* **where** *fresh S* ≡ *LEAST n*. *n* ∉ *S*

lemma *fresh-notin*: *finite S* ⟹ *fresh S* ∉ *S*
by (*metis LeastI ex-new-if-finite infinite-UNIV-nat fresh-def*)

1.5 Basic laws of opening and substitution

Free variables of a substitution. (A name may occur at several types, so x is only removed under the safe over-approximation.)

lemma *fvs-fsub*: *fvs* (*fsub x* σ *u t*) ⊆ *fvs t* ∪ *fvs u* **by** (*induction t*) *auto*
lemma *fvs-opn*: *fvs* (*opn k u t*) ⊆ *fvs t* ∪ *fvs u* **by** (*induction t arbitrary*: *k*) *auto*

Opening with a free variable preserves size — the measure for the size-recursive denotation of an abstraction (its body is opened with a fresh variable of the same size).

lemma *size-opn-Fre* [*simp*]: *size* (*opn k* ($x^f{}_\sigma$) *t*) = *size t* **by** (*induction t arbitrary*: *k*) *auto*

1.6 Local closure

A term is *locally closed* if every bound index is captured by an enclosing binder. As is standard for the locally-nameless representation, the *Abs* rule uses a *cofinite* quantifier: opening the body with a fresh free variable is locally closed. A locally closed term denotes exactly an α-equivalence class of BKK's named terms (BKK Section 2.1).

inductive *lc* :: ′*p tm* ⇒ *bool* **where**
lc-Fre [*intro*]: *lc* ($n^f{}_\sigma$)
| *lc-Par* [*intro*]: *lc* ($p^p{}_\sigma$)
| *lc-Neg* [*intro*]: *lc Neg*
| *lc-Dis* [*intro*]: *lc Dis*
| *lc-Pi* [*intro*]: *lc* (*Pi* σ)
| *lc-Iota* [*intro*]: *lc* (*Iota* σ)
| *lc-Eq* [*intro*]: *lc* (*Eq* σ)
| *lc-App* [*intro*]: *lc s* ⟹ *lc t* ⟹ *lc* ($s \cdot t$)

| *lc-Abs*: *finite L* $\Longrightarrow$ ($\bigwedge x.\ x \notin L \Longrightarrow lc\ (b\langle x^f{}_\sigma\rangle)) \Longrightarrow lc\ (\mathbf{\Lambda}_\sigma\ b)$

If opening at j is already fixed by a later opening at $i \neq j$, then opening at i alone was already the identity.

lemma *opn-core*: $i \neq j \Longrightarrow opn\ j\ v\ t = opn\ i\ u\ (opn\ j\ v\ t) \Longrightarrow opn\ i\ u\ t = t$
proof (*induction t arbitrary*: *i j*)
  case (*Abs* τ *b*)
  have *opn* (*Suc i*) *u b* = *b*
   by (*metis Abs.IH opn.simps*(*10*) *Abs.prems*(*1*) *tm.inject*(*8*) *Abs.prems*(*2*) *old.nat.inject*)
  thus *?case* **by** *simp*
qed (*auto split*: *if-splits*)

A locally closed term ignores opening.

lemma *opn-lc* [*simp*]: *lc t* $\Longrightarrow$ *opn k u t* = *t*
proof (*induction arbitrary*: *k rule*: *lc.induct*)
  case (*lc-Abs L b* σ)
  obtain *x* **where** *x*: $x \notin L$
    using *lc-Abs.hyps*(*1*) **by** (*meson ex-new-if-finite infinite-UNIV-nat*)
  have *IH*: *opn* (*Suc k*) *u* ($b\langle x^f{}_\sigma\rangle$) = $b\langle x^f{}_\sigma\rangle$ **using** *lc-Abs.IH x* **by** *auto*
  have *opn* (*Suc k*) *u b* = *b* **by** (*metis opn-core IH nat.simps*(*3*))
  thus *?case* **by** *simp*
qed *simp-all*

fsub commutes with opening when the substituted term is locally closed.

lemma *fsub-opn*: *lc u* $\Longrightarrow$ *fsub x* σ *u* (*opn k v t*) = *opn k* (*fsub x* σ *u v*) (*fsub x* σ *u t*)
  by (*induction t arbitrary*: *k*) *auto*

Opening with u equals opening with a fresh free variable and then substituting u for it.

lemma *fsub-intro*: $x \notin fvs\ t \Longrightarrow opn\ k\ u\ t = fsub\ x\ \sigma\ u\ (opn\ k\ (x^f{}_\sigma)\ t)$
  by (*induction t arbitrary*: *k*) *auto*

Openings at distinct indices commute (for locally closed fillers).

lemma *opn-opn-comm*: $i \neq j \Longrightarrow lc\ u \Longrightarrow lc\ v \Longrightarrow opn\ i\ u\ (opn\ j\ v\ t) = opn\ j\ v\ (opn\ i\ u\ t)$
  by (*induction t arbitrary*: *i j*) *auto*

1.7 Typing: well-formed formulae

$wff_\sigma(t)$ is BKK's $t \in wff_\sigma(\Sigma)$ (BKK Section 2.1): t is a well-formed formula of type σ. The *Abs* rule uses the cofinite quantifier, so a well-formed formula is in particular locally closed.

inductive *wff* :: *ty* $\Rightarrow$ *'p tm* $\Rightarrow$ *bool* (‹*wff*_'(-')› [*0*,*0*] *1000*) **where**
    wff-Fre: $wff_\sigma(n^f{}_\sigma)$
  | *wff-Par*: $wff_\sigma(p^p{}_\sigma)$
  | *wff-Neg*: $wff_{o \Rightarrow o}(Neg)$
  | *wff-Dis*: $wff_{o \Rightarrow o \Rightarrow o}(Dis)$

| *wff-Pi*: $\mathit{wff}_{(\sigma \Rightarrow o) \Rightarrow o}(\mathit{Pi}\ \sigma)$
| *wff-Iota*: $\mathit{wff}_{(\sigma \Rightarrow o) \Rightarrow \sigma}(\mathit{Iota}\ \sigma)$
| *wff-Eq*: $\mathit{wff}_{\sigma \Rightarrow \sigma \Rightarrow o}(\mathit{Eq}\ \sigma)$
| *wff-App*: $\mathit{wff}_{\sigma \Rightarrow \tau}(s) \Longrightarrow \mathit{wff}_{\sigma}(t) \Longrightarrow \mathit{wff}_{\tau}(s \cdot t)$
| *wff-Abs*: *finite* $L \Longrightarrow (\bigwedge x.\ x \notin L \Longrightarrow \mathit{wff}_{\tau}(b\langle x^{f}{}_{\sigma}\rangle)) \Longrightarrow \mathit{wff}_{\sigma \Rightarrow \tau}(\mathbf{\Lambda}_{\sigma}\ b)$

lemma *wff-lc*: $\mathit{wff}_{\sigma}(t) \Longrightarrow \mathit{lc}\ t$ **by** (*induction rule*: *wff.induct*) (*auto intro*: *lc-Abs*)

inductive-cases *wff-FreE* [*elim*!]: $\mathit{wff}_{\tau}(n^{f}{}_{\sigma})$
inductive-cases *wff-ParE* [*elim*!]: $\mathit{wff}_{\tau}(p^{p}{}_{\sigma})$
inductive-cases *wff-NegE* [*elim*!]: $\mathit{wff}_{\tau}(\mathit{Neg})$
inductive-cases *wff-DisE* [*elim*!]: $\mathit{wff}_{\tau}(\mathit{Dis})$
inductive-cases *wff-PiE* [*elim*!]: $\mathit{wff}_{\tau}(\mathit{Pi}\ \sigma)$
inductive-cases *wff-IotaE* [*elim*!]: $\mathit{wff}_{\tau}(\mathit{Iota}\ \sigma)$
inductive-cases *wff-EqE* [*elim*!]: $\mathit{wff}_{\tau}(\mathit{Eq}\ \sigma)$
inductive-cases *wff-AppE* [*elim*]: $\mathit{wff}_{\tau}(s \cdot t)$
inductive-cases *wff-AbsE* [*elim*]: $\mathit{wff}_{\tau}(\mathbf{\Lambda}_{\sigma}\ b)$

Types are unique.

lemma *wff-unique*: $\mathit{wff}_{\sigma}(t) \Longrightarrow \mathit{wff}_{\tau}(t) \Longrightarrow \sigma = \tau$
proof (*induction* σ *t arbitrary*: τ *rule*: *wff.induct*)
  case (*wff-Abs* $L\ \varrho\ b\ \sigma$)
  from *wff-Abs.prems* **obtain** $L'\ \varrho'$ **where** *t*: $\tau = \sigma \Rightarrow \varrho'$
    and *fL'*: *finite* L' **and** *hb*: $\bigwedge x.\ x \notin L' \Longrightarrow \mathit{wff}_{\varrho'}(b\langle x^{f}{}_{\sigma}\rangle)$
    by *auto*
  obtain *x* **where** *x*: $x \notin L \cup L'$ **using** *wff-Abs*(*1*) *fL'*
    by (*meson ex-new-if-finite finite-UnI infinite-UNIV-nat*)
  have $\varrho = \varrho'$ **using** *wff-Abs.IH hb x* **by** *auto*
  thus *?case* **using** *t* **by** *simp*
qed (*fast*)+

Well-typedness is preserved when a free variable is replaced by a term of the same type (BKK Section 2.1: substitution respects the typing; the paper leaves this implicit).

lemma *wff-fsub*: $\mathit{wff}_{\tau}(t) \Longrightarrow \mathit{wff}_{\varrho}(u) \Longrightarrow \mathit{wff}_{\tau}(\mathit{fsub}\ x\ \varrho\ u\ t)$
proof (*induction* τ *t rule*: *wff.induct*)
  case (*wff-Abs* $L\ \tau\ b\ \sigma$)
  have *lcu*: *lc u* **using** *wff-Abs.prems* **by** (*rule wff-lc*)
  have $\mathit{wff}_{\sigma \Rightarrow \tau}(\mathbf{\Lambda}_{\sigma}\ (\mathit{fsub}\ x\ \varrho\ u\ b))$
    by (*metis finite-insert fsub.simps*(*2*) *fsub-opn insert-iff lcu wff.wff-Abs*
        wff-Abs.IH wff-Abs.hyps(*1*) *wff-Abs.prems*)
  thus *?case* **by** *simp*
qed (*auto intro*: *wff.intros simp*: *wff.wff-Fre*)

Consequently an abstraction may be opened with *any* fresh free variable and stays well-typed — the locally-nameless counterpart of BKK's α-invariance (BKK Section 2.1).

lemma *wff-Abs-open*: **assumes** *w*: $\mathit{wff}_{\sigma \Rightarrow \tau}(\mathbf{\Lambda}_{\sigma}\ b)$ **and** *x*: $x \notin \mathit{fvs}\ b$

shows $wff_\tau(b\langle x^f{}_\sigma\rangle)$
proof −
from w **obtain** $L\ \tau'$ **where** $L\tau'$: *finite* L $\tau = \tau'$ $\bigwedge y.\ y \notin L \Longrightarrow wff_{\tau'}(b\langle y^f{}_\sigma\rangle)$
by *auto*
obtain y **where** y: $y \notin L \cup fvs\ b$
using $L\tau'(1)$ **by** (*meson ex-new-if-finite finite-UnI finite-fvs infinite-UNIV-nat*)
have $b\langle x^f{}_\sigma\rangle = fsub\ y\ \sigma\ (x^f{}_\sigma)\ (b\langle y^f{}_\sigma\rangle)$ **using** y
by (*intro fsub-intro*) *auto*
moreover have $wff_{\tau'}(fsub\ y\ \sigma\ (x^f{}_\sigma)\ (b\langle y^f{}_\sigma\rangle))$ **using** $L\tau'(3)$ y *wff-Fre*
by (*auto intro*: *wff-fsub*)
ultimately show *?thesis* **using** $L\tau'(2)$ **by** *simp*
qed

Opening a well-typed abstraction with a well-typed argument stays well-typed — the typing counterpart of β-reduction (implicit in BKK Section 2.1; the semantic analogue is BKK Lemma 3.20).

lemma *wff-opn*: $wff_{\sigma\Rightarrow\tau}(\mathbf{\Lambda}_\sigma\ b) \Longrightarrow wff_\sigma(a) \Longrightarrow wff_\tau(b\langle a\rangle)$
by (*metis finite-fvs fresh-notin fsub-intro wff-Abs-open wff-fsub*)

Parameter renaming leaves the typing unchanged (there is no BKK counterpart: parameter renaming is part of the locally-nameless infrastructure).

lemma *wff-prn*[*intro*!]: $wff_\sigma(t) \Longrightarrow wff_\sigma(prn\ \varrho\ t)$
proof (*induction* σ t *rule*: *wff.induct*)
case *wff-Abs* **thus** *?case* **by** (*metis* (*mono-tags*, *lifting*) *prn-opn tm.simps*(*120*,*128*) *wff.wff-Abs*)
qed (*auto intro*: *wff.intros*)

1.8 Turning one parameter into a free variable

pvar $w\ \sigma\ x\ t$ replaces every occurrence of the parameter w at type σ by the free variable x. This realizes the evaluation variant of BKK's proof of Theorem 7.3 (case $NK(\Pi I)$): from an evaluation $\mathcal{E}$ "one can define another evaluation function $\mathcal{E}'$ such that $\mathcal{E}'(w) \equiv a$ and $\mathcal{E}'(A) \equiv \mathcal{E}(A)$ if w does not occur in A".

primrec *pvar* :: $'p \Rightarrow ty \Rightarrow nat \Rightarrow 'p\ tm \Rightarrow 'p\ tm$ **where**
  $pvar\ w\ \sigma\ x\ (Bnd\ i) = Bnd\ i$
| $pvar\ w\ \sigma\ x\ (n^f{}_\tau) = n^f{}_\tau$
| $pvar\ w\ \sigma\ x\ (p^p{}_\tau) = (if\ p = w \wedge \tau = \sigma\ then\ x^f{}_\sigma\ else\ p^p{}_\tau)$
| $pvar\ w\ \sigma\ x\ Neg = Neg$
| $pvar\ w\ \sigma\ x\ Dis = Dis$
| $pvar\ w\ \sigma\ x\ (Pi\ \tau) = Pi\ \tau$
| $pvar\ w\ \sigma\ x\ (Iota\ \tau) = Iota\ \tau$
| $pvar\ w\ \sigma\ x\ (Eq\ \tau) = Eq\ \tau$
| $pvar\ w\ \sigma\ x\ (s \cdot t) = (pvar\ w\ \sigma\ x\ s) \cdot (pvar\ w\ \sigma\ x\ t)$
| $pvar\ w\ \sigma\ x\ (\mathbf{\Lambda}_\tau\ b) = \mathbf{\Lambda}_\tau\ (pvar\ w\ \sigma\ x\ b)$

lemma *pvar-opn*: $pvar\ w\ \sigma\ x\ (opn\ k\ u\ t) = opn\ k\ (pvar\ w\ \sigma\ x\ u)\ (pvar\ w\ \sigma\ x\ t)$
by (*induction* t *arbitrary*: k) *auto*

lemma *pvar-id* [*simp*]: $w \notin pars\ t \Longrightarrow pvar\ w\ \sigma\ x\ t = t$
by (*induction t*) *auto*
lemma *wff-pvar*: $wff_\tau(t) \Longrightarrow wff_\tau(pvar\ w\ \sigma\ x\ t)$
proof (*induction* τ *t rule*: *wff.induct*)
case *wff-Abs* **thus** *?case* **by** (*metis pvar.simps*(*10*,*2*) *pvar-opn wff.wff-Abs*)
qed (*auto intro*: *wff.intros*)

1.9 β-conversion (BKK Section 2)

BKK's β-equality $\equiv_\beta$ (BKK Section 2.1): the congruence closure of the β-redex $(\lambda x.\ b)\ a \to b[a/x]$. In the locally-nameless presentation the redex reduces to $b\langle a\rangle$, and the rule under an abstraction is stated cofinitely, as for typing and local closure.

The relation is *type-indexed*: $s \approx_\varrho t$ holds only for well-formed terms of type ϱ. Carrying the type keeps the symmetric/transitive rules type-preserving (a β-expansion does not otherwise determine the argument's type), which is exactly what the soundness proof of $NK(\beta)$ needs.

inductive *beq* :: *'p tm* ⇒ *ty* ⇒ *'p tm* ⇒ *bool* (‹- ≈_ -› [*51*, *0*, *51*] *50*) **where**
beta: $wff_{\sigma\Rightarrow\varrho}(\mathbf{\Lambda}_\sigma\ b) \Longrightarrow wff_\sigma(a) \Longrightarrow (\mathbf{\Lambda}_\sigma\ b) \cdot a \approx_\varrho b\langle a\rangle$
| *refl*: $wff_\varrho(t) \Longrightarrow t \approx_\varrho t$
| *sym*: $s \approx_\varrho t \Longrightarrow t \approx_\varrho s$
| *trans*: $r \approx_\varrho s \Longrightarrow s \approx_\varrho t \Longrightarrow r \approx_\varrho t$
| *appL*: $s \approx_{\sigma\Rightarrow\varrho} s' \Longrightarrow wff_\sigma(t) \Longrightarrow s \cdot t \approx_\varrho s' \cdot t$
| *appR*: $t \approx_\sigma t' \Longrightarrow wff_{\sigma\Rightarrow\varrho}(s) \Longrightarrow s \cdot t \approx_\varrho s \cdot t'$
| *abs*: *finite L* ⟹ $(\bigwedge x.\ x \notin L \Longrightarrow b\langle x^f{}_\sigma\rangle \approx_\tau b'\langle x^f{}_\sigma\rangle)$
⟹ $\mathbf{\Lambda}_\sigma\ b \approx_{\sigma\Rightarrow\tau} \mathbf{\Lambda}_\sigma\ b'$

β-conversion relates well-formed terms of the stated type.

lemma *beq-wff*: $s \approx_\varrho t \Longrightarrow wff_\varrho(s) \wedge wff_\varrho(t)$
proof (*induction rule*: *beq.induct*)
case *beta* **thus** *?case* **by** (*auto intro*: *wff-App wff-opn*)
next
case *abs* **thus** *?case* **by** (*metis wff-Abs*)
qed (*auto intro*: *wff-App*)

lemma *beq-wffL*: $s \approx_\varrho t \Longrightarrow wff_\varrho(s)$ **and** *beq-wffR*: $s \approx_\varrho t \Longrightarrow wff_\varrho(t)$
using *beq-wff* **by** *blast+*

β-conversion is stable under parameter renaming.

lemma *beq-rename*[*intro*!]: $s \approx_\tau t \Longrightarrow prn\ \pi\ s \approx_\tau prn\ \pi\ t$
proof (*induction rule*: *beq.induct*)
case *beta* **thus** *?case* **by** *simp* (*metis beq.beta prn-opn tm.simps*(*128*) *wff-prn*)
next case *refl* **thus** *?case* **using** *beq.refl wff-prn* **by** *blast*
next case *sym* **thus** *?case* **using** *beq.sym* **by** *blast*
next case *trans* **thus** *?case* **using** *beq.trans* **by** *blast*
next case *appL* **thus** *?case* **using** *beq.appL* **by** *force*
next case *appR* **thus** *?case* **using** *beq.appR* **by** *force*

next case *abs* **thus** *?case* **by** (*metis* (*mono-tags*, *lifting*) *beq.abs prn-opn tm.simps*(*120*,*128*))
qed

1.10 The defined logical layer (BKK Section 2.2)

Following BKK (BKK Section 2.2), everything beyond the primitive constants of the signature — $\neg$, $\vee$, Pi_σ, the primitive equality $Eq\ \sigma$ and the description operators $Iota\ \sigma$ (above) — is defined. The universal quantifier is BKK's shorthand $\forall\, X_\sigma.\ A \equiv \mathbf{\Pi}_\sigma(\lambda X_\sigma.\ A)$ and implication their $A \supset B \equiv (\neg A) \vee B$. For falsity we deviate mildly from BKK, who use $F_\mathrm{o} \equiv \neg\forall\, P_\mathrm{o}.\ P \vee \neg P$ (BKK Lemma 3.43, footnote 11): our $\bot$ is $\forall\, X_\mathrm{o}.\ X$ and $\top \equiv \neg\bot$. The two choices are interderivable in NK_β (cf. BKK Remark 7.2) and both falsa are unsatisfied in every Σ-model, so every use below is invariant under the exchange. Leibniz equality — always expressible, alongside the primitive $Eq\ \alpha$ of the signature — is BKK's Leibniz combinator $Q_\alpha \equiv \lambda X_\alpha\ Y_\alpha.\ \forall\, P_{\alpha\Rightarrow\mathrm{o}}.\ P\ X \supset P\ Y$ (BKK Section 2.2, the Leibniz formula for equality). Because bound variables are de Bruijn indices, $Leib\ \alpha$ is a genuinely *closed* term: BKK's reserved bound names X, Y, P are simply the indices *2*, *1*, *0*.

definition *Forall* :: *ty* ⇒ *'p tm* ⇒ *'p tm* (‹**Π**_ -› [*0*, *200*] *200*) **where**
  $\mathbf{\Pi}_\sigma\ b = (Pi\ \sigma) \cdot (\mathbf{\Lambda}_\sigma\ b)$
definition *FalseB* :: *'p tm* (‹⊥›) **where** $\bot = \mathbf{\Pi}_\mathrm{o}\ (Bnd\ 0)$
definition *TrueB* :: *'p tm* (‹⊤›) **where** $\top = \neg\ \bot$
definition *ImpB* :: *'p tm* ⇒ *'p tm* ⇒ *'p tm* (**infixr** ‹⊃› *60*) **where**
  $\varphi \supset \psi = (\neg\ \varphi) \vee \psi$
definition *Leib* :: *ty* ⇒ *'p tm* **where**
  $Leib\ \alpha = \mathbf{\Lambda}_\alpha\ (Abs\ \alpha\ (\mathbf{\Pi}_{\alpha\Rightarrow\mathrm{o}}\ ((Bnd\ 0) \cdot (Bnd\ 2) \supset (Bnd\ 0) \cdot (Bnd\ 1))))$
abbreviation *LeibA* :: *'p tm* ⇒ *ty* ⇒ *'p tm* ⇒ *'p tm* (‹- ≐_ -› [*66*,*0*,*66*] *65*) **where**
  $A \doteq_\alpha B \equiv ((Leib\ \alpha) \cdot A) \cdot B$

Primitive equality applied (BKK's optional $=_\alpha \in \Sigma_{\alpha\to\alpha\to\mathrm{o}}$, BKK Remark 7.9).

abbreviation *PEqA* :: *'p tm* ⇒ *ty* ⇒ *'p tm* ⇒ *'p tm* (‹- =_ -› [*66*,*0*,*66*] *65*) **where**
  $A =_\alpha B \equiv ((Eq\ \alpha) \cdot A) \cdot B$
lemma *wff-PEq* [*intro*]: $wff_\alpha(A) \Longrightarrow wff_\alpha(B) \Longrightarrow wff_\mathrm{o}(A =_\alpha B)$
  by (*rule wff-App*[*OF wff-App*[*OF wff-Eq*]])

1.11 Opening distributes over the defined layer

lemma *opn-Forall* [*simp*]: $opn\ k\ u\ (\mathbf{\Pi}_\sigma\ b) = \mathbf{\Pi}_\sigma\ (opn\ (Suc\ k)\ u\ b)$
  by (*simp add*: *Forall-def*)
lemma *opn-ImpB* [*simp*]: $opn\ k\ u\ (\varphi \supset \psi) = (opn\ k\ u\ \varphi) \supset (opn\ k\ u\ \psi)$
  by (*simp add*: *ImpB-def*)
lemma *opn-FalseB* [*simp*]: $opn\ k\ u\ (\bot :: 'p\ tm) = \bot$
  by (*simp add*: *FalseB-def*)
lemma *opn-Leib* [*simp*]: $opn\ k\ u\ (Leib\ \alpha :: 'p\ tm) = Leib\ \alpha$
  by (*simp add*: *Leib-def*)

1.12 A convenient abstraction-typing rule

For the closed defined terms, opening the body with *any* free variable is well-typed, so the cofinite side-condition collapses to a universal one.

lemma *wff-AbsI*: $(\bigwedge x.\ \mathit{wff}_\tau(b\langle x^f{}_\sigma\rangle)) \Longrightarrow \mathit{wff}_{\sigma\Rightarrow\tau}(\mathbf{\Lambda}_\sigma\ b)$
by (*rule wff-Abs*[*of* {}]) *auto*

1.13 Typing of the defined layer

lemma *wff-Forall* [*intro*]: $(\bigwedge x.\ \mathit{wff}_{\mathrm{o}}(b\langle x^f{}_\sigma\rangle)) \Longrightarrow \mathit{wff}_{\mathrm{o}}(\mathbf{\Pi}_\sigma\ b)$
by (*metis Forall-def wff-AbsI wff-App wff-Pi*)
lemma *wff-Not* [*intro*]: $\mathit{wff}_{\mathrm{o}}(\varphi) \Longrightarrow \mathit{wff}_{\mathrm{o}}(\neg\ \varphi)$
by (*rule wff-App*[*OF wff-Neg*])
lemma *wff-Or* [*intro*]: $\mathit{wff}_{\mathrm{o}}(\varphi) \Longrightarrow \mathit{wff}_{\mathrm{o}}(\psi) \Longrightarrow \mathit{wff}_{\mathrm{o}}(\varphi \vee \psi)$
by (*rule wff-App*[*OF wff-App*[*OF wff-Dis*]])
lemma *wff-ImpB* [*intro*]: $\mathit{wff}_{\mathrm{o}}(\varphi) \Longrightarrow \mathit{wff}_{\mathrm{o}}(\psi) \Longrightarrow \mathit{wff}_{\mathrm{o}}(\varphi \supset \psi)$
by (*metis ImpB-def wff-Not wff-Or*)
lemma *wff-FalseB* [*intro*, *simp*]: $\mathit{wff}_{\mathrm{o}}(\bot)$ **unfolding** *FalseB-def*
by (*rule wff-Forall*) (*simp add*: *wff-Fre*)
lemma *wff-TrueB* [*intro*, *simp*]: $\mathit{wff}_{\mathrm{o}}(\top)$ **unfolding** *TrueB-def*
by (*rule wff-Not*[*OF wff-FalseB*])
lemma *wff-App-FreFre* [*intro*]: $\mathit{wff}_{\mathrm{o}}((p^f{}_\sigma \Rightarrow \mathrm{o}) \cdot (x^f{}_\sigma))$
by (*rule wff-App*[*OF wff-Fre wff-Fre*])
lemma *wff-Leib* [*intro*, *simp*]: $\mathit{wff}_{\alpha\Rightarrow\alpha\Rightarrow\mathrm{o}}(\mathit{Leib}\ \alpha)$
by (*auto simp*: *Leib-def del*: *wff-Forall wff-ImpB wff-App-FreFre*
intro!: *wff-AbsI wff-Forall wff-ImpB wff-App-FreFre*)
lemma *wff-LeibE* [*intro*]: $\mathit{wff}_\alpha(A) \Longrightarrow \mathit{wff}_\alpha(B) \Longrightarrow \mathit{wff}_{\mathrm{o}}(A \doteq_\alpha B)$
by (*rule wff-App*[*OF wff-App*[*OF wff-Leib*]])

1.14 Free variables and local closure of the defined layer

lemma *fvs-defs* [*simp*]: *fvs* $(\mathbf{\Pi}_\sigma\ b) = \mathit{fvs}\ b$ *fvs* $(\bot :: 'p\ tm) = \{\}$ *fvs* $(\top :: 'p\ tm) = \{\}$
fvs $(\varphi \supset \psi) = \mathit{fvs}\ \varphi \cup \mathit{fvs}\ \psi$ *fvs* $(\mathit{Leib}\ \alpha :: 'p\ tm) = \{\}$
by (*simp-all add*: *Forall-def FalseB-def TrueB-def ImpB-def Leib-def*)

lemma *pars-defs* [*simp*]: *pars* $(\mathbf{\Pi}_\sigma\ b) = \mathit{pars}\ b$ *pars* $(\bot :: 'p\ tm) = \{\}$
pars $(\top :: 'p\ tm) = \{\}$ *pars* $(\varphi \supset \psi) = \mathit{pars}\ \varphi \cup \mathit{pars}\ \psi$ *pars* $(\mathit{Leib}\ \alpha :: 'p\ tm) = \{\}$
by (*simp-all add*: *Forall-def FalseB-def TrueB-def ImpB-def Leib-def*)

1.15 Parameter renaming distributes over the defined layer

lemma *prn-Forall* [*simp*]: *prn* ϱ $(\mathbf{\Pi}_\sigma\ b) = \mathbf{\Pi}_\sigma$ $(\mathit{prn}\ \varrho\ b)$
by (*simp add*: *Forall-def*)
lemma *prn-ImpB* [*simp*]: *prn* ϱ $(\varphi \supset \psi) = (\mathit{prn}\ \varrho\ \varphi) \supset (\mathit{prn}\ \varrho\ \psi)$
by (*simp add*: *ImpB-def*)
lemma *prn-FalseB* [*simp*]: *prn* ϱ $(\bot :: 'p\ tm) = \bot$
by (*simp add*: *FalseB-def*)
lemma *prn-Leib* [*simp*]: *prn* ϱ $(\mathit{Leib}\ \alpha :: 'p\ tm) = \mathit{Leib}\ \alpha$

by (*simp add*: *Leib-def*)

1.16 The defined existential quantifier

abbreviation *ExistsB* :: *ty* ⇒ *'p tm* ⇒ *'p tm* (‹∃ _ -› [*0*, *200*] *200*) **where**
∃ $_\sigma$ *b* ≡ ¬ (**Π**$_\sigma$ (¬ *b*))

1.17 Named binders for the defined quantifiers

Named-binder input syntax for the defined quantifiers: *clos k x σ t* abstracts the free variable $x^f{}_\sigma$ to the de Bruijn index *k* (the converse of *opn*), so that ∃ x_σ. *φ* and **Π**x_σ. *φ* bind an ordinary named variable; the locally-nameless representation is recovered by computation (the *-eq* lemmas below).

primrec *clos* :: *nat* ⇒ *nat* ⇒ *ty* ⇒ *'p tm* ⇒ *'p tm* **where**
clos k x σ (*Bnd i*) = *Bnd i*
| *clos k x σ* ($n^f{}_\tau$) = (*if n* = *x* ∧ *τ* = *σ then Bnd k else* $n^f{}_\tau$)
| *clos k x σ* ($p^p{}_\tau$) = $p^p{}_\tau$
| *clos k x σ Neg* = *Neg*
| *clos k x σ Dis* = *Dis*
| *clos k x σ* (*Pi τ*) = *Pi τ*
| *clos k x σ* (*Iota τ*) = *Iota τ*
| *clos k x σ* (*Eq τ*) = *Eq τ*
| *clos k x σ* (*s* · *t*) = (*clos k x σ s*) · (*clos k x σ t*)
| *clos k x σ* (**Λ**$_\tau$ *b*) = **Λ**$_\tau$ (*clos* (*Suc k*) *x σ b*)

lemma *clos-Forall* [*simp*]: *clos k x σ* (**Π**$_\tau$ *b*) = **Π**$_\tau$ (*clos* (*Suc k*) *x σ b*)
by (*simp add*: *Forall-def*)
lemma *clos-ImpB* [*simp*]: *clos k x σ* (*φ* ⊃ *ψ*) = (*clos k x σ φ*) ⊃ (*clos k x σ ψ*)
by (*simp add*: *ImpB-def*)
lemma *clos-Leib* [*simp*]: *clos k x σ* (*Leib α* :: *'p tm*) = *Leib α*
by (*simp add*: *Leib-def*)
definition *ExN* :: *nat* ⇒ *ty* ⇒ *'p tm* ⇒ *'p tm* (‹∃ -_. -› [*1000*, *0*, *61*] *61*) **where**
∃ x_σ. *b* = ∃ $_\sigma$ (*clos 0 x σ b*)
definition *AllN* :: *nat* ⇒ *ty* ⇒ *'p tm* ⇒ *'p tm* (‹**Π**-_. -› [*1000*, *0*, *61*] *61*) **where**
Πx_σ. *b* = **Π**$_\sigma$ (*clos 0 x σ b*)
definition *LamN* :: *nat* ⇒ *ty* ⇒ *'p tm* ⇒ *'p tm* (‹**Λ**-_. -› [*1000*, *0*, *61*] *61*) **where**
Λx_σ. *b* = **Λ**$_\sigma$ (*clos 0 x σ b*)

Fixed variable names for readable named-binder statements: the script letters $\mathcal{G}$, $\mathcal{F}$, $\mathcal{X}$, $\mathcal{I}$, $\mathcal{H}$ name the (numeric) free variables *0*, …, *4*.

definition $\mathcal{G}$ :: *nat* **where** $\mathcal{G}$ = *0*
definition $\mathcal{F}$ :: *nat* **where** $\mathcal{F}$ = *1*
definition $\mathcal{X}$ :: *nat* **where** $\mathcal{X}$ = *2*
definition $\mathcal{I}$ :: *nat* **where** $\mathcal{I}$ = *3*
definition $\mathcal{H}$ :: *nat* **where** $\mathcal{H}$ = *4*

Stronger size-based induction.

lemma *size-induct*[*case-names Bnd Fre Par Neg Dis Pi Iota Eq App Abs*]:

assumes ‹$\bigwedge x.\ P\ (Bnd\ x)$›
and ‹$\bigwedge x\ \alpha.\ P\ x^f{}_\alpha$›
and ‹$\bigwedge x\ \alpha.\ P\ x^p{}_\alpha$›
and ‹$P\ Neg$›
and ‹$P\ Dis$›
and ‹$\bigwedge \alpha.\ P\ (Pi\ \alpha)$›
and ‹$\bigwedge \alpha.\ P\ (Iota\ \alpha)$›
and ‹$\bigwedge \alpha.\ P\ (Eq\ \alpha)$›
and ‹$\bigwedge A\ B.\ (\bigwedge C.\ size\ C \leq size\ A \Longrightarrow P\ C) \Longrightarrow (\bigwedge C.\ size\ C \leq size\ B \Longrightarrow P\ C) \Longrightarrow P\ (A \cdot B)$›
and ‹$\bigwedge \alpha\ A.\ (\bigwedge C.\ size\ C \leq size\ A \Longrightarrow P\ C) \Longrightarrow P\ (\mathbf{\Lambda}_\alpha\ A)$›
shows ‹$P\ x$›
using *assms* **proof** (*induct x rule*: *measure-induct-rule*[**where** $f = size$])
case (*less x*) **thus** *?case* **by** (*induct x*; *auto*)
qed

2 Semantics: applicative structures, Henkin models and standard models

The semantics in BKK's own layered terminology, in their order: *applicative structures* $(D, @)$ (BKK Definition 3.1), Σ-*evaluations* $J = (D, @, E)$ (BKK Definition 3.18) with an *abstract* evaluation function subject to BKK's four conditions, Σ-*valuations* and Σ-*models* $M = (D, @, E, \upsilon)$ (BKK Definitions 3.40 and 3.41), and the model class $\mathcal{M}_{\beta fb}$ (BKK Definition 3.49, the Σ-Henkin models of Definition 3.50). The signature Σ is a static parameter of the whole development: the logical constants fixed by the term datatype plus the typed parameters drawn from $'p$, exactly as BKK fix Σ at the start of their Section 3.

After the abstract notions, the recursive denotation $(\!|t|\!)_\xi$ is introduced as the *canonical construction* of an evaluation function over frame-like structures (BKK's Σ-evaluations over frames); it is the vehicle for building concrete models (the standard models here, and the term model of the completeness proof in Section 3).

2.1 Assignment update

BKK's assignment update $\varphi,[a/X]$ (BKK Definition 3.17), written in function-update style with the variable's type subscripted: $\xi(x_\sigma := d)$.

definition *upd* :: $(nat \Rightarrow ty \Rightarrow 'u) \Rightarrow nat \Rightarrow ty \Rightarrow 'u \Rightarrow (nat \Rightarrow ty \Rightarrow 'u)$
($\langle$-'(-_ := -')› [*1000*, *0*, *0*, *0*] *1000*) **where**
$\xi(x_\sigma := d) \equiv \lambda n\ \tau.$ *if* $n = x \wedge \tau = \sigma$ *then* d *else* $\xi\ n\ \tau$

lemma *upd-same* [*simp*]: $\xi(x_\sigma := d)\ x\ \sigma = d$ **by** (*simp add*: *upd-def*)
lemma *upd-comm*: $(x, \sigma) \neq (w, \tau) \Longrightarrow (\xi(x_\sigma := d))(w_\tau := e) = (\xi(w_\tau := e))(x_\sigma := d)$

by (*auto simp*: *upd-def fun-eq-iff*)

2.2 Applicative structures (BKK Definition 3.1)

A typed collection of non-empty domains with an application operator. BKK's typing discipline @ : $D_{\alpha\to\beta} \times D_\alpha \to D_\beta$ becomes a closure condition on the one abstract operator. BKK's *frames* (Definition 3.4, $D_{\alpha\to\beta} \subseteq F(D_\alpha; D_\beta)$) are a set-theoretic notion with no direct analogue over an abstract carrier; by BKK Remark 3.6 every frame is functional, and it is *functionality* (BKK Definition 3.5; named property f in Definition 3.46) that all mathematical arguments consume, so the model class below is delineated by functionality.

locale *app-struct* = **fixes** *Dm* :: *ty* ⇒ *'u* ⇒ *bool* **and** *Ap* :: *'u* ⇒ *'u* ⇒ *'u*
  assumes *as-nonempty*: ∃ *a*. *Dm α a* **and** *as-appTy*: *Dm* (*α* ⇒ *β*) *f* ⟹ *Dm α a* ⟹ *Dm β* (*Ap f a*)
begin

Variable assignments into the structure (BKK Definition 3.17); the update $\xi(x_\sigma := d)$ is BKK's $\varphi,[d/X]$.

definition *asg* :: (*nat* ⇒ *ty* ⇒ *'u*) ⇒ *bool* **where** *asg ξ* ≡ ∀ *n τ*. *Dm τ* (*ξ n τ*)

lemma *asg-upd*: *asg ξ* ⟹ *Dm σ d* ⟹ *asg* (*ξ*(*x*$_\sigma$:= *d*))
  by (*auto simp*: *asg-def upd-def*)

Functionality: property f of BKK Definition 3.5.

definition *functional* :: *bool* **where**
  functional ≡ ∀ *α β f g*. *Dm* (*α* ⇒ *β*) *f* ⟶ *Dm* (*α* ⇒ *β*) *g* ⟶
    (∀ *a*. *Dm α a* ⟶ *Ap f a* = *Ap g a*) ⟶ *f* = *g*

end

2.3 Σ-evaluations (BKK Definition 3.18)

An evaluation function E maps assignments to typed functions from well-formed formulae into the domains, subject to BKK's four conditions: (1) it extends the assignment on variables, (2) it is homomorphic for application, (3) it depends only on the assignment's values at the free variables (coincidence), and (4) it respects β-conversion (BKK state this via β-normal forms; over our typed β-equality $\approx_\tau$ of Section 1 the two formulations coincide, cf. BKK Remark 3.19). In addition E is typed: well-formed formulae of type τ denote in D_τ.

locale *sigma-eval* = *app-struct Dm Ap* **for** *Dm* :: *ty* ⇒ *'u* ⇒ *bool* **and** *Ap* :: *'u* ⇒ *'u* ⇒ *'u* +
  fixes *Ee* :: (*nat* ⇒ *ty* ⇒ *'u*) ⇒ *'p tm* ⇒ *'u*
  assumes *ev-type*: *wff*$_\tau$(*A*) ⟹ *asg ξ* ⟹ *Dm τ* (*Ee ξ A*)
    and *ev-var*: *asg ξ* ⟹ *Ee ξ* (*n*$^f{}_\sigma$) = *ξ n σ*

and *ev-app*: $wff_{\sigma \Rightarrow \tau}(F) \Longrightarrow wff_{\sigma}(A) \Longrightarrow asg\ \xi \Longrightarrow Ee\ \xi\ (F \cdot A) = Ap\ (Ee\ \xi\ F)\ (Ee\ \xi\ A)$
and *ev-coin*: $wff_{\tau}(A) \Longrightarrow asg\ \xi \Longrightarrow asg\ \xi' \Longrightarrow (\bigwedge n\ \sigma.\ (n, \sigma) \in occ\ A \Longrightarrow \xi\ n\ \sigma = \xi'\ n\ \sigma) \Longrightarrow Ee\ \xi\ A = Ee\ \xi'\ A$
and *ev-beta*: $A \approx_{\tau} B \Longrightarrow asg\ \xi \Longrightarrow Ee\ \xi\ A = Ee\ \xi\ B$
begin

The derived β-application law: the denotation of an abstraction is determined applicatively by the openings of its body (from conditions (1), (2), (4) and coincidence; the vehicle for all abstraction reasoning below).

lemma *ev-abs-app*:
assumes *wb*: $wff_{\sigma \Rightarrow \tau}(\mathbf{\Lambda}_{\sigma}\ b)$ **and** *xi*: $asg\ \xi$ **and** *x*: $x \notin fvs\ b$ **and** *d*: $Dm\ \sigma\ d$
shows $Ap\ (Ee\ \xi\ (\mathbf{\Lambda}_{\sigma}\ b))\ d = Ee\ (\xi(x_{\sigma} := d))\ (b\langle x^{f}{}_{\sigma}\rangle)$
by (*smt* (*verit*, *del-insts*) *asg-upd beta d fvs.simps*(*10*) *fvs-eq-fst-occ image-eqI prod.sel*(*1*)
sigma-eval.ev-app sigma-eval.ev-beta sigma-eval.ev-coin sigma-eval.ev-var sigma-eval-axioms
upd-def wb wff-Fre x xi)

end

2.4 Σ-valuations and Σ-models (BKK Definitions 3.40 and 3.41)

A Σ-valuation is a (total) function $\upsilon : D_{\mathrm{o}} \to \{T, F\}$ — rendered as a HOL predicate — satisfying the properties $L_{\neg}(E(\neg))$, $L_{\vee}(E(\vee))$ and $L^{\alpha}{}_{\forall}(E(\Pi_{\alpha}))$ of BKK's Figure 2. A Σ-evaluation together with such a valuation is a Σ-model. Following BKK Definition 3.41 (and Remark 3.42) we include primitive equality $L^{\alpha}{}_{=}(E(=_{\alpha}))$, and — extending BKK Definition 3.41, whose Σ-models have no description operator — a description property for $E(\iota_{\alpha})$, matching $NK(\iota)$. Since the logical constants are closed, their denotations are assignment- independent (coincidence); we fix a canonical assignment to name them.

locale *sigma-model* = *sigma-eval Dm Ap Ee* **for**
Dm :: $ty \Rightarrow 'u \Rightarrow bool$ **and** *Ap* :: $'u \Rightarrow 'u \Rightarrow 'u$ **and** *Ee* :: $(nat \Rightarrow ty \Rightarrow 'u) \Rightarrow 'p\ tm \Rightarrow 'u$ +
fixes *vl* :: $'u \Rightarrow bool$
assumes *vl-neg*: $asg\ \xi \Longrightarrow Dm\ \mathrm{o}\ a \Longrightarrow vl\ (Ap\ (Ee\ \xi\ Neg)\ a) \longleftrightarrow \neg\ vl\ a$
and *vl-dis*: $asg\ \xi \Longrightarrow Dm\ \mathrm{o}\ a \Longrightarrow Dm\ \mathrm{o}\ b \Longrightarrow vl\ (Ap\ (Ap\ (Ee\ \xi\ Dis)\ a)\ b) \longleftrightarrow vl\ a \vee vl\ b$
and *vl-pi*: $asg\ \xi \Longrightarrow Dm\ (\sigma \Rightarrow \mathrm{o})\ f \Longrightarrow vl\ (Ap\ (Ee\ \xi\ (Pi\ \sigma))\ f) \longleftrightarrow (\forall d.\ Dm\ \sigma\ d \longrightarrow vl\ (Ap\ f\ d))$
and *vl-eq*: $asg\ \xi \Longrightarrow Dm\ \sigma\ a \Longrightarrow Dm\ \sigma\ b \Longrightarrow vl\ (Ap\ (Ap\ (Ee\ \xi\ (Eq\ \sigma))\ a)\ b) \longleftrightarrow a = b$
and *vl-iota*: $asg\ \xi \Longrightarrow Dm\ (\sigma \Rightarrow \mathrm{o})\ f \Longrightarrow Dm\ \sigma\ a \Longrightarrow (\bigwedge b.\ Dm\ \sigma\ b \Longrightarrow vl\ (Ap\ f\ b) \longleftrightarrow b = a)$

$\Longrightarrow$ *Ap* (*Ee* ξ (*Iota* σ)) $f = a$

begin

Satisfaction and validity (BKK Definition 3.41): $M \models_\varphi A$ iff $\upsilon(E_\varphi(A)) \equiv T$.

definition *satisfies* :: (*nat* $\Rightarrow$ *ty* $\Rightarrow$ *'u*) $\Rightarrow$ *'p tm* $\Rightarrow$ *bool* (‹⊨_ -› [*0*, *40*] *40*) **where**
$\models_\xi A \equiv vl\ (Ee\ \xi\ A)$
definition *valid* :: *'p tm* $\Rightarrow$ *bool* (‹⊨ -› [*40*] *40*) **where**
$\models A \equiv \forall \xi.\ asg\ \xi \longrightarrow \models_\xi A$

end

2.5 The model class $\mathcal{M}_{\beta f b}$ (BKK Definition 3.49)

BKK's completeness class for *NK* is $\mathcal{M}_{\beta f b}$: Σ-models satisfying properties q, f and b (with primitive equality, property q holds automatically, BKK Definition 3.49 — the q-witness at type α is the denotation $E(=_\alpha)$, cf. the satisfaction lemma for Leibniz equality; with property b the valuation is two-valued on D_o). This class coincides with the Σ-Henkin models of BKK Definition 3.50 up to isomorphism (BKK Lemma 3.67 and Theorem 3.68).

locale *bkk-model* = *sigma-model Dm Ap Ee vl*
for *Dm* :: *ty* $\Rightarrow$ *'u* $\Rightarrow$ *bool* **and** *Ap* :: *'u* $\Rightarrow$ *'u* $\Rightarrow$ *'u*
and *Ee* :: (*nat* $\Rightarrow$ *ty* $\Rightarrow$ *'u*) $\Rightarrow$ *'p tm* $\Rightarrow$ *'u* **and** *vl* :: *'u* $\Rightarrow$ *bool* +
assumes *prop-f*: *functional* **and** *prop-b*: *Dm* o *a* $\Longrightarrow$ *Dm* o *b* $\Longrightarrow$ *vl a* $\longleftrightarrow$ *vl b* $\Longrightarrow$ *a* = *b*

2.6 Truth values in a Σ-model (BKK Lemma 3.43)

context *sigma-model*
begin

The canonical assignment, from non-emptiness of the domains.

definition *xi0* :: *nat* $\Rightarrow$ *ty* $\Rightarrow$ *'u* **where** *xi0* $\equiv \lambda n\ \tau.$ *SOME a. Dm* τ *a*
lemma *asg-xi0*: *asg xi0* **using** *as-nonempty* **by** (*auto simp*: *asg-def xi0-def intro*: *someI-ex*)

Closed terms evaluate independently of the assignment; the canonical assignment serves as the reference.

lemma *Ee-closed*: $wff_\tau(A) \Longrightarrow occ\ A = \{\} \Longrightarrow asg\ \xi \Longrightarrow Ee\ \xi\ A = Ee\ xi0\ A$
using *ev-coin asg-xi0* **by** *blast*

Evaluating $\bot = \mathbf{\Pi}_o(Bnd\ 0)$: its truth means every boolean object is true.

lemma *vl-FalseB*: **assumes** *xi*: *asg* ξ **shows** *vl* (*Ee* ξ $\bot$) $\longleftrightarrow$ ($\forall$ *d. Dm* o *d* $\longrightarrow$ *vl d*)
proof −
have *wI*: $wff_{o \Rightarrow o}(\mathbf{\Lambda}_o\ (Bnd\ 0) :: 'p\ tm)$
by (*rule wff-AbsI*) (*simp add*: *wff-Fre*)

have *e*: *Ee* ξ ($\bot$ :: $'p$ *tm*) = *Ap* (*Ee* ξ (*Pi* o)) (*Ee* ξ ($\mathbf{\Lambda}_{\mathrm{o}}$ (*Bnd 0*)))
by (*metis FalseB-def Forall-def ev-app wI wff-Pi xi*)
have *b*: *Ap* (*Ee* ξ ($\mathbf{\Lambda}_{\mathrm{o}}$ (*Bnd 0*))) *d* = *d* **if** *d*: *Dm* o *d* **for** *d*
using *asg-upd ev-abs-app ev-var that wI xi* **by** *auto*
show *?thesis* **unfolding** *e* **using** *vl-pi xi ev-type wI xi b* **by** *auto*
qed

BKK Lemma 3.43: $\upsilon(E(\top)) \equiv T$ and $\upsilon(E(\bot)) \equiv F$; in particular D_{o} contains a true and a false object (BKK Remark 3.44).

lemma *vl-TF*: **assumes** *xi*: *asg* ξ **shows** *vl* (*Ee* ξ $\top$) $\wedge$ $\neg$ *vl* (*Ee* ξ $\bot$)
by (*metis* (*mono-tags*, *lifting*) *TrueB-def sigma-eval.ev-app sigma-eval.ev-type sigma-eval-axioms*
vl-FalseB vl-neg wff-FalseB wff-Neg wff-TrueB xi)

end

2.7 Model-relative truth and validity at a carrier

Truth of A in a model $\langle D,@,E,\upsilon\rangle$ under an assignment, and validity over *all* models of the class $\mathcal{M}_{\beta fb}$ at a given value carrier $'u$ — the carrier appears explicitly in the notation $\models('u)$ A.

definition *rel-truth* ::
((*nat* $\Rightarrow$ *ty* $\Rightarrow$ $'u$) $\Rightarrow$ $'p$ *tm* $\Rightarrow$ $'u$) $\Rightarrow$ ($'u$ $\Rightarrow$ *bool*) $\Rightarrow$ (*nat* $\Rightarrow$ *ty* $\Rightarrow$ $'u$) $\Rightarrow$ $'p$ *tm* $\Rightarrow$ *bool*
(‹⟨-,-⟩,- ⊨ -› [*0*, *0*, *0*, *61*] *60*) **where**
⟨*Ee*,*vl*⟩,ξ ⊨ *A* ≡ *vl* (*Ee* ξ *A*)

definition *bkk-valid* :: $'u$ *itself* $\Rightarrow$ $'p$ *tm* $\Rightarrow$ *bool* **where**
bkk-valid u A ≡ $\forall$ *Dm Ap Ee* (*vl* :: $'u$ $\Rightarrow$ *bool*) ξ.
bkk-model Dm Ap Ee vl ⟶ *app-struct.asg Dm* ξ ⟶ ⟨*Ee*,*vl*⟩,ξ ⊨ *A*

syntax *-bkk-valid* :: *type* $\Rightarrow$ *logic* $\Rightarrow$ *logic* (‹⊨'(-') -› [*1000*, *61*] *60*)
syntax-consts *-bkk-valid* == *bkk-valid*
translations *-bkk-valid t A* == *CONST bkk-valid* (*-TYPE t*) *A*

definition *bkk-consequence* :: $'u$ *itself* $\Rightarrow$ $'p$ *tm set* $\Rightarrow$ $'p$ *tm* $\Rightarrow$ *bool* **where**
bkk-consequence u Γ *A* ≡ $\forall$ *Dm Ap Ee* (*vl* :: $'u$ $\Rightarrow$ *bool*) ξ.
bkk-model Dm Ap Ee vl ⟶ *app-struct.asg Dm* ξ ⟶
($\forall$ *B* ∈ Γ . *wff* o *B* $\wedge$ ⟨*Ee*,*vl*⟩,ξ ⊨ *B*) ⟶ ⟨*Ee*,*vl*⟩,ξ ⊨ *A*

syntax *-bkk-consequence* :: *logic* $\Rightarrow$ *type* $\Rightarrow$ *logic* $\Rightarrow$ *logic* (‹- ⊨'(-') -› [*61*, *1000*, *61*] *60*)
syntax-consts *-bkk-consequence* == *bkk-consequence*
translations *-bkk-consequence* Γ *t A* == *CONST bkk-consequence* (*-TYPE t*) Γ *A*

2.8 Σ-evaluations over frames: the canonical construction

A *frame signature* fixes the applicative structure (BKK Definition 3.1) and the denotation objects of the logical constants and parameters. It carries no conditions; the denotation and its purely semantic properties (coincidence, BKK Definition 3.18(3)) live here.

locale *frame-sig* =
  fixes *Dm* :: *ty* ⇒ *'u* ⇒ *bool* (‹𝒟_›)
    and *Ap* :: *'u* ⇒ *'u* ⇒ *'u* (**infixl** ‹@› *200*)
    and *Lm* :: *ty* ⇒ (*'u* ⇒ *'u*) ⇒ *'u* **and** *Tv* :: *'u* **and** *Fv* :: *'u*
    and *Ngv* :: *'u* **and** *Dsv* :: *'u* **and** *Iv* :: *ty* ⇒ *'u*
    and *Ev* :: *ty* ⇒ *'u* **and** *Piv* :: *ty* ⇒ *'u* **and** *Jv* :: *'p* ⇒ *ty* ⇒ *'u*
begin

The denotation $(\!|A|\!)_\xi$ of a term under an assignment ξ — BKK's evaluation function (BKK Definition 3.18).

fun *den* :: *'p tm* ⇒ (*nat* ⇒ *ty* ⇒ *'u*) ⇒ *'u* (‹(|-|)_› [*0*,*0*] *1000*) **where**
    $(\!|Bnd\ i|\!)_\xi$ = *Tv* — junk: no free bound index in a locally closed term
  | $(\!|n^f{}_\sigma|\!)_\xi$ = ξ *n* σ
  | $(\!|p^p{}_\sigma|\!)_\xi$ = *Jv p* σ
  | $(\!|Neg|\!)_\xi$ = *Ngv*
  | $(\!|Dis|\!)_\xi$ = *Dsv*
  | $(\!|Pi\ \sigma|\!)_\xi$ = *Piv* σ
  | $(\!|Iota\ \sigma|\!)_\xi$ = *Iv* σ
  | $(\!|s \cdot t|\!)_\xi$ = $(\!|s|\!)_\xi$ @ $(\!|t|\!)_\xi$
  | $(\!|\mathbf{\Lambda}_\sigma\ b|\!)_\xi$ = *Lm* σ ($\lambda d.\ (\!|b\langle(fresh\ (fvs\ b))^f{}_\sigma\rangle|\!)_{\xi((fresh\ (fvs\ b))_\sigma\ :=\ d)}$)
  | $(\!|Eq\ \sigma|\!)_\xi$ = *Ev* σ

The recursive equations must be kept out of the default simpset: the *Abs* equation unfolds under a λ and would loop.

declare *den.simps*(*8*,*9*) [*simp del*]

Coincidence (BKK Definition 3.18(3)): the denotation depends only on the assignment at the (typed) free occurrences.

lemma *den-coincidence*: ($\bigwedge n\ \tau.\ (n,\ \tau) \in occ\ t \Longrightarrow \xi\ n\ \tau = \xi'\ n\ \tau$) $\Longrightarrow$ $(\!|t|\!)_\xi$ = $(\!|t|\!)_{\xi'}$
proof (*induct t arbitrary*: ξ ξ' *rule*: *size-induct*)
  case (*App s1 t1*)
  hence $(\!|s1|\!)_\xi$ = $(\!|s1|\!)_{\xi'}$ **using** *App* **by** *fastforce*
  moreover have $(\!|t1|\!)_\xi$ = $(\!|t1|\!)_{\xi'}$
    using *App* **by** *fastforce*
  ultimately show *?case*
    by (*simp add*: *den.simps*(*8*))
next
  case (*Abs* σ *b*)
  let *?x* = *fresh* (*fvs b*)
  have *xnb*: *?x* ∉ *fvs b* **by** (*rule fresh-notin*) *simp*

```
  have (|b⟨?x^f σ⟩|)_ξ(?x_σ := d) = (|b⟨?x^f σ⟩|)_ξ'(?x_σ := d) for d
    by (auto intro!: Abs)
       (smt (verit, best) Abs.prems Un-iff occ.simps(10,2) occ-opn prod.inject
                          singleton-iff subset-eq upd-def)
  thus ?case by (simp only: Abs den.simps(9))
qed auto
```

α-invariance: opening with either of two fresh free variables gives the same denotation. This makes the fresh choice built into *den* irrelevant, and is the key to the substitution-value lemma below. The abstraction case renames both internal fresh choices to a common fresh w (inner induction hypothesis), commutes the openings ($[\![?i \neq ?j;\ lc\ ?u;\ lc\ ?v]\!] \Longrightarrow opn\ ?i\ ?u\ (opn\ ?j\ ?v\ ?t) = opn\ ?j\ ?v\ (opn\ ?i\ ?u\ ?t)$), then swaps the two names (outer induction hypothesis).

```
lemma den-rename: x ∉ fvs t ⟹ y ∉ fvs t ⟹ (|opn k (x^f σ) t|)_ξ(x_σ := d)
    = (|opn k (y^f σ) t|)_ξ(y_σ := d)
proof (induction t arbitrary: k ξ x y σ d rule: size-induct)
  case (App s1 t1)
  hence (|opn k (x^f σ) s1|)_ξ(x_σ := d) = (|opn k (y^f σ) s1|)_ξ(y_σ := d) and
        (|opn k (x^f σ) t1|)_ξ(x_σ := d) = (|opn k (y^f σ) t1|)_ξ(y_σ := d)
    by auto
  thus ?case by (simp only: App opn.simps den.simps(8))
next
  case (Abs τ c)
  define w where w = fresh (fvs c ∪ {x, y})
  have wc: w ∉ fvs c and wx: w ≠ x and wy: w ≠ y
    using fresh-notin[of fvs c ∪ {x, y}] unfolding w-def by auto
  have xc: x ∉ fvs c and yc: y ∉ fvs c using Abs by auto
  let ?px = fresh (fvs (opn (Suc k) (x^f σ) c))
  let ?py = fresh (fvs (opn (Suc k) (y^f σ) c))
  have pxf: ?px ∉ fvs (opn (Suc k) (x^f σ) c)
    by (rule fresh-notin) simp
  have pyf: ?py ∉ fvs (opn (Suc k) (y^f σ) c)
    by (rule fresh-notin) simp
  have wnx: w ∉ fvs (opn (Suc k) (x^f σ) c)
    using wc wx fvs-opn[of Suc k x^f σ c] by auto
  have wny: w ∉ fvs (opn (Suc k) (y^f σ) c)
    using wc wy fvs-opn[of Suc k y^f σ c] by auto
  have cw: (opn (Suc k) (z^f σ) c)⟨w^f τ⟩ = opn (Suc k) (z^f σ) (c⟨w^f τ⟩) for z
    by (rule opn-opn-comm) auto
  have (|(opn (Suc k) (x^f σ) c)⟨?px^f τ⟩|)_(ξ(x_σ := d))(?px_τ := e)
      = (|(opn (Suc k) (y^f σ) c)⟨?py^f τ⟩|)_(ξ(y_σ := d))(?py_τ := e)
    for e
  proof −
    have (|(opn (Suc k) (x^f σ)
      c)⟨?px^f τ⟩|)_(ξ(x_σ := d))(?px_τ := e)
      = (|(opn (Suc k) (x^f σ) c)⟨w^f τ⟩|)_(ξ(x_σ := d))(w_τ := e)
```

using *Abs pxf wnx* **by** (*metis nle-le size-opn-Fre*)
also have $\ldots = (\!|opn\ (Suc\ k)\ (x^f\sigma)\ (c\langle w^f\tau\rangle)|\!)_{(\xi(w_\tau := e))(x_\sigma := d)}$
using *wx* **by** (*simp add*: *cw upd-comm*)
also have $\ldots = (\!|opn\ (Suc\ k)\ (y^f\sigma)\ (c\langle w^f\tau\rangle)|\!)_{(\xi(w_\tau := e))(y_\sigma := d)}$
using *Abs xc yc wc wx wy fvs-opn*[*of 0* $w^f\tau$ *c*]
by (*smt* (*verit*, *best*) *Un-insert-right dual-order.refl fvs.simps*(2) *insertE size-opn-Fre*
subset-eq sup-bot.right-neutral)
also have $\ldots = (\!|(opn\ (Suc\ k)\ (y^f\sigma)\ c)\langle w^f\tau\rangle|\!)_{(\xi(y_\sigma := d))(w_\tau := e)}$
using *wy* **by** (*simp add*: *cw upd-comm*)
also have $\ldots = (\!|(opn\ (Suc\ k)\ (y^f\sigma)\ c)\langle ?py^f\tau\rangle|\!)_{(\xi(y_\sigma := d))(?py_\tau := e)}$
using *Abs pyf wny* **by** (*metis order-refl size-opn-Fre*)
finally show *?thesis*.
qed
thus *?case* **using** *Abs* **by** (*simp add*: *den.simps*(9))
qed(*auto simp*: *upd-def*)

Any sufficiently fresh variable may be used to compute an abstraction's denotation.

lemma *den-Abs*: $x \notin fvs\ b \Longrightarrow (\!|\mathbf{\Lambda}_\sigma\ b|\!)_\xi = Lm\ \sigma\ (\lambda d.\ (\!|b\langle x^f\sigma\rangle|\!)_{\xi(x_\sigma := d)})$
by (*metis* (*no-types*, *lifting*) *ext den.simps*(9) *finite-fvs frame-sig.den-rename fresh-notin*)

The substitution-value lemma (BKK Lemma 3.20).

lemma *den-fsub*: *lc u* $\Longrightarrow (\!|fsub\ x\ \sigma\ u\ t|\!)_\xi = (\!|t|\!)_{\xi(x_\sigma := (\!|u|\!)_\xi)}$
proof (*induction t arbitrary*: ξ *rule*: *size-induct*)
case (*Fre n* ϱ) **thus** *?case* **by** (*auto simp*: *upd-def*)
next
case (*App s1 t1*)
have $(\!|fsub\ x\ \sigma\ u\ s1|\!)_\xi = (\!|s1|\!)_{\xi(x_\sigma := (\!|u|\!)_\xi)}$ **and**
$(\!|fsub\ x\ \sigma\ u\ t1|\!)_\xi = (\!|t1|\!)_{\xi(x_\sigma := (\!|u|\!)_\xi)}$
using *App* **by** *auto*
thus *?case* **by** (*simp only*: *App fsub.simps den.simps*(8))
next
case (*Abs* τ *b*)
define *w* **where** $w = fresh\ (fvs\ b \cup fvs\ u \cup \{x\})$
have *wb*: $w \notin fvs\ b$ **and** *wu*: $w \notin fvs\ u$ **and** *wx*: $w \neq x$
using *fresh-notin*[*of* $fvs\ b \cup fvs\ u \cup \{x\}$] **unfolding** *w-def* **by** *auto*
have *wfsb*: $w \notin fvs\ (fsub\ x\ \sigma\ u\ b)$
using *wb wu fvs-fsub*[*of x* σ *u b*] **by** *auto*
have $(\!|fsub\ x\ \sigma\ u\ (\mathbf{\Lambda}_\tau\ b)|\!)_\xi = Lm\ \tau\ (\lambda d.\ (\!|(fsub\ x\ \sigma\ u\ b)\langle w^f\tau\rangle|\!)_{\xi(w_\tau := d)})$
by (*simp only*: *fsub.simps den-Abs*[*OF wfsb*])
also have $\ldots = Lm\ \tau\ (\lambda d.\ (\!|fsub\ x\ \sigma\ u\ (b\langle w^f\tau\rangle)|\!)_{\xi(w_\tau := d)})$
using *Abs wx* **by** (*simp add*: *fsub-opn*)
also have $\ldots = Lm\ \tau\ (\lambda d.\ (\!|b\langle w^f\tau\rangle|\!)_{(\xi(w_\tau := d))(x_\sigma := (\!|u|\!)_{\xi(w_\tau := d)})})$

```
    using Abs by auto
  also have ... = Lm τ (λd. (|b⟨w^f τ⟩|)_(ξ(x_σ := (|u|)_ξ))(w_τ := d))
  proof (rule arg-cong[where f = Lm τ], rule ext)
    fix d
    have (|u|)_(ξ(w_τ := d)) = (|u|)_ξ
      using den-coincidence wu fvs-eq-fst-occ
      by (smt (verit, ccfv-threshold) fst-conv image-eqI upd-def)
    thus (|b⟨w^f τ⟩|)_(ξ(w_τ := d))(x_σ := (|u|)_(ξ(w_τ := d)))
        = (|b⟨w^f τ⟩|)_(ξ(x_σ := (|u|)_ξ))(w_τ := d)
      using wx by (simp add: upd-comm)
  qed
  also have ... = (|Λ_τ b|)_(ξ(x_σ := (|u|)_ξ))
    by (rule den-Abs[OF wb, symmetric])
  finally show ?case unfolding Abs .
qed auto
```

The β form of the substitution-value lemma (BKK Lemma 3.20): opening an abstraction body with a (locally closed) argument u is computed by evaluating the body under the assignment updated with the denotation of u. This is the semantic counterpart of β-reduction and the workhorse of soundness.

```
lemma den-beta: assumes u: lc u and x: x ∉ fvs b
  shows (|b⟨u⟩|)_ξ = (|b⟨x^f σ⟩|)_ξ(x_σ := (|u|)_ξ)
  by (metis den-fsub fsub-intro u x)

end
```

Satisfaction of the connectives (the Σ-valuation conditions of BKK Definition 3.41 and Figure 2; for the *defined* connectives cf. BKK Remark 3.47 and Lemma 3.48): model-theoretic facts, stated here so that the calculus can use them for soundness.

```
context sigma-model
begin

lemma sat-Neg: assumes wff_o(A) and asg ξ
  shows vl (Ee ξ (¬ A)) ⟷ ¬ vl (Ee ξ A)
  using ev-app wff-Neg assms vl-neg ev-type by metis
lemma sat-Dis: assumes wff_o(A) and wff_o(B) and asg ξ
  shows vl (Ee ξ (A ∨ B)) ⟷ vl (Ee ξ A) ∨ vl (Ee ξ B)
  using ev-app wff-App wff-Dis assms vl-dis ev-type by (smt (verit, ccfv-SIG))
lemma sat-ImpB: assumes wff_o(A) and wff_o(B) and asg ξ
  shows vl (Ee ξ (A ⊃ B)) ⟷ (vl (Ee ξ A) ⟶ vl (Ee ξ B))
  unfolding ImpB-def using sat-Dis wff-Not assms sat-Neg by auto
lemma sat-Pi: assumes wff_(σ⇒o)(G) and asg ξ
  shows vl (Ee ξ (Pi σ · G)) ⟷ (∀ d. Dm σ d ⟶ vl (Ap (Ee ξ G) d))
  using ev-app wff-Pi assms vl-pi ev-type by metis
```

lemma *sat-Forall*: **assumes** *wA*: $wff_{\sigma\Rightarrow o}(\mathbf{\Lambda}_\sigma\ b)$ **and** *x*: $x \notin fvs\ b$ **and** *xi*: $asg\ \xi$
shows $vl\ (Ee\ \xi\ (\mathbf{\Pi}_\sigma\ b)) \longleftrightarrow (\forall d.\ Dm\ \sigma\ d \longrightarrow vl\ (Ee\ (\xi(x_\sigma := d))\ (b\langle x^f{}_\sigma\rangle)))$
unfolding *Forall-def* **using** *sat-Pi wA xi ev-abs-app x* **by** *auto*

BKK Lemma 4.2: in a Σ-model with primitive equality (which gives property q with witness $E(=_\alpha)$), Leibniz equality is satisfied exactly by equal denotations.

end

2.9 Σ-Henkin models: the general-model locale (BKK Definition 3.50)

BKK's soundness and completeness theorems (BKK Theorem 7.3, Corollary 7.7) cover all eight model classes $\mathcal{M}_*$; this development instantiates the most specialised one, the class $\mathcal{M}_{\beta f b}$ of Σ-*Henkin models* (BKK Definition 3.50): Σ-models (BKK Definition 3.41) satisfying property b, property f (functionality), and property q (BKK Definitions 3.46 and 3.49). Crucially the function domains need not be full (BKK Definition 3.5): following Henkin — in BKK's words, it is sufficient to require that $\mathcal{D}_{\alpha\Rightarrow\beta}$ "has enough members that any well-formed formula can be evaluated" (BKK Section 2.3.1). We therefore require the λ-conditions — λ-comprehension *gm-lamTy* and the β-condition *gm-beta* of the Σ-evaluation (BKK Definition 3.18) — only for the functions that are *denotations of λ-terms*; equivalently, every wff denotes. A term model cannot be full: with property b the domain $\mathcal{D}_{\iota\Rightarrow o}$ of a full frame over infinite $\mathcal{D}_\iota$ would be uncountable, whereas the term model is countable. (BKK avoid Andrews' term *general models* for this notion; we keep it in the locale name *general-model*, in Andrews' sense.) Standard models — the full case, BKK Definition 3.51 — appear at the end of this section as a sublocale. Beyond BKK, the locale carries the description condition *gm-descB* for the typed description operators *Iota* σ (Andrews 1972, BKK's reference [3]), matched by the rule $NK(\iota)$ of the calculus.

Note that we render the applicative structure abstractly (an application operation **@** on a carrier $'u$) rather than literally over a frame of functions (BKK Definition 3.4); by functionality and BKK Theorem 3.68 the two presentations describe the same class of models up to isomorphism.

locale *general-model* = *frame-sig* +
— every domain is inhabited (part of BKK Definition 3.1, applicative structures)
assumes *gm-nonempty*: $\exists d.\ \mathcal{D}_\sigma\ d$
— property b (BKK Definition 3.46): $\mathcal{D}_o = \{Tv, Fv\}$
and *gm-TF*: $Tv \neq Fv$ **and** *gm-boolean*: $\mathcal{D}_o\ a \longleftrightarrow a = Tv \vee a = Fv$
— application stays in the codomain, and the constants inhabit their domains
and *gm-appTy*: $[\![\mathcal{D}_{\sigma\Rightarrow\tau}\ f;\ \mathcal{D}_\sigma\ a]\!] \Longrightarrow \mathcal{D}_\tau\ (f\ @\ a)$
and *gm-negTy*: $\mathcal{D}_{o\Rightarrow o}\ Ngv$ **and** *gm-disTy*: $\mathcal{D}_{o\Rightarrow o\Rightarrow o}\ Dsv$
and *gm-piTy*: $\mathcal{D}_{(\sigma\Rightarrow o)\Rightarrow o}\ (Piv\ \sigma)$
and *gm-iotaTy*: $\mathcal{D}_{(\sigma\Rightarrow o)\Rightarrow\sigma}\ (Iv\ \sigma)$ **and** *gm-parTy*: $\mathcal{D}_\sigma\ (Jv\ p\ \sigma)$

— Σ-valuation (BKK Figure 2 / Definition 3.41) with $\upsilon = (\lambda a.\ a = Tv)$
and *gm-negB*: $\mathcal{D}_{\mathrm{o}}\ a \Longrightarrow (Ngv\ @\ a = Tv) = (a \neq Tv)$
and *gm-disB*: $[\![\mathcal{D}_{\mathrm{o}}\ a;\ \mathcal{D}_{\mathrm{o}}\ b]\!] \Longrightarrow (Dsv\ @\ a\ @\ b = Tv) = (a = Tv \vee b = Tv)$
and *gm-piB*: $\mathcal{D}_{\sigma\Rightarrow\mathrm{o}}\ f \Longrightarrow (Piv\ \sigma\ @\ f = Tv) = (\forall\, d.\ \mathcal{D}_\sigma\ d \longrightarrow f\ @\ d = Tv)$
— property f (functionality, BKK Definition 3.46) and property q (BKK Definitions 3.46 and 3.49)
and *gm-funct*: $[\![\mathcal{D}_{\sigma\Rightarrow\tau}\ g;\ \mathcal{D}_{\sigma\Rightarrow\tau}\ k;\ \bigwedge a.\ \mathcal{D}_\sigma\ a \Longrightarrow g\ @\ a = k\ @\ a]\!] \Longrightarrow g = k$
— primitive equality (BKK Remark 7.9): $Ev\ \sigma$ satisfies $L^\sigma{}_=$ of BKK Figure 2; property q (BKK Definitions 3.46 and 3.49) follows with witness $Ev\ \sigma$
and *gm-eqTy*: $\mathcal{D}_{\sigma\Rightarrow\sigma\Rightarrow\mathrm{o}}\ (Ev\ \sigma)$
and *gm-eqB*: $[\![\mathcal{D}_\sigma\ a;\ \mathcal{D}_\sigma\ b]\!] \Longrightarrow (Ev\ \sigma\ @\ a\ @\ b = Tv) = (a = b)$
— the description condition (beyond BKK, cf. Andrews 1972): if f behaves as the singleton $\{a\}$, description picks out a
and *gm-descB*: $[\![\mathcal{D}_{\sigma\Rightarrow\mathrm{o}}\ f;\ \mathcal{D}_\sigma\ a;\ \forall\, b.\ \mathcal{D}_\sigma\ b \longrightarrow (f\ @\ b = Tv) = (b = a)]\!] \Longrightarrow Iv\ \sigma\ @\ f = a$
— Henkin λ-conditions: λ-comprehension (the Σ-evaluation is total on wffs, BKK Definition 3.18) and the β-condition (BKK Definition 3.18(4)), at the *denotation* functions only
and *gm-lamTy*: $[\![wff_{\sigma\Rightarrow\tau}(\mathbf{\Lambda}_\sigma\ bd);\ \forall\, n\ \varrho.\ \mathcal{D}_\varrho\ (\xi\ n\ \varrho)]\!] \Longrightarrow \mathcal{D}_{\sigma\Rightarrow\tau}\ (\!|\mathbf{\Lambda}_\sigma\ bd|\!)_\xi)$
and *gm-beta*: $[\![wff_\tau(bd\langle x^f{}_\sigma\rangle);\ x \notin fvs\ bd;\ \forall\, n\ \varrho.\ \mathcal{D}_\varrho\ (\xi\ n\ \varrho);\ \mathcal{D}_\sigma\ a]\!]$
$\Longrightarrow (\!|\mathbf{\Lambda}_\sigma\ bd|\!)_\xi\ @\ a = (\!|bd\langle x^f{}_\sigma\rangle|\!)_{\xi(x_\sigma := a)}$
begin

An assignment is *total* (*resp*, respects the domains everywhere) if it maps every typed variable into the matching domain — BKK's assignment *into* the structure.

definition *resp* **where** $resp\ \xi \longleftrightarrow (\forall\, (n{::}nat)\ \tau.\ \mathcal{D}_\tau\ (\xi\ n\ \tau))$
lemma *Tv-dom* [*simp*]: $\mathcal{D}_{\mathrm{o}}\ Tv$ **and** *Fv-dom* [*simp*]: $\mathcal{D}_{\mathrm{o}}\ Fv$
using *gm-boolean* **by** *auto*

Every well-formed term denotes in the domain of its type (BKK Definition 3.18): the λ-comprehension condition *gm-lamTy* is exactly what makes the abstraction case go through.

lemma *den-dom*: $wff_\sigma(t) \Longrightarrow resp\ \xi \Longrightarrow \mathcal{D}_\sigma\ (\!|t|\!)_\xi)$
proof (*induction arbitrary*: ξ *rule*: *wff.induct*)
case *wff-App* **thus** *?case* **using** *den.simps*(*8*) *gm-appTy* **by** *fastforce*
next
case *wff-Abs* **thus** *?case* **using** *gm-lamTy resp-def wff.wff-Abs* **by** *blast*
qed(*auto simp*: *resp-def gm-parTy gm-negTy gm-disTy gm-piTy gm-iotaTy gm-eqTy*)

The semantic β rule at the term level (BKK Remark 3.19): applying an abstraction to a well-formed argument evaluates the opened body.

lemma *den-App-Abs*:
assumes *wb*: $wff_{\sigma\Rightarrow\tau}(\mathbf{\Lambda}_\sigma\ b)$ **and** *wa*: $wff_\sigma(a)$ **and** *r*: *resp* ξ
shows $(\!|(\mathbf{\Lambda}_\sigma\ b)\cdot a|\!)_\xi = (\!|b\langle a\rangle|\!)_\xi$
proof −
define x **where** $x = fresh\ (fvs\ b)$
have *xb*: $x \notin fvs\ b$

using *fresh-notin* **unfolding** *x-def* **by** *simp*
have *da*: $\mathcal{D}_\sigma\ (\llparenthesis a \rrparenthesis_\xi)$ **using** *wa r* **by** (*rule den-dom*)
have $\llparenthesis (\mathbf{\Lambda}_\sigma\ b) \cdot a \rrparenthesis_\xi = \llparenthesis \mathbf{\Lambda}_\sigma\ b \rrparenthesis_\xi\ @\ \llparenthesis a \rrparenthesis_\xi$ **by** (*simp add*: *den.simps*(*8*))
also have $\ldots = \llparenthesis b\langle x^f{}_\sigma \rangle \rrparenthesis_{\xi(x_\sigma := \llparenthesis a \rrparenthesis_\xi)}$
using *da gm-beta r resp-def wff-Abs-open wb xb* **by** *blast*
also have $\ldots = \llparenthesis b\langle a \rangle \rrparenthesis_\xi$ **using** *den-beta wff-lc wa xb* **by** *metis*
finally show *?thesis*.
qed

end

2.10 Closed well-formed terms and simultaneous substitution

A *closed* well-formed term, BKK's $cwff_\sigma$ (BKK Section 2.2; BKK reserve *sentence* for closed formulae of type o — parameters are allowed, they play the role of BKK's constants). These are the carriers of the term model.

definition *cwff* :: *ty* ⇒ *'p tm* ⇒ *bool* **where** $cwff\ \sigma\ A \longleftrightarrow wff_\sigma(A) \wedge fvs\ A = \{\}$
lemma *cwffI*: $wff_\sigma(A) \Longrightarrow fvs\ A = \{\} \Longrightarrow cwff\ \sigma\ A$ **by** (*simp add*: *cwff-def*)
lemma *cwff-wff*: $cwff\ \sigma\ A \Longrightarrow wff_\sigma(A)$ **by** (*simp add*: *cwff-def*)
lemma *cwff-closed*: $cwff\ \sigma\ A \Longrightarrow fvs\ A = \{\}$ **by** (*simp add*: *cwff-def*)
lemma *cwff-lc*: $cwff\ \sigma\ A \Longrightarrow lc\ A$ **by** (*metis cwff-def wff-lc*)
lemma *cwff-Neg*: $cwff$ (o⇒o) *Neg*
and *cwff-Dis*: $cwff$ (o⇒o⇒o) *Dis*
and *cwff-Pi*: $cwff$ ((σ⇒o)⇒o) (*Pi* σ)
and *cwff-Iota*: $cwff$ ((σ⇒o)⇒σ) (*Iota* σ)
and *cwff-TrueB*: $cwff$ o ⊤
and *cwff-FalseB*: $cwff$ o ⊥
by (*auto simp*: *cwff-def wff-Neg wff-Dis wff-Pi wff-Iota*)
lemma *cwff-Eq*: $cwff$ (σ⇒σ⇒o) (*Eq* σ) **by** (*simp add*: *cwff-def wff-Eq*)
lemma *cwff-Par*: $cwff\ \sigma\ (p^p{}_\sigma)$ **by** (*simp add*: *cwff-def wff-Par*)
lemma *cwff-App*: $cwff\ (\sigma \Rightarrow \tau)\ s \Longrightarrow cwff\ \sigma\ t \Longrightarrow cwff\ \tau\ (s \cdot t)$
by (*auto simp*: *cwff-def intro*: *wff.wff-App*)
lemma *cwff-opn*: $cwff\ (\sigma \Rightarrow \tau)\ (\mathbf{\Lambda}_\sigma\ b) \Longrightarrow cwff\ \sigma\ a \Longrightarrow cwff\ \tau\ (b\langle a \rangle)$
by (*metis* (*no-types*, *opaque-lifting*) *cwff-def fvs.simps*(*10*) *fvs-opn subset-empty sup.idem wff-opn*)
lemma *cwff-unique*: $cwff\ \sigma\ A \Longrightarrow cwff\ \tau\ A \Longrightarrow \sigma = \tau$
by (*auto simp*: *cwff-def dest*: *wff-unique*)

Converse of $[\![wff\ {}_{?\sigma} \Rightarrow {}_{?\tau}(\mathbf{\Lambda}_{?\sigma}\ ?b);\ ?x \notin fvs\ ?b]\!] \Longrightarrow wff\ {}_{?\tau}(?b\langle ?x^f\ {}_{?\sigma} \rangle)$: a body that is well-typed when opened with *one* fresh variable yields a well-typed abstraction (all fresh openings are α-variants).

lemma *wff-Abs-open-rev*: $wff_\tau(b\langle x^f{}_\sigma \rangle) \Longrightarrow x \notin fvs\ b \Longrightarrow wff_{\sigma \Rightarrow \tau}(\mathbf{\Lambda}_\sigma\ b)$
by (*metis fsub-intro wff-AbsI wff-Fre wff-fsub*)

Simultaneous substitution of a (closed) term for every free variable — the analogue of the closing substitution σ in BKK's term evaluation (BKK Definition 3.35). Because the replacements are closed, it commutes with

opening — the locally-nameless analogue of BKK's parallel substitution, with no binder renaming.

primrec *msub* :: (*nat* ⇒ *ty* ⇒ *'p tm*) ⇒ *'p tm* ⇒ *'p tm* **where**
  msub ϱ (*Bnd i*) = *Bnd i*
| *msub ϱ* ($n^f\ \tau$) = *ϱ n τ*
| *msub ϱ* ($p^p\ \tau$) = $p^p\ \tau$
| *msub ϱ Neg* = *Neg*
| *msub ϱ Dis* = *Dis*
| *msub ϱ* (*Pi τ*) = *Pi τ*
| *msub ϱ* (*Iota τ*) = *Iota τ*
| *msub ϱ* (*Eq τ*) = *Eq τ*
| *msub ϱ* (*s* ∙ *t*) = (*msub ϱ s*) ∙ (*msub ϱ t*)
| *msub ϱ* (**Λ**$_\tau$ *b*) = **Λ**$_\tau$ (*msub ϱ b*)

Parallel-substitution notation: $u[\![\varrho]\!]$ applies the substitution ϱ to every free variable of u.

syntax *-msub* :: *logic* ⇒ *logic* ⇒ *logic* (‹-[![-]!]› [*1000*, *0*] *1000*)
syntax-consts *-msub* == *msub*
translations $u[\![\varrho]\!]$ ⇌ *CONST msub ϱ u*

lemma *msub-opn*: (⋀*n τ*. *lc* (*ϱ n τ*)) ⟹ *msub ϱ* (*opn k u t*) = *opn k* (*msub ϱ u*) (*msub ϱ t*)
  by (*induction t arbitrary*: *k*) *auto*
lemma *msub-cong*: (⋀*n τ*. (*n*, *τ*) ∈ *occ t* ⟹ *ϱ n τ* = *ϱ' n τ*) ⟹ *msub ϱ t* = *msub ϱ' t*
  by (*induction t*) *auto*
lemma *fvs-msub*: (⋀*n τ*. *fvs* (*ϱ n τ*) = {}) ⟹ *fvs* (*msub ϱ t*) = {}
  by (*induction t*) *auto*
lemma *wff-msub*:
  $wff_\sigma(t)$ ⟹ (⋀*n τ*. wff_τ(*ϱ n τ*)) ⟹ wff_σ(*msub ϱ t*)
proof (*induction σ t arbitrary*: *ϱ rule*: *wff.induct*)
  case (*wff-Abs L τ b σ*)
  have *lcr*: *lc* (*ϱ n τ'*) **for** *n τ'* **using** *wff-Abs.prems*
    by (*rule wff-lc*)
  have $wff_{\sigma\Rightarrow\tau}$(**Λ**$_\sigma$ (*msub ϱ b*))
  proof (*rule wff.wff-Abs*[*of L* ∪ *fvs b*])
    fix *y* **assume** *y*: *y* ∉ *L* ∪ *fvs b*
    let *?ϱ* = *ϱ*(*y* := (*ϱ y*)(*σ* := $y^f{}_\sigma$))
    have *e*: *msub ?ϱ* (*b*⟨$y^f{}_\sigma$⟩) = (*msub ϱ b*)⟨$y^f{}_\sigma$⟩
      by (*smt* (*verit*, *ccfv-SIG*) *Un-iff fun-upd-other fun-upd-same*
          fvs-eq-fst-occ image-eqI lc-Fre lcr msub.simps(*2*)
          msub-cong msub-opn prod.sel(*1*) *y*)
    have wff_τ(*msub ?ϱ* (*b*⟨$y^f{}_\sigma$⟩))
      using *wff-Abs wff.wff-Fre y* **by** (*metis Un-iff fun-upd-other fun-upd-same*)
    thus wff_τ((*msub ϱ b*)⟨$y^f{}_\sigma$⟩) **using** *e* **by** *simp*
  qed (*simp add*: *wff-Abs*)
  thus *?case* **by** *simp*
qed(*auto intro*: *wff.intros*)

A well-formed term becomes a *closed* well-formed term under a closed simultaneous substitution — the instance used to build the term model.

lemma *cwff-msub*: $wff_\sigma(t) \Longrightarrow (\bigwedge n\ \tau.\ cwff\ \tau\ (\varrho\ n\ \tau)) \Longrightarrow cwff\ \sigma\ (msub\ \varrho\ t)$
by (*metis cwff-def wff-msub fvs-msub*)

A closed term is untouched by simultaneous substitution.

lemma *msub-closed*: $fvs\ t = \{\} \Longrightarrow msub\ \varrho\ t = t$ **by** (*induction t*) *auto*

2.11 The valuation locale

The *valuation* locale axiomatises what a term model provides: a carrier $'u$ with an application **@** and an evaluation $\mathcal{V}$ of closed well-formed terms. It abstracts the *quotient* of BKK's term evaluation (BKK Definition 3.35) by Leibniz equality, as constructed in the model-existence proof (BKK Theorem 6.33) — note that the bare term evaluation $\mathcal{TE}(\Sigma)^\beta$ is *not* functional (BKK Remark 3.37); functionality only holds after the quotient. The axioms: *v-app*/*v-beta* are the evaluation conditions (BKK Definition 3.18(2),(4)); *v-neg*, *v-dis*, *v-pi* are $L_\neg$, $L_\vee$, $L^\sigma{}_\forall$ (BKK Figure 2); *v-ext* is functionality (property f), *v-eq* is primitive equality $L^\sigma{}_=$ (whence property q), *v-desc* the description condition, *v-type* type-disjointness of the domains, and *v-TF*/*v-bool* are property b (BKK Definition 3.46).

locale *valuation* =
  fixes $Dv :: ty \Rightarrow 'u \Rightarrow bool$ ($\langle\mathcal{D}\_\rangle$)
    and $Vap :: 'u \Rightarrow 'u \Rightarrow 'u$ (**infixl** ‹@› *200*)
    and $Val :: 'p\ tm \Rightarrow 'u$ ($\langle\mathcal{V}\rangle$)
  assumes *v-dom*: $\mathcal{D}_\sigma\ d \longleftrightarrow (\exists\, t.\ cwff\ \sigma\ t \wedge d = \mathcal{V}\ t)$
      and *v-app*: $cwff\ (\sigma{\Rightarrow}\tau)\ s \Longrightarrow cwff\ \sigma\ t \Longrightarrow \mathcal{V}\ (s \cdot t) = \mathcal{V}\ s\ @\ \mathcal{V}\ t$
      and *v-beta*: $cwff\ (\sigma{\Rightarrow}\tau)\ (\mathbf{\Lambda}_\sigma\ b) \Longrightarrow cwff\ \sigma\ a \Longrightarrow \mathcal{V}\ (\mathbf{\Lambda}_\sigma\ b)\ @\ \mathcal{V}\ a = \mathcal{V}\ (b\langle a\rangle)$
      and *v-TF*: $\mathcal{V}\ \top \neq \mathcal{V}\ \bot$
      and *v-bool*: $cwff\ o\ \varphi \Longrightarrow \mathcal{V}\ \varphi = \mathcal{V}\ \top \vee \mathcal{V}\ \varphi = \mathcal{V}\ \bot$
      and *v-neg*: $cwff\ o\ \varphi \Longrightarrow \mathcal{V}\ (\neg\ \varphi) = (if\ \mathcal{V}\ \varphi = \mathcal{V}\ \top\ then\ \mathcal{V}\ \bot\ else\ \mathcal{V}\ \top)$
      and *v-dis*: $cwff\ o\ \varphi \Longrightarrow cwff\ o\ \psi \Longrightarrow$
            $\mathcal{V}\ (\varphi \vee \psi) = (if\ \mathcal{V}\ \varphi = \mathcal{V}\ \top \vee \mathcal{V}\ \psi = \mathcal{V}\ \top\ then\ \mathcal{V}\ \top\ else\ \mathcal{V}\ \bot)$
      and *v-pi*: $cwff\ (\sigma{\Rightarrow}o)\ f \Longrightarrow \mathcal{V}\ ((Pi\ \sigma) \cdot f) =$
          $(if\ (\forall\, a.\ cwff\ \sigma\ a \longrightarrow \mathcal{V}\ f\ @\ \mathcal{V}\ a\ = \mathcal{V}\ \top)\ then\ \mathcal{V}\ \top\ else\ \mathcal{V}\ \bot)$
      and *v-ext*: $cwff\ (\sigma{\Rightarrow}\tau)\ g \Longrightarrow cwff\ (\sigma{\Rightarrow}\tau)\ h$
            $\Longrightarrow (\bigwedge a.\ cwff\ \sigma\ a \Longrightarrow \mathcal{V}\ g\ @\ \mathcal{V}\ a = \mathcal{V}\ h\ @\ \mathcal{V}\ a) \Longrightarrow \mathcal{V}\ g = \mathcal{V}\ h$
      and *v-eq*: $cwff\ \sigma\ a \Longrightarrow cwff\ \sigma\ b \Longrightarrow (\mathcal{V}\ (Eq\ \sigma)\ @\ \mathcal{V}\ a\ @\ \mathcal{V}\ b = \mathcal{V}\ \top) = (\mathcal{V}\ a = \mathcal{V}\ b)$
      and *v-desc*: $cwff\ (\sigma{\Rightarrow}o)\ f \Longrightarrow cwff\ \sigma\ a$
            $\Longrightarrow (\forall\, b.\ cwff\ \sigma\ b \longrightarrow (\mathcal{V}\ f\ @\ \mathcal{V}\ b = \mathcal{V}\ \top) = (\mathcal{V}\ b = \mathcal{V}\ a))$
            $\Longrightarrow \mathcal{V}\ ((Iota\ \sigma) \cdot f) = \mathcal{V}\ a$
      and *v-type*: $cwff\ \sigma\ s \Longrightarrow cwff\ \tau\ t \Longrightarrow \mathcal{V}\ s = \mathcal{V}\ t \Longrightarrow \sigma = \tau$
begin

lemma *v-domI*: $cwff\ \sigma\ t \Longrightarrow \mathcal{D}_\sigma\ (\mathcal{V}\ t)$ **using** *v-dom* **by** *blast*

end

2.12 The term evaluation (BKK Section 6)

A valuation extends to an evaluation function by simultaneous substitution of representatives: $E_\xi(A) := \mathcal{V}([\varrho_\xi]A)$, BKK's evaluation for the term structure. β-respect is inherited from *v-beta* under closing substitutions, functionality is *v-ext*, and the remaining valuation conditions supply the L-properties — so every valuation is directly a Σ-model in the class $\mathcal{M}_{\beta fb}$, with no detour through the recursive denotation.

context *valuation*
begin

definition *vresp* **where** *vresp* $\xi \longleftrightarrow (\forall (n{::}nat)\ \tau.\ \mathcal{D}_\tau\ (\xi\ n\ \tau))$
definition *rep-of* **where** *rep-of* $\xi\ (n{::}nat)\ \tau = (SOME\ t.\ cwff\ \tau\ t \wedge \xi\ n\ \tau = \mathcal{V}\ t)$
lemma *rep-of-spec*: **assumes** *vresp* ξ **shows** *cwff* τ (*rep-of* $\xi\ n\ \tau) \wedge \xi\ n\ \tau = \mathcal{V}$ (*rep-of* $\xi\ n\ \tau$)
proof −
  have $\mathcal{D}_\tau\ (\xi\ n\ \tau)$ **using** *assms*[*unfolded vresp-def*] **by** *blast*
  hence $\exists\, t.\ cwff\ \tau\ t \wedge \xi\ n\ \tau = \mathcal{V}\ t$ **using** *v-dom* **by** *blast*
  thus *?thesis* **unfolding** *rep-of-def* **by** (*rule someI-ex*)
qed

The valuation respects β-conversion under closing substitutions (BKK's quotient of the term structure by β, Section 6): the abstraction case is pointwise by *v-beta* and closed by *v-ext*.

lemma *beq-V*: $s \approx_{\varrho'} t \Longrightarrow (\bigwedge n\ \tau.\ cwff\ \tau\ (\varrho\ n\ \tau)) \Longrightarrow \mathcal{V}\ (msub\ \varrho\ s) = \mathcal{V}\ (msub\ \varrho\ t)$
proof (*induction arbitrary*: ϱ *rule*: *beq.induct*)
  case (*beta* $\sigma\ \varrho'\ b\ a$)
  have *lcr*: *lc* $(\varrho\ n\ \tau)$ **for** $n\ \tau$
    using *beta.prems* **by** (*auto intro*: *wff-lc cwff-wff*)
  have *cw1*: *cwff* $(\sigma \Rightarrow \varrho')\ (\mathbf{\Lambda}_\sigma\ (msub\ \varrho\ b))$
    using *cwff-msub*[*OF beta.hyps*(*1*) *beta.prems*] **by** *simp*
  have $\mathcal{V}\ (msub\ \varrho\ ((\mathbf{\Lambda}_\sigma\ b) \cdot a)) = \mathcal{V}\ (\mathbf{\Lambda}_\sigma\ (msub\ \varrho\ b))$ @ $\mathcal{V}\ (msub\ \varrho\ a)$
    by (*simp add*: *v-app*[*OF cw1 cwff-msub*[*OF beta.hyps*(*2*) *beta.prems*]])
  also have $\ldots = \mathcal{V}\ ((msub\ \varrho\ b)\langle msub\ \varrho\ a\rangle)$
    by (*rule v-beta*[*OF cw1 cwff-msub*[*OF beta.hyps*(*2*) *beta.prems*]])
  also have $\ldots = \mathcal{V}\ (msub\ \varrho\ (b\langle a\rangle))$ **by** (*simp add*: *msub-opn*[*OF lcr*])
  finally show *?case*.
next case (*appL* $s\ \sigma\ \varrho'\ s'\ t$) **thus** *?case*
  by (*smt* (*verit*, *ccfv-SIG*) *beq-wff cwff-msub msub.simps*(*9*) *valuation.v-app valuation-axioms*)
next case (*appR* $t\ \sigma\ t'\ \varrho'\ s$)
  have *wt*: $wff_\sigma(t)$ **and** *wt'*: $wff_\sigma(t')$
    using *beq-wff*[*OF appR.hyps*(*1*)] **by** *auto*
  thus *?case* **using** *appR cwff-msub* **by** (*metis msub.simps*(*9*) *v-app*)
next case (*abs* $L\ b\ \sigma\ \tau\ b'$)

obtain x **where** x: $x \notin L \cup fvs\ b \cup fvs\ b'$
by (*meson abs.hyps*(1) *ex-new-if-finite finite-UnI finite-fvs infinite-UNIV-nat*)
have *wb*: $wff_\tau(b\langle x^f{}_\sigma\rangle)$ **and** *wb'*: $wff_\tau(b'\langle x^f{}_\sigma\rangle)$
using *beq-wff*[*OF abs.hyps*(2)[*of* x]] x **by** *auto*
have *wA*: $wff_{\sigma\Rightarrow\tau}(\mathbf{\Lambda}_\sigma\ b)$ **using** *wff-Abs-open-rev*[*OF wb*] x **by** *auto*
have *wA'*: $wff_{\sigma\Rightarrow\tau}(\mathbf{\Lambda}_\sigma\ b')$ **using** *wff-Abs-open-rev*[*OF wb'*] x **by** *auto*
have *cwff* $(\sigma\Rightarrow\tau)$ $(\mathbf{\Lambda}_\sigma\ (msub\ \varrho\ b))$
using *cwff-msub*[*OF wA abs.prems*] **by** *simp*
moreover have *cwff* $(\sigma\Rightarrow\tau)$ $(\mathbf{\Lambda}_\sigma\ (msub\ \varrho\ b'))$
using *cwff-msub*[*OF wA' abs.prems*] **by** *simp*
moreover have $\mathcal{V}\ (\mathbf{\Lambda}_\sigma\ (msub\ \varrho\ b))\ @\ \mathcal{V}\ a = \mathcal{V}\ (\mathbf{\Lambda}_\sigma\ (msub\ \varrho\ b'))\ @\ \mathcal{V}\ a$ **if** *a*: *cwff* σ a **for** a
proof −
let *?ϱ* = $\varrho(x := (\varrho\ x)(\sigma := a))$
have *cr*: *cwff* τ' (*?ϱ* n τ') **for** n τ' **using** *abs.prems a* **by** *auto*
have *lcr*: *lc* (*?ϱ* n τ') **for** n τ' **using** *cr*
by (*meson wff-lc cwff-wff*)
have *mb*: *msub ?ϱ b* = *msub ϱ b*
using *msub-cong x fvs-eq-fst-occ image-iff*
by (*smt* (*verit*, *best*) *UnCI fst-conv fun-upd-other*)
have *mb'*: *msub ?ϱ b'* = *msub ϱ b'*
using *msub-cong fvs-eq-fst-occ image-iff x*
by (*smt* (*verit*, *ccfv-threshold*) *Un-iff fst-conv fun-upd-other*)
have *e2*: *msub ?ϱ* $((x^f{}_\sigma) :: 'p\ tm) = a$ **by** *simp*
have *ob*: *msub ?ϱ* $(b\langle x^f{}_\sigma\rangle) = (msub\ \varrho\ b)\langle a\rangle$
by (*simp only*: *msub-opn*[*OF lcr*] *e2 mb*)
have *ob'*: *msub ?ϱ* $(b'\langle x^f{}_\sigma\rangle) = (msub\ \varrho\ b')\langle a\rangle$
by (*simp only*: *msub-opn*[*OF lcr*] *e2 mb'*)
have *IH*: $\mathcal{V}\ (msub\ ?\varrho\ (b\langle x^f{}_\sigma\rangle)) = \mathcal{V}\ (msub\ ?\varrho\ (b'\langle x^f{}_\sigma\rangle))$
by (*rule abs.IH*) (*use x cr* **in** *auto*)
have $\mathcal{V}\ ((msub\ \varrho\ b)\langle a\rangle) = \mathcal{V}\ ((msub\ \varrho\ b')\langle a\rangle)$ **using** *IH*
by (*simp only*: *ob ob'*)
thus *?thesis* **using** *v-beta calculation a* **by** *simp*
qed
ultimately show *?case* **using** *v-ext* **by** *simp*
qed *auto*

BKK's evaluation function for the term model.

definition *Ev* :: $(nat \Rightarrow ty \Rightarrow 'u) \Rightarrow 'p\ tm \Rightarrow 'u$ **where** *Ev* $\xi\ A = \mathcal{V}\ (msub\ (rep\text{-}of\ \xi)\ A)$

end

Every valuation is a Σ-model in the class $\mathcal{M}_{\beta f b}$ (the direct construction of BKK Section 6).

sublocale *valuation* ⊆ *bkkA*: *app-struct Dv Vap*
proof (*unfold-locales*, *goal-cases*)
case (*1* α) **show** *?case* **using** *v-domI*[*OF cwff-Par*] **by** *blast*
next case (*2* α β f a) **thus** *?case* **using** *cwff-App v-dom valuation.v-app*

valuation-axioms **by** *fastforce*
qed

sublocale *valuation* $\subseteq$ *bkkM*: *bkk-model Dv Vap Ev* $\lambda a.\ a = \mathcal{V}\ \top$
proof (*unfold-locales*, *goal-cases*)
case (*1* τ *A* ξ) **thus** *?case*
by (*metis Ev-def bkkA.asg-def cwff-msub rep-of-spec v-domI vresp-def*)
next case (*2* ξ *n* σ) **thus** *?case*
using *Ev-def bkkA.asg-def rep-of-spec vresp-def* **by** *auto*
next case (*3* σ τ *F A* ξ) **thus** *?case*
by (*metis Ev-def bkkA.asg-def cwff-msub msub.simps*(*9*) *rep-of-spec v-app vresp-def*)
next case (*4* τ *A* ξ ξ')
hence *rep-of* ξ *n* σ = *rep-of* ξ' *n* σ **if** (*n*, σ) $\in$ *occ A* **for** *n* σ
using *that* **by** (*auto simp*: *rep-of-def*)
thus *?case* **unfolding** *Ev-def* **by** (*simp cong*: *msub-cong*)
next case *5* **thus** *?case* **using** *Ev-def beq-V bkkA.asg-def rep-of-spec vresp-def* **by** *auto*
next case *6* **thus** *?case* **by** (*metis Ev-def cwff-Neg msub.simps*(*4*) *v-TF v-app v-dom v-neg*)
next case (*7* ξ *a b*)
then obtain φ ψ **where** *ab*: *a* = $\mathcal{V}$ φ *b* = $\mathcal{V}$ ψ **and** *c*: *cwff* o φ *cwff* o ψ
using *v-dom* **by** *blast*
have *e*: *Ev* ξ *Dis* = $\mathcal{V}$ *Dis* **by** (*simp add*: *Ev-def*)
show *?case* **unfolding** *e ab* **using** *v-TF*
by (*metis c*(*1*,*2*) *cwff-App cwff-Dis v-app v-dis*)
next case (*8* ξ σ *f*)
then obtain *g* **where** *fg*: *f* = $\mathcal{V}$ *g* **and** *cg*: *cwff* (σ $\Rightarrow$ o) *g*
using *v-dom* **by** *blast*
have *e*: *Ev* ξ (*Pi* σ) = $\mathcal{V}$ (*Pi* σ) **by** (*simp add*: *Ev-def*)
have *q*: ($\forall$ *d*. $\mathcal{D}_\sigma$ *d* $\longrightarrow$ $\mathcal{V}$ *g* @ *d* = $\mathcal{V}$ $\top$) = ($\forall$ *a*. *cwff* σ *a* $\longrightarrow$ $\mathcal{V}$ *g* @ $\mathcal{V}$ *a* = $\mathcal{V}$ $\top$)
using *v-dom* **by** *metis*
show *?case* **unfolding** *e fg* **using** *v-TF*
by (*auto simp*: *v-app*[*OF cwff-Pi cg*, *symmetric*] *v-pi*[*OF cg*] *q*)
next case *9* **thus** *?case* **using** *Ev-def v-dom v-eq* **by** *fastforce*
next case *10* **thus** *?case* **by** (*smt* (*verit*, *best*) *Ev-def cwff-Iota msub.simps*(*7*) *v-app v-desc v-dom*)
next case *11* **thus** *?case* **by** (*smt* (*verit*, *ccfv-threshold*) *bkkA.functional-def v-dom v-ext*)
next case *12* **thus** *?case* **by** (*metis v-bool v-dom*)
qed

2.13 Standard models (BKK Definition 3.51)

A Σ-*standard model* (BKK Definition 3.51) is a Σ-Henkin model over a *full* frame (BKK Definition 3.5): every set-function between domains has a representative. We record fullness by the universal abstraction laws *Lm-dom* and *beta-Lm*, quantified over *all* functions $h :: 'u \Rightarrow 'u$ — strictly stronger than the Henkin conditions of *general-model*. On top of the full frame we

assume the Σ-valuation conditions (BKK Definition 3.41, Figure 2) with property b, property f (functionality) and property q (BKK Definition 3.46).

locale *standard-model* = *frame-sig* +
— fullness (BKK Definition 3.5): every function has a representative, @ computes it
assumes *beta-Lm*: $[\![\bigwedge d.\ \mathcal{D}_\sigma\ d \Longrightarrow \mathcal{D}_\tau\ (h\ d);\ \mathcal{D}_\sigma\ a]\!] \Longrightarrow Lm\ \sigma\ h\ @\ a = h\ a$
and *Lm-dom*: $(\bigwedge d.\ \mathcal{D}_\sigma\ d \Longrightarrow \mathcal{D}_\tau\ (h\ d)) \Longrightarrow \mathcal{D}_{\sigma\Rightarrow\tau}\ (Lm\ \sigma\ h)$
and *Ap-dom*: $[\![\mathcal{D}_{\sigma\Rightarrow\tau}\ f;\ \mathcal{D}_\sigma\ a]\!] \Longrightarrow \mathcal{D}_\tau\ (f\ @\ a)$
— the logical constants and parameters inhabit their domains
and *Ngv-dom*: $\mathcal{D}_{o\Rightarrow o}\ Ngv$
and *Dsv-dom*: $\mathcal{D}_{o\Rightarrow o\Rightarrow o}\ Dsv$
and *Piv-dom*: $\mathcal{D}_{(\sigma\Rightarrow o)\Rightarrow o}\ (Piv\ \sigma)$
and *Iv-dom*: $\mathcal{D}_{(\sigma\Rightarrow o)\Rightarrow\sigma}\ (Iv\ \sigma)$
and *Ev-dom*: $\mathcal{D}_{\sigma\Rightarrow\sigma\Rightarrow o}\ (Ev\ \sigma)$ **and** *Jv-dom*: $\mathcal{D}_\sigma\ (Jv\ p\ \sigma)$
— property b (BKK Definition 3.46): $\mathcal{D}_o = \{Tv, Fv\}$
and *TF*: $Tv \neq Fv$ **and** *boolean*: $\mathcal{D}_o\ a \longleftrightarrow a = Tv \vee a = Fv$
— Σ-valuation (BKK Definition 3.41, Figure 2) with $\upsilon = (\lambda a.\ a = Tv)$
and *Lneg*: $\mathcal{D}_o\ a \Longrightarrow (Ngv\ @\ a = Tv) = (a \neq Tv)$
and *Ldis*: $[\![\mathcal{D}_o\ a;\ \mathcal{D}_o\ b]\!] \Longrightarrow (Dsv\ @\ a\ @\ b = Tv) = (a = Tv \vee b = Tv)$
and *Lall*: $\mathcal{D}_{\sigma\Rightarrow o}\ f \Longrightarrow (Piv\ \sigma\ @\ f = Tv) = (\forall d.\ \mathcal{D}_\sigma\ d \longrightarrow f\ @\ d = Tv)$
and *Leq*: $[\![\mathcal{D}_\sigma\ a;\ \mathcal{D}_\sigma\ b]\!] \Longrightarrow (Ev\ \sigma\ @\ a\ @\ b = Tv) = (a = b)$
— properties f and q (BKK Definition 3.46)
and *funct*: $[\![\mathcal{D}_{\sigma\Rightarrow\tau}\ g;\ \mathcal{D}_{\sigma\Rightarrow\tau}\ k;\ \bigwedge a.\ \mathcal{D}_\sigma\ a \Longrightarrow g\ @\ a = k\ @\ a]\!] \Longrightarrow g = k$
— the description condition (beyond BKK, cf. Andrews 1972)
and *descB*: $[\![\mathcal{D}_{\sigma\Rightarrow o}\ f;\ \mathcal{D}_\sigma\ a;\ \forall b.\ \mathcal{D}_\sigma\ b \longrightarrow (f\ @\ b = Tv) = (b = a)]\!] \Longrightarrow Iv\ \sigma\ @\ f = a$
begin

As in *general-model*, membership of the truth values follows from property b.

lemma *Tv-dom* [*simp*]: $\mathcal{D}_o\ Tv$ **and** *Fv-dom* [*simp*]: $\mathcal{D}_o\ Fv$
by (*simp-all add*: *boolean*)

In a full frame every domain is inhabited (for o by Tv, for ι by applying Iv to a representable predicate, for function types by a constant function). BKK build non-emptiness into the applicative structure (BKK Definition 3.1).

lemma *dom-nonempty*: $\exists d.\ \mathcal{D}_\tau\ d$ **by** (*metis Jv-dom*)

In a full frame every well-formed term denotes in the domain of its type (the standard homomorphic construction, BKK Section 2.3.1): the abstraction case is immediate from fullness.

lemma *std-den-dom*: $wff_\sigma(t) \Longrightarrow \forall n\ \varrho.\ \mathcal{D}_\varrho\ (\xi\ n\ \varrho) \Longrightarrow \mathcal{D}_\sigma\ (\!|t|\!)_\xi$
proof (*induction arbitrary*: ξ *rule*: *wff.induct*)
case (*wff-App* $\sigma\ \tau\ s\ t$) **thus** *?case*
by (*metis Ap-dom frame-sig.den.simps*(*8*))
next

case (*wff-Abs L* τ *b* σ)
define *x* **where** $x = fresh\ (L \cup fvs\ b)$
have *xL*: $x \notin L$ **and** *xb*: $x \notin fvs\ b$
using *fresh-notin*[*of* $L \cup fvs\ b$] *wff-Abs* **unfolding** *x-def* **by** *auto*
have $\mathcal{D}_{\sigma \Rightarrow \tau}\ (Lm\ \sigma\ (\lambda d.\ (\!|b\langle x^f{}_\sigma\rangle|\!)_{\xi(x_\sigma\ :=\ d)}))$
by (*simp add*: *Lm-dom upd-def wff-Abs.IH wff-Abs.prems xL*)
thus *?case* **by** (*simp add*: *den-Abs*[*OF xb*])
qed (*simp-all add*: *Jv-dom Ngv-dom Dsv-dom Piv-dom Iv-dom Ev-dom*)

end

Every standard model is a Σ-Henkin general model (BKK Definition 3.51 is a special case of Definition 3.50): the universal abstraction laws specialise to the denotation functions.

sublocale *standard-model* ⊆ *general-model Dm Ap Lm Tv Fv Ngv Dsv Iv Ev Piv Jv*
proof
show $\mathcal{D}_{\mathrm{o}}\ a \longleftrightarrow a = Tv \vee a = Fv$ **for** *a* **using** *boolean Tv-dom Fv-dom* **by** *blast*
show $\mathcal{D}_{\sigma \Rightarrow \tau}\ g \Longrightarrow \mathcal{D}_{\sigma \Rightarrow \tau}\ k \Longrightarrow (\bigwedge a.\ \mathcal{D}_\sigma\ a \Longrightarrow g\ @\ a = k\ @\ a) \Longrightarrow g = k$ **for** $\sigma\ \tau\ g\ k$
by (*rule funct*)
show $wff_{\sigma \Rightarrow \tau}(\mathbf{\Lambda}_\sigma\ bd) \Longrightarrow \forall n\ \varrho.\ \mathcal{D}_\varrho\ (\xi\ n\ \varrho) \Longrightarrow \mathcal{D}_{\sigma \Rightarrow \tau}\ (\!|\mathbf{\Lambda}_\sigma\ bd|\!)_\xi$ **for** $\sigma\ \tau\ bd\ \xi$
using *std-den-dom* **by** *blast*
next
fix $\tau\ x\ \sigma$ **and** *bd* :: ‹*'b tm*› **and** ξ :: ‹*nat* ⇒ *ty* ⇒ *'a*› **and** *a*
assume *wb*: $wff_\tau(bd\langle x^f{}_\sigma\rangle)$ **and** *xb*: $x \notin fvs\ bd$
and *r*: $\forall n\ \varrho.\ \mathcal{D}_\varrho\ (\xi\ n\ \varrho)$ **and** *da*: $\mathcal{D}_\sigma\ a$
have $(\!|\mathbf{\Lambda}_\sigma\ bd|\!)_\xi = Lm\ \sigma\ (\lambda d.\ (\!|bd\langle x^f{}_\sigma\rangle|\!)_{\xi(x_\sigma\ :=\ d)})$
by (*rule den-Abs*[*OF xb*])
moreover have $Lm\ \sigma\ (\lambda d.\ (\!|bd\langle x^f{}_\sigma\rangle|\!)_{\xi(x_\sigma\ :=\ d)})\ @\ a = (\!|bd\langle x^f{}_\sigma\rangle|\!)_{\xi(x_\sigma\ :=\ a)}$
using *r std-den-dom wb* **by** (*auto intro*!: *beta-Lm*[**where** $\tau{=}\tau$, *OF - da*] *simp*: *upd-def*)
ultimately show ‹$(\!|\mathbf{\Lambda}_\sigma\ bd|\!)_\xi\ @\ a = (\!|bd\langle x^f{}_\sigma\rangle|\!)_{\xi(x_\sigma\ :=\ a)}$› **by** *simp*
qed(*safe intro*!: *Lall Ldis Lneg Leq Jv-dom Ev-dom Iv-dom Piv-dom Dsv-dom Ngv-dom Ap-dom TF*
descB dom-nonempty)

3 The natural-deduction calculus NK

We formalise the calculus $NK_{\beta f b}$ of BKK (BKK Definition 7.1, Figures 6 and 7) — the base system NK_β together with the extensional rules $NK(f)$ and $NK(b)$ — extended by BKK's rules $NK(=_r)$ and $NK(=_l)$ for the primitive equality of our signature (BKK Figure 9, Remark 7.9) and by the description rule $NK(\iota)$, a premise-free axiom scheme describing Leibniz singletons (beyond BKK, following Andrews 1972, their reference [3]). BKK

prove $NK_{\beta fb}$ sound and complete for the class of Σ-Henkin models $\mathcal{M}_{\beta fb}$ (BKK Theorem 7.3, Theorem 7.6, Corollary 7.7), and sketch the extension by primitive equality in Remark 7.9; we prove the fully extended calculus sound and complete for the correspondingly enriched class (with primitive equality and description, Section 2). The provability judgement $\Phi \vdash A$ relates a set of sentences Φ to a sentence A. Eigenvariables are *parameters* (BKK's w_α), which — unlike the free variables used only during evaluation — never occur bound.

3.1 The inference rules of $NK_{\beta fb}$ (BKK Figures 6, 7 and 9)

Following BKK we work with the primitive constants $\neg$, $\vee$, Π_α, $=_\alpha$ and $\boldsymbol{\iota}_\alpha$; the remaining operators ($\supset$, $\bot$, Leibniz equality $\doteq$) are defined. The rule $NK(\Pi I)$ discharges an eigen-parameter w that must not occur in the context Φ or in the quantified predicate.

inductive *bprov* :: *'p tm set* ⇒ *'p tm* ⇒ *bool* (**infix** ‹⊢› *40*) **where**
  Hyp: $A \in \Phi \Longrightarrow \Phi \vdash A$ — BKK $NK(Hyp)$
| *Beta*: $A \approx_o B \Longrightarrow \Phi \vdash A \Longrightarrow \Phi \vdash B$ — BKK $NK(\beta)$
| *NegI*: $\Phi \cup \{A\} \vdash \bot \Longrightarrow \mathit{wff}_o(A) \Longrightarrow \Phi \vdash \neg A$ — BKK $NK(\neg I)$
| *NegE*: $\Phi \vdash \neg A \Longrightarrow \Phi \vdash A \Longrightarrow \mathit{wff}_o(C) \Longrightarrow \Phi \vdash C$ — BKK $NK(\neg E)$
| *DisIL*: $\Phi \vdash A \Longrightarrow \mathit{wff}_o(B) \Longrightarrow \Phi \vdash A \vee B$ — BKK $NK(\vee I_L)$
| *DisIR*: $\Phi \vdash B \Longrightarrow \mathit{wff}_o(A) \Longrightarrow \Phi \vdash A \vee B$ — BKK $NK(\vee I_R)$
| *DisE*: $\Phi \vdash A \vee B \Longrightarrow \Phi \cup \{A\} \vdash C \Longrightarrow \Phi \cup \{B\} \vdash C$
    $\Longrightarrow \mathit{wff}_o(A) \Longrightarrow \mathit{wff}_o(B) \Longrightarrow \Phi \vdash C$ — BKK $NK(\vee E)$
| *PiI*: $\Phi \vdash G \cdot (w^p \alpha) \Longrightarrow \mathit{wff}_{\alpha \Rightarrow o}(G) \Longrightarrow w \notin \mathit{pars}\ G$
    $\Longrightarrow (\forall D \in \Phi.\ w \notin \mathit{pars}\ D)$
    $\Longrightarrow \Phi \vdash (Pi\ \alpha) \cdot G$ — BKK $NK(\Pi I)$
| *PiE*: $\Phi \vdash (Pi\ \alpha) \cdot G \Longrightarrow \mathit{wff}_\alpha(A) \Longrightarrow \Phi \vdash G \cdot A$ — BKK $NK(\Pi E)$
| *Contr*: $\Phi \cup \{\neg A\} \vdash \bot \Longrightarrow \mathit{wff}_o(A) \Longrightarrow \Phi \vdash A$ — BKK $NK(Contr)$
| *FuncE*: $\Phi \vdash \mathbf{\Pi}_\alpha\ (G \cdot (Bnd\ 0) \doteq_\beta H \cdot (Bnd\ 0))$
    $\Longrightarrow \mathit{wff}_{\alpha \Rightarrow \beta}(G) \Longrightarrow \mathit{wff}_{\alpha \Rightarrow \beta}(H)$
    $\Longrightarrow \Phi \vdash G \doteq_{\alpha \Rightarrow \beta} H$ — BKK $NK(f)$
| *BoolE*: $\Phi \cup \{A\} \vdash B \Longrightarrow \Phi \cup \{B\} \vdash A \Longrightarrow \mathit{wff}_o(A) \Longrightarrow \mathit{wff}_o(B)$
    $\Longrightarrow \Phi \vdash A \doteq_o B$ — BKK $NK(b)$
| *Desc*: $\mathit{wff}_\alpha(A) \Longrightarrow \Phi \vdash (Iota\ \alpha) \cdot ((Leib\ \alpha) \cdot A) \doteq_\alpha A$
  — $NK(\iota)$, beyond BKK: Leibniz-singleton description (Andrews 1972)
| *EqR*: $\mathit{wff}_\alpha(A) \Longrightarrow \Phi \vdash A =_\alpha A$ — BKK $NK(=_r)$, Figure 9
| *EqL*: $\Phi \vdash C =_\alpha D \Longrightarrow \Phi \vdash C \doteq_\alpha D$ — BKK $NK(=_l)$, Figure 9

Provability from the empty hypothesis set, with its own turnstile.

abbreviation *provable* :: *'p tm* ⇒ *bool* (‹⊢ -› [*61*] *60*) **where**
  provable A ≡ {} ⊢ *A*

Everything derivable from a set of propositions is a proposition.

lemma *bprov-wff*: $\Phi \vdash C \Longrightarrow (\bigwedge A\ .\ A \in \Phi \Longrightarrow \mathit{wff}_o(A)) \Longrightarrow \mathit{wff}_o(C)$
proof (*induct rule*: *bprov.induct*)
  case *PiI* **thus** *?case* **by** (*meson wff-App wff-Pi*)

next
case *Desc* **thus** *?case* **by** (*metis wff-Leib wff-App wff-Iota*)
qed(*auto simp*: *beq-wff wff-App*)

Derivability is stable under injective parameter renaming — injectivity keeps the eigen-parameter side-conditions of $NK(\Pi I)$. This lets us move eigen-parameters out of the way, which is what makes weakening (and later, extension of consistent sets) admissible.

lemma *bprov-rename*: **assumes** π: *inj* π **shows** $\Phi \vdash C \Longrightarrow prn\ \pi\ `\ \Phi \vdash prn\ \pi\ C$
proof (*induction rule*: *bprov.induct*)
case *DisE* **thus** *?case* **using** *DisE bprov.DisE* **by** *auto*
next
case *PiI* **thus** *?case*
by *auto* (*smt* (*verit*) *assms bprov.PiI image-iff inj-image-mem-iff tm.set-map wff-prn*)
qed(*auto intro*: *bprov.intros*)

3.2 Weakening

Weakening is admissible. BKK leave this structural property implicit (their contexts are sets and the rules mention the context only via membership and extension); in the formalisation the eigen-parameter condition of $NK(\Pi I)$ makes it a genuine lemma, proved by transposing the eigen-parameter out of the way of the enlarged context.

Transposing two parameters — an involutive, injective renaming — lets us shift an eigen-parameter to a fresh one when weakening the context.

definition *swp* :: *'p* ⇒ *'p* ⇒ *'p* ⇒ *'p* **where**
swp a b = (λ*x*. *if x* = *a then b else if x* = *b then a else x*)
lemma *swp-inj*: *inj* (*swp a b*) **by** (*auto simp*: *swp-def inj-def*)
lemma *swp-swp* [*simp*]: *swp a b* (*swp a b x*) = *x* **by** (*auto simp*: *swp-def*)
lemma *swp-apply*: *swp a b a* = *b* **by** (*simp add*: *swp-def*)
lemma *prn-swp-swp* [*simp*]: *prn* (*swp a b*) (*prn* (*swp a b*) *t*) = *t* **by** (*simp add*: *prn-prn*)
lemma *image-prn-swp-swp* [*simp*]: *prn* (*swp a b*) ` (*prn* (*swp a b*) ` *S*) = *S*
by (*simp add*: *image-image*)

The parameters used by a context, and the property of leaving infinitely many free. *freep* is our rendering of BKK's *sufficiently* Σ*-pure* (BKK Definition 6.3): since a parameter name may be used at every type, infinitely many unused names provide, for each type, a witness reservoir of the cardinality of the (countable) language.

definition *usedp* :: *'p tm set* ⇒ *'p set* **where** *usedp* Φ ≡ (⋃ *D* ∈ Φ. *pars D*)
definition *freep* :: *'p tm set* ⇒ *bool* **where** *freep* Φ ≡ *infinite* (− *usedp* Φ)
lemma *usedp-insert*: *usedp* (*insert A* Φ) = *pars A* ∪ *usedp* Φ **by** (*auto simp*: *usedp-def*)
lemma *usedp-prn*: *usedp* (*prn* π ` Φ) = π ` *usedp* Φ **by** (*auto simp*: *usedp-def pars-prn*)

lemma *bij-swp*: *bij* (*swp a b*) **by** (*simp add*: *involuntory-imp-bij*)
lemma *infinite-inj-image*: *inj f* $\Longrightarrow$ *infinite A* $\Longrightarrow$ *infinite* (*f* ‘ *A*)
by (*metis finite-imageD inj-on-subset subset-UNIV*)
lemma *freep-add*: *freep* Φ $\Longrightarrow$ *freep* (*insert A* Φ)
by (*simp add*: *usedp-insert freep-def*)
(*metis finite-pars Diff-eq Diff-infinite-finite inf.commute*)
lemma *freep-fresh*: *freep* Φ $\Longrightarrow$ *finite F* $\Longrightarrow$ $\exists\, w.\ w \notin$ *usedp* $\Phi \wedge w \notin F$
by (*meson ComplD freep-def rev-finite-subset subsetI*)
lemma *freep-prn*: *freep* Φ $\Longrightarrow$ *freep* (*prn* (*swp a b*) ‘ Φ)
by (*metis bij-image-Compl-eq bij-swp freep-def infinite-inj-image*
swp-inj usedp-prn)

lemma *bprov-weaken*: $\Phi \vdash C \Longrightarrow \Phi \subseteq \Psi \Longrightarrow$ *freep* $\Psi \Longrightarrow \Psi \vdash C$
proof (*induction arbitrary*: Ψ *rule*: *bprov.induct*)
case *NegI* **thus** *?case*
by (*simp add*: *bprov.NegI freep-add sup.order-iff*)
next
case *DisE* **thus** *?case*
by (*metis Un-insert-right bprov.DisE freep-add sup.cobounded2*
sup.order-iff sup-bot-right)
next
case (*PiI* Φ *G w* α)
then obtain w' **where** w': $w' \notin$ *usedp* Ψ $w' \notin$ *pars G* $w' \neq w$
using *freep-fresh*[*of* - *pars G* $\cup$ {*w*}] **by** *force*
have *wsub*: $\Phi \subseteq$ *prn* (*swp w w'*) ‘ Ψ
proof
fix *D* **assume** *D*: $D \in \Phi$
hence $w \notin$ *pars D* **and** $w' \notin$ *pars D*
by (*simp add*: *PiI.hyps*(*4*)) (*metis D PiI.prems*(*1*) *UN-I subset-iff usedp-def*
w'(*1*))
hence *prn* (*swp w w'*) *D* = *D* **by** (*auto simp*: *swp-def intro*: *prn-cong*)
thus *D* $\in$ *prn* (*swp w w'*) ‘ Ψ **using** *D PiI.prems*(*1*) **by** *force*
qed
have *prn* (*swp w w'*) ‘ $\Psi \vdash G \cdot ({w^p}_\alpha)$
using *PiI freep-prn wsub* **by** *meson*
hence $\Psi \vdash$ (*prn* (*swp w w'*) *G*) $\cdot$ $({w'^p}_\alpha)$
by (*metis bprov-rename image-prn-swp-swp swp-inj tm.simps*(*121*,*127*) *swp-def*)
moreover have *prn* (*swp w w'*) *G* = *G*
using *PiI.hyps*(*3*) *w'*(*2*)
by (*auto simp*: *swp-def intro*: *prn-cong*)
ultimately have $\Psi \vdash G \cdot ({w'^p}_\alpha)$ **by** *simp*
moreover have $\forall\, D \in \Psi.\ w' \notin$ *pars D* **using** *w'*(*1*)
by (*auto simp*: *usedp-def*)
ultimately show *?case* **using** *PiI.hyps*(*2*) *w'*(*2*)
by (*auto intro*: *bprov.PiI*)
next case *Contr* **thus** *?case*
by (*metis Un-insert-right bprov.Contr freep-add sup.cobounded2*
sup.order-iff sup-bot.right-neutral)
next case *BoolE* **thus** *?case* **by** (*simp add*: *bprov.BoolE freep-add sup.absorb-iff2*)

qed(*auto intro*: *bprov.intros*)

3.3 Compactness

Every derivation uses only finitely many hypotheses — BKK: "since every NK^*-proof is finite" (used in the proof of BKK Corollary 7.8). With set-based contexts this is again a genuine lemma.

When the parameter type is infinite, every finite context leaves infinitely many parameters free, and every derivation uses only a finite part of its context.

lemma *freep-finite*: **fixes** Φ :: *'p::infinite tm set* **assumes** *finite* Φ
shows *freep* Φ
by (*metis assms freep-def usedp-def finite-pars infinite-UNIV finite-Diff2 Compl-eq-Diff-UNIV finite-UN*)

lemma *bprov-finite*: $\Phi \vdash (C :: 'p::infinite\ tm) \Longrightarrow \exists \Phi 0.\ finite\ \Phi 0 \wedge \Phi 0 \subseteq \Phi \wedge \Phi 0 \vdash C$
proof (*induction rule*: *bprov.induct*)
case (*NegI* Φ *A*) **thus** *?case*
by (*smt* (*verit*) *Un-infinite Un-insert-right bprov.NegI bprov-weaken freep-add freep-finite subset-UnE subset-insertI subset-singleton-iff*)
next
case (*NegE* Φ *A C*) **thus** *?case*
by (*smt* (*verit*) *bprov.NegE bprov-weaken finite-UnI freep-finite le-sup-iff sup-ge1 sup-ge2*)
next
case (*DisE* Φ *A B C*)
then obtain $\Phi 1$ $\Phi 2$ $\Phi 3$ **where**
Q1: *finite* $\Phi 1$ $\Phi 1 \subseteq \Phi$ $\Phi 1 \vdash A \vee B$ **and**
Q2: *finite* $\Phi 2$ $\Phi 2 \subseteq \Phi \cup \{A\}$ $\Phi 2 \vdash C$ **and**
Q3: *finite* $\Phi 3$ $\Phi 3 \subseteq \Phi \cup \{B\}$ $\Phi 3 \vdash C$
by *blast*
moreover define $\Phi 0$ **where** $\Phi 0 = \Phi 1 \cup (\Phi 2 - \{A\}) \cup (\Phi 3 - \{B\})$
ultimately have *finite* $\Phi 0$ **and** $\Phi 0 \subseteq \Phi$ **by** *auto*
moreover {
have $\Phi 0 \vdash A \vee B$
by (*metis Q1*(*3*) *Φ0-def bprov-weaken dual-order.refl calculation*(*1*) *freep-finite le-sup-iff*)
moreover have $\Phi 0 \cup \{A\} \vdash C$ **and** $\Phi 0 \cup \{B\} \vdash C$
by (*auto intro*: *bprov-weaken*[*OF Q2*(*3*)] *bprov-weaken*[*OF Q3*(*3*)]
simp: *Φ0-def Q1*(*1*) *Q2*(*1*) *Q3*(*1*) *freep-finite*)
ultimately have $\Phi 0 \vdash C$ **by** (*auto intro*: *DisE bprov.DisE*)
}
ultimately show *?case* **by** *blast*
next case (*Contr* Φ *A*) **show** *?case* **by** (*smt* (*verit*, *del-insts*) *Contr.IH Contr.hyps*(*2*) *Un-infinite Un-insert-right bprov.Contr bprov-weaken freep-add freep-finite subset-UnE*

subset-insertI subset-singleton-iff)
next case (*BoolE* Φ *A B*)
then obtain $\Phi 1$ $\Phi 2$ **where**
Q1: *finite* $\Phi 1$ $\Phi 1 \subseteq \Phi \cup \{A\}$ $\Phi 1 \vdash B$ **and**
Q2: *finite* $\Phi 2$ $\Phi 2 \subseteq \Phi \cup \{B\}$ $\Phi 2 \vdash A$
by *blast*
moreover define $\Phi 0$ **where** $\Phi 0 = (\Phi 1 - \{A\}) \cup (\Phi 2 - \{B\})$
ultimately have *finite* $\Phi 0$ **and** $\Phi 0 \subseteq \Phi$ **by** *auto*
moreover {
have $\Phi 0 \cup \{A\} \vdash B$ **and** $\Phi 0 \cup \{B\} \vdash A$
by (*auto intro*: *bprov-weaken*[*OF Q1*(*3*)] *bprov-weaken*[*OF Q2*(*3*)]
simp: *calculation Q1*(*1*,*2*) *Q2*(*1*) $\Phi 0$-*def freep-finite*)
hence $\Phi 0 \vdash A \doteq_{\mathrm{o}} B$ **using** *BoolE* **by** (*auto intro*: *bprov.intros*)
}
ultimately show *?case* **by** *blast*
qed(*auto intro*: *bprov.intros*)

4 Soundness

This section proves *NK* sound for the model class $\mathcal{M}_{\beta fb}$ (BKK Theorem 7.3, including BKK's evaluation-variant argument).

4.1 The canonical construction respects β-conversion (BKK Remark 3.19)

In a Σ-Henkin model, β-convertible terms denote the same object under any total assignment; the abstraction-congruence case uses functionality (property f).

context *general-model*
begin

lemma *beq-den*: $s \approx_{\varrho} t \Longrightarrow (\forall\, n\ \tau.\ \mathcal{D}_\tau\ (\xi\ n\ \tau)) \Longrightarrow (\!|s|\!)_\xi = (\!|t|\!)_\xi$
proof (*induction arbitrary*: ξ *rule*: *beq.induct*)
case *beta* **thus** *?case* **using** *den-App-Abs resp-def* **by** *auto*
next
case (*abs L b* σ τ *b'*)
define *x* **where** *x* = *fresh* (*L* $\cup$ *fvs b* $\cup$ *fvs b'*)
have *xL*: $x \notin L$ **and** *xb*: $x \notin$ *fvs b* **and** *xb'*: $x \notin$ *fvs b'*
using *fresh-notin*[*of L* $\cup$ *fvs b* $\cup$ *fvs b'*] *abs* **unfolding** *x-def* **by** *auto*
have *wb*: $\mathit{wff}_\tau(b\langle x^f{}_\sigma\rangle)$ **and** *wb'*: $\mathit{wff}_\tau(b'\langle x^f{}_\sigma\rangle)$
using *beq-wffL beq-wffR abs xL* **by** *blast*+
{
fix *a*
assume *a*: $\mathcal{D}_\sigma\ a$
hence *tot*: $\forall\, n\ \gamma.\ \mathcal{D}_\gamma\ (\xi(x_\sigma := a)\ n\ \gamma)$
using *abs.prems* **by** (*auto simp*: *upd-def*)

```
    have (|Λσ b|)ξ @ a = (|b⟨x^f σ⟩|)ξ(xσ := a)
      by (rule gm-beta[OF wb xb abs.prems a])
    also have ... = (|b'⟨x^f σ⟩|)ξ(xσ := a)
      using abs.IH[OF xL, of ξ(xσ := a)] tot by blast
    also have ... = (|Λσ b'|)ξ @ a
      using gm-beta wb' xb' abs.prems a by simp
    finally have (|Λσ b|)ξ @ a = (|Λσ b'|)ξ @ a.
  }
  thus ?case
    using gm-funct gm-lamTy wff-Abs-open-rev wb xb abs.prems wb' xb' by blast
qed(auto simp: den.simps(8))

end
```

4.2 Abstract soundness (BKK Theorem 7.3)

Soundness over the abstract Σ-models of Section 2, following BKK's proof of Theorem 7.3 case by case. The $NK(\Pi I)$ case uses BKK's device verbatim: "from the evaluation function E, one can define another evaluation function E' such that $E'(w) \equiv a$ and $E'_\varphi(A) \equiv E_\varphi(A)$ if w does not occur in A" — realised below by reading the parameter as a fresh variable after an injective shift of all free variables. The extensionality cases $NK(f)$ and $NK(b)$ rest on BKK's Lemma 4.2 on Leibniz equality, proven here abstractly (BKK themselves route them through Theorem 4.3(3) and Lemma 3.48/Theorem 4.3(4), which rest on Lemma 4.2).

4.2.1 The variable shift

vshift renames every free variable n to $n + 1$, freeing the name *0* at every type; β-equality and typing are stable under it and under the parameter-to-variable rename *pvar*.

```
definition vshift :: 'p tm ⇒ 'p tm where vshift = msub (λn τ. (Suc n)^f τ)
lemma occ-vshift: occ (vshift t) = (λ(n, τ). (Suc n, τ)) ‘ occ t
  unfolding vshift-def by (induction t) (auto simp: image-Un)
lemma pars-vshift: pars (vshift t) = pars t
  unfolding vshift-def by (induction t) auto
lemma wff-vshift: wff_τ(t) ⟹ wff_τ(vshift t)
  by (auto elim: wff-msub intro: wff-Fre simp: vshift-def)
lemma vshift-opn: vshift (t⟨x^f σ⟩) = (vshift t)⟨(Suc x)^f σ⟩
  unfolding vshift-def by (subst msub-opn) auto
lemma beq-vshift: s ≈ϱ t ⟹ vshift s ≈ϱ vshift t
proof (induction rule: beq.induct)
  case (beta σ ϱ b a)
  moreover have vshift ((Λσ b) · a) = (Λσ (vshift b)) · (vshift a)
    by (simp add: vshift-def)
  moreover have vshift (b⟨a⟩) = (vshift b)⟨vshift a⟩
    unfolding vshift-def by (subst msub-opn) auto
```

moreover have $wff_{\sigma\Rightarrow\varrho}(\mathbf{\Lambda}_\sigma\ (vshift\ b))$
using *wff-vshift beta* **unfolding** *vshift-def* **by** *force*
ultimately show *?case* **using** *beq.beta wff-vshift* **by** *metis*
next case *appL* **thus** *?case* **by** (*metis beq.appL msub.simps*(*9*) *vshift-def wff-vshift*)
next case *appR* **thus** *?case* **by** (*metis beq.appR msub.simps*(*9*) *vshift-def wff-vshift*)
next case (*abs L b* σ τ b')
have $\mathbf{\Lambda}_\sigma\ (vshift\ b) \approx_{\sigma\Rightarrow\tau} \mathbf{\Lambda}_\sigma\ (vshift\ b')$
proof (*rule beq.abs*[*of* $\{0\} \cup Suc\ `\ L$])
show *finite* ($\{0\} \cup Suc\ `\ L$) **using** *abs.hyps*(*1*) **by** *simp*
fix *y* **assume** *y*: $y \notin \{0\} \cup Suc\ `\ L$
then obtain *x* **where** *x*: $y = Suc\ x\ x \notin L$ **by** (*cases y*) *auto*
show $(vshift\ b)\langle {y^f}_\sigma\rangle \approx_\tau (vshift\ b')\langle {y^f}_\sigma\rangle$ **using** *abs.IH*[*OF x*(*2*)]
by (*simp add*: *x*(*1*) *vshift-opn*[*symmetric*])
qed
thus *?case* **by** (*simp add*: *vshift-def*)
qed(*auto simp*: *wff-vshift intro*: *beq.intros*)

lemma *beq-pvar*: $s \approx_\varrho t \Longrightarrow pvar\ w\ \sigma w\ x\ s \approx_\varrho pvar\ w\ \sigma w\ x\ t$
proof (*induction rule*: *beq.induct*)
case *beta* **thus** *?case* **by** (*smt* (*verit*, *best*) *beq.simps pvar.simps*(*10*,*9*) *pvar-opn wff-pvar*)
qed(*auto simp*: *wff-pvar beq.abs pvar-opn intro*: *beq.intros*)

lemma *occ-pvar*: $occ\ (pvar\ w\ \sigma w\ x\ t) \subseteq occ\ t \cup \{(x, \sigma w)\}$
by (*induction t*) *auto*

4.2.2 Satisfaction of Leibniz equality (BKK Lemma 4.2)

Leibniz equality β-reduces to its $\forall$-form (the syntactic prelude to Lemma 4.2).

lemma *Leib-beq*: **assumes** *wA*: $wff_\alpha(A)$ **and** *wB*: $wff_\alpha(B)$
shows $(A \doteq_\alpha B) \approx_o \mathbf{\Pi}_{\alpha\Rightarrow o}\ ((Bnd\ 0 \cdot A) \supset (Bnd\ 0 \cdot B))$
proof −
have *lcA*: *lc A* **and** *lcB*: *lc B* **using** *wA wB* **by** (*auto intro*: *wff-lc*)
let *?inner* = $\mathbf{\Lambda}_\alpha\ (\mathbf{\Pi}_{\alpha\Rightarrow o}\ ((Bnd\ 0 \cdot A) \supset (Bnd\ 0 \cdot Bnd\ 1)))$
have $(Leib\ \alpha \cdot A) \approx_{\alpha\Rightarrow o}$ *?inner*
using *beq.beta*[*OF wff-Leib*[*unfolded Leib-def*] *wA*]
unfolding *Leib-def* **by** (*simp add*: *opn-lc*[*OF lcA*])
moreover {
have $wff_{\alpha\Rightarrow o}$(*?inner*) **using** *beq-wffR calculation* **by** *blast*
with *beq.beta*[*OF this wB*]
have $(?inner \cdot B) \approx_o \mathbf{\Pi}_{\alpha\Rightarrow o}\ ((Bnd\ 0 \cdot A) \supset (Bnd\ 0 \cdot B))$
using *opn-lc*[*OF lcA*] *opn-lc*[*OF lcB*] **by** *simp*
}
ultimately show *?thesis* **using** *beq.trans beq.appL wB* **by** *blast*
qed

context *sigma-model*
begin

lemma *sat-Leib*: **assumes** *wA*: $wff_\alpha(A)$ **and** *wB*: $wff_\alpha(B)$ **and** *xi*: *asg* ξ
shows *vl* (*Ee* ξ ($A \doteq_\alpha B$)) $\longleftrightarrow$ *Ee* ξ *A* = *Ee* ξ *B*
proof −
have *lcA*: *lc A* **and** *lcB*: *lc B* **using** *wA wB* **by** (*auto intro*: *wff-lc*)
define *p* **where** *p* = *fresh* (*fvs A* ∪ *fvs B*)
have *p*: $p \notin$ *fvs A* $p \notin$ *fvs B*
unfolding *p-def* **using** *fresh-notin*[*of fvs A* ∪ *fvs B*] **by** *auto*
let *?b* = (*Bnd 0* · *A*) ⊃ (*Bnd 0* · *B*)
have *wI*: $wff_{(\alpha \Rightarrow o) \Rightarrow o}(\mathbf{\Lambda}\alpha \Rightarrow o\ ?b)$
by (*smt* (*verit*, *del-insts*) *ImpB-def lcA lcB opn.simps*(*1*,*4*,*5*,*9*) *opn-lc wA wB*
wff-AbsI wff-App wff-Fre wff-ImpB)
have *pf*: $p \notin$ *fvs ?b* **using** *p* **by** (*auto simp*: *ImpB-def*)
have *unf*: *vl* (*Ee* ξ ($A \doteq_\alpha B$)) $\longleftrightarrow$ (∀ *r*. *Dm* ($\alpha \Rightarrow o$) *r*
$\longrightarrow$ (*vl* (*Ap r* (*Ee* ξ *A*)) $\longrightarrow$ *vl* (*Ap r* (*Ee* ξ *B*))))
proof −
have *inner*: *vl* (*Ee* ($\xi(p_\alpha \Rightarrow o := r)$) (*?b*⟨$p^f\alpha \Rightarrow o$⟩))
$\longleftrightarrow$ (*vl* (*Ap r* (*Ee* ξ *A*)) $\longrightarrow$ *vl* (*Ap r* (*Ee* ξ *B*)))
if *r*: *Dm* ($\alpha \Rightarrow o$) *r* **for** *r*
proof −
let *?ξ* = $\xi(p_\alpha \Rightarrow o := r)$
have *ob*: *?b*⟨$p^f\alpha \Rightarrow o$⟩ = ($p^f\alpha \Rightarrow o$ · *A*) ⊃ ($p^f\alpha \Rightarrow o$ · *B*)
by (*simp add*: *opn-lc*[*OF lcA*] *opn-lc*[*OF lcB*])
have *cA*: *Ee ?ξ A* = *Ee* ξ *A*
using *p ev-coin*[*OF wA asg-upd*[*OF xi r*] *xi*]
unfolding *upd-def fvs-eq-fst-occ image-iff* **by** *force*
have *cB*: *Ee ?ξ B* = *Ee* ξ *B*
using *p ev-coin*[*OF wB asg-upd*[*OF xi r*] *xi*]
unfolding *upd-def fvs-eq-fst-occ image-iff* **by** *force*
show *?thesis* **unfolding** *ob*
by (*smt* (*verit*, *del-insts*) *asg-upd cA cB ev-app ev-var sat-ImpB that upd-same wA wB*
wff-App wff-AppE wff-App-FreFre wff-FreE xi)
qed
show *?thesis*
unfolding *ev-beta*[*OF Leib-beq*[*OF wA wB*] *xi*] *sat-Forall*[*OF wI pf xi*]
using *inner* **by** *blast*
qed
show *?thesis*
proof
assume *L*: *vl* (*Ee* ξ ($A \doteq_\alpha B$))
let *?r* = *Ap* (*Ee* ξ (*Eq* α)) (*Ee* ξ *A*)
have *rA*: *vl* (*Ap ?r* (*Ee* ξ *A*))
using *vl-eq*[*OF xi ev-type*[*OF wA xi*] *ev-type*[*OF wA xi*]] **by** *simp*
have *vl* (*Ap ?r* (*Ee* ξ *B*))
using *unf L as-appTy*[*OF ev-type*[*OF wff-Eq xi*] *ev-type*[*OF wA xi*]] *rA* **by** *blast*
thus *Ee* ξ *A* = *Ee* ξ *B*
using *vl-eq*[*OF xi ev-type*[*OF wA xi*] *ev-type*[*OF wB xi*]] **by** *simp*

qed(*simp add*: *unf*)
qed

end

4.2.3 The evaluation variant at a parameter

context *bkk-model*
begin

Agreement under the variable shift: shifting all free variables and the assignment in step leaves denotations unchanged (property f resolves the abstraction case applicatively).

lemma *Ee-vshift*:
assumes ‹$wff_\tau(A)$› ‹*asg* ξ› ‹*asg* ξ'› ‹$\bigwedge n\ \tau'.\ (n, \tau') \in occ\ A \Longrightarrow \xi'\ (Suc\ n)\ \tau' = \xi\ n\ \tau'$›
shows ‹*Ee* ξ' (*vshift A*) = *Ee* ξ *A*›
using *assms* **proof** (*induction A arbitrary*: τ ξ ξ' *rule*: *size-induct*)
case (*App s u*)
then obtain σ **where** *ws*: $wff_{\sigma \Rightarrow \tau}(s)$ **and** *wu*: $wff_\sigma(u)$
using *App* **by** *auto*
have *IHs*: *Ee* ξ' (*vshift s*) = *Ee* ξ *s*
using *App wf* **by** *auto*
have *IHu*: *Ee* ξ' (*vshift u*) = *Ee* ξ *u*
using *App* **by** *auto*
have *Ee* ξ' (*vshift* (*s* · *u*)) = *Ee* ξ' (*vshift s* · *vshift u*)
by (*simp add*: *vshift-def*)
also have ... = *Ap* (*Ee* ξ' (*vshift s*)) (*Ee* ξ' (*vshift u*))
by (*meson ev-app App.prems*(3) *wff-vshift ws wu*)
also have ... = *Ap* (*Ee* ξ *s*) (*Ee* ξ *u*) **by** (*simp add*: *IHs IHu*)
also have ... = *Ee* ξ (*s* · *u*)
by (*simp add*: *ev-app*[*OF ws wu App.prems*(2)])
finally show *?case* **using** *App* **by** *simp*
next
case (*Abs* σ *b*)
then obtain τ' **where** *t*: $\tau = \sigma \Rightarrow \tau'$ **and** *wA*: $wff_{\sigma \Rightarrow \tau'}(\mathbf{\Lambda}_\sigma\ b)$
by (*metis wff-AbsE*)
have *wS*: $wff_{\sigma \Rightarrow \tau'}(\mathbf{\Lambda}_\sigma\ (vshift\ b))$
using *wff-vshift*[*OF wA*] **by** (*simp add*: *vshift-def*)
define *y* **where** *y* = *fresh* (*fvs b*)
have *y*: *y* ∉ *fvs b* **unfolding** *y-def* **by** (*simp add*: *fresh-notin*)
have *y'*: *Suc y* ∉ *fvs* (*vshift b*)
using *y* **by** (*force simp*: *occ-vshift fvs-eq-fst-occ*)
{
fix *d*
assume *d*: *Dm* σ *d*
have *IH*: *Ee* (ξ'(*Suc* y_σ := *d*)) (*vshift* ($b\langle y^f{}_\sigma\rangle$)) = *Ee* ($\xi$($y_\sigma$:= *d*)) ($b\langle y^f{}_\sigma\rangle$)
proof (*rule Abs.IH*)
show *size* ($b\langle y^f{}_\sigma\rangle$) ≤ *size b* **using** *Abs* **by** *simp*

```
    next
      fix n τ''
      assume o: (n, τ'') ∈ occ (b⟨y^f_σ⟩)
      hence (n, τ'') ∈ occ b ∨ (n, τ'') = (y, σ)
          using occ-opn[of 0 y^f_σ b] by auto
      thus (ξ'(Suc y_σ := d)) (Suc n) τ'' = (ξ(y_σ := d)) n τ''
        using Abs by (auto simp: upd-def)
    qed(auto intro: wff-Abs-open[OF wA y] asg-upd[OF Abs.prems(2) d] asg-upd[OF
Abs.prems(3) d])
    hence Ap (Ee ξ' (Λ_σ (vshift b))) d = Ap (Ee ξ (Λ_σ b)) d
    using ev-abs-app[OF wS Abs.prems(3) y', OF d] ev-abs-app[OF wA Abs.prems(2)
y, OF d]
      by (simp add: vshift-opn)
  }
  hence Ee ξ' (Λ_σ (vshift b)) = Ee ξ (Λ_σ b)
      using prop-f ev-type[OF wS Abs.prems(3)] ev-type[OF wA
      Abs.prems(2)] unfolding functional-def by blast
  thus ?case using Abs by (simp add: vshift-def)
qed(auto simp: Ee-closed vshift-def ev-var)
```

BKK's E' (the $NK(\Pi I)$ case of Theorem 7.3): the parameter w is read as the freshly freed variable *0*, assigned a.

```
definition upshift :: (nat ⇒ ty ⇒ 'u) ⇒ ty ⇒ 'u ⇒ nat ⇒ ty ⇒ 'u where
  upshift ξ σ a = (λn τ. case n of 0 ⇒ (if τ = σ then a else ξ 0 τ) | Suc m ⇒ ξ
m τ)
definition Evar :: 'p ⇒ ty ⇒ 'u ⇒ (nat ⇒ ty ⇒ 'u) ⇒ 'p tm ⇒ 'u where
  Evar w σ a ξ A = Ee (upshift ξ σ a) (pvar w σ 0 (vshift A))
lemma asg-upshift: asg ξ ⟹ Dm σ a ⟹ asg (upshift ξ σ a)
  by (auto simp: asg-def upshift-def split: nat.splits)
lemma Evar-par: asg ξ ⟹ Dm σ a ⟹ Evar w σ a ξ (w^p_σ) = a
  by (smt (verit, del-insts) Evar-def asg-upshift bkk-model.upshift-def
    bkk-model-axioms ev-var msub.simps(3) old.nat.simps(4) pvar.simps(3) vshift-def)
lemma Evar-agree: wff_τ(A) ⟹ asg ξ ⟹ Dm σ a ⟹ w ∉ pars A ⟹ Evar w
σ a ξ A = Ee ξ A
  using Evar-def asg-upshift bkk-model.Ee-vshift bkk-model-axioms
      pars-vshift upshift-def by fastforce
lemma Evar-bkk: assumes a: Dm σ a shows bkk-model Dm Ap (Evar w σ a) vl
proof (unfold-locales, goal-cases)
  case 1 thus ?case
    by (simp add: Evar-def asg-upshift assms ev-type wff-pvar wff-vshift)
next
  case 2 thus ?case
    by (metis Evar-agree assms emptyE ev-var tm.simps(230) wff-Fre)
next
  case 3 thus ?case
    by (simp add: Evar-def vshift-def)
      (metis asg-upshift assms ev-app vshift-def wff-pvar wff-vshift)
next
  case (4 τ A ξ ξ')
```

```
  have ag: upshift ξ σ a n τ' = upshift ξ' σ a n τ'
    if o: (n, τ') ∈ occ (pvar w σ 0 (vshift A)) for n τ'
  proof −
    from o have (n, τ') ∈ occ (vshift A) ∪ {(0, σ)}
        using occ-pvar[of w σ 0 vshift A] by blast
    then consider (sh) m where n = Suc m (m, τ') ∈ occ A |
        (zero) n = 0 τ' = σ by (auto simp: occ-vshift)
    thus ?thesis using 4(4) upshift-def by fastforce
  qed
  show ?case unfolding Evar-def
    by (rule ev-coin[OF wff-pvar[OF wff-vshift[OF 4(1)]]
        asg-upshift[OF 4(2) a] asg-upshift[OF 4(3) a] ag])
next
  case 5 thus ?case
    by (metis (full-types) Evar-def asg-upshift assms beq-pvar beq-vshift ev-beta)
qed(auto simp: Evar-def asg-upshift assms vl-eq vl-pi vl-dis vl-neg vl-iota
              vshift-def prop-b prop-f)

end
```

4.2.4 Soundness

BKK Theorem 7.3, for the class $\mathcal{M}_{\beta fb}$ (with primitive equality and description): each case mirrors BKK's proof text.

```
theorem soundness-bkk:
  assumes Φ ⊢ C bkk-model Dm Ap Ee vl app-struct.asg Dm ξ
        ∀ A ∈ Φ. wff_o(A) ∀ A ∈ Φ. vl (Ee ξ A)
    shows vl (Ee ξ C)
using assms proof (induction arbitrary: Dm Ap Ee vl ξ rule: bprov.induct)
  case Hyp thus ?case by blast
next
  case Beta thus ?case
    by (metis bkk-model-def sigma-eval.ev-beta sigma-model.axioms(1))
next
  case NegI thus ?case
     by (metis UnE bkk-model-def sigma-model.sat-Neg sigma-model.vl-TF singleton-iff)
next
  case NegE thus ?case
    using bkk-model.axioms(1) bprov-wff sigma-model.sat-Neg by fastforce
next
  case DisIL thus ?case
    by (metis Hyp bprov-wff bkk-model.axioms(1) sigma-model.sat-Dis)
next
  case DisIR thus ?case
    by (metis bprov-wff sigma-model.sat-Dis bkk-model.axioms(1))
next
  case DisE thus ?case
    by (metis UnE bkk-model.axioms(1) sigma-model.sat-Dis singleton-iff)
```

next
case (*PiI* Φ G w α)
interpret *M*: *bkk-model Dm Ap Ee vl* **by** (*rule PiI.prems*(*1*))
have *wGw*: *wff*$_{\mathrm{o}}$($G \cdot (w^p{}_\alpha)$)
using *bprov-wff*[*OF PiI.hyps*(*1*)] *PiI.prems*(*3*) **by** *blast*
have *vl* (*Ap* (*Ee* ξ G) *a*) **if** *a*: *Dm* α *a* **for** *a*
proof −
let *?E* = *M.Evar w* α *a*
interpret *V*: *bkk-model Dm Ap ?E vl* **by** (*rule M.Evar-bkk*[*OF a*])
have *sat*: $\forall A \in \Phi$. *vl* (*?E* ξ *A*) **using** *PiI.prems*(*3*,*4*) *PiI.hyps*(*4*)
by (*auto simp*: *M.Evar-agree*[*OF* - *PiI.prems*(*2*) *a*])
have *vl* (*?E* ξ ($G \cdot (w^p{}_\alpha)$)) **by** (*rule PiI.IH*[*OF M.Evar-bkk*[*OF a*]
PiI.prems(*2*) *PiI.prems*(*3*) *sat*])
moreover have *?E* ξ ($G \cdot (w^p{}_\alpha)$) = *Ap* (*Ee* ξ G) *a*
by (*simp add*: *V.ev-app*[*OF PiI.hyps*(*2*) *wff-Par PiI.prems*(*2*)]
M.Evar-agree[*OF PiI.hyps*(*2*) *PiI.prems*(*2*) *a PiI.hyps*(*3*)]
M.Evar-par[*OF PiI.prems*(*2*) *a*])
ultimately show *?thesis* **by** *simp*
qed
thus *?case* **by** (*simp add*: *M.sat-Pi*[*OF PiI.hyps*(*2*) *PiI.prems*(*2*)])
next
case (*PiE* Φ α G *A*)
interpret *M*: *bkk-model Dm Ap Ee vl* **by** (*rule PiE.prems*(*1*))
have *wPiG*: *wff*$_{\mathrm{o}}$(*Pi* $\alpha \cdot G$) **using** *bprov-wff*[*OF PiE.hyps*(*1*)]
PiE.prems(*3*) **by** *blast*
have *wG*: *wff*$_{\alpha \Rightarrow \mathrm{o}}$($G$) **using** *wPiG* **by** (*auto dest*: *wff-unique*)
have *vl* (*Ap* (*Ee* ξ G) (*Ee* ξ *A*)) **using** *PiE.IH*[*OF PiE.prems*]
M.sat-Pi[*OF wG PiE.prems*(*2*)]
M.ev-type[*OF PiE.hyps*(*2*) *PiE.prems*(*2*)] **by** *simp*
thus *?case* **by** (*simp add*: *M.ev-app*[*OF wG PiE.hyps*(*2*) *PiE.prems*(*2*)])
next
case *Contr* **thus** *?case*
using *bkk-model.axioms*(*1*) *sigma-model.sat-Neg sigma-model.vl-TF* **by** *fast-force*
next
case (*FuncE* Φ α G β H)
interpret *M*: *bkk-model Dm Ap Ee vl* **by** (*rule FuncE.prems*(*1*))
have *lcG*: *lc G* **and** *lcH*: *lc H* **using** *FuncE.hyps*(*2*,*3*)
by (*auto intro*: *wff-lc*)
define *y* **where** *y* = *fresh* (*fvs G* ∪ *fvs H*)
have *y*: *y* ∉ *fvs G y* ∉ *fvs H* **unfolding** *y-def*
using *fresh-notin*[*of fvs G* ∪ *fvs H*] **by** *auto*
let *?b* = $G \cdot$ *Bnd 0* $\doteq_\beta$ $H \cdot$ *Bnd 0*
have *wI*: *wff*$_{\alpha \Rightarrow \mathrm{o}}$($\mathbf{\Lambda}_\alpha$ *?b*)
by (*auto intro*!: *wff-AbsI wff-App*[*OF FuncE.hyps*(*2*) *wff-Fre*] *wff-App*[*OF FuncE.hyps*(*3*) *wff-Fre*]
simp: *opn-lc*[*OF lcG*] *opn-lc*[*OF lcH*])
have *yb*: *y* ∉ *fvs ?b* **using** *y* **by** (*auto simp*: *Leib-def Forall-def ImpB-def*)
have *pointwise*: *Ap* (*Ee* ξ G) *d* = *Ap* (*Ee* ξ H) *d* **if** *d*: *Dm* α *d* **for** *d*

proof −
let *?ξ* = *ξ*(*y*$_\alpha$:= *d*)
have *vl* (*Ee ξ* (**Π**$_\alpha$ *?b*)) **using** *FuncE.IH*[*OF FuncE.prems*] **.**
hence *vl* (*Ee ?ξ* (*?b*⟨*y*$^f{}_\alpha$⟩))
using *FuncE.prems*(*2*) *M.sat-Forall that wI yb* **by** *blast*
hence *vl* (*Ee ?ξ* (*G* · (*y*$^f{}_\alpha$) $\doteq_\beta$ *H* · (*y*$^f{}_\alpha$)))
by (*simp add*: *opn-lc*[*OF lcG*] *opn-lc*[*OF lcH*])
hence *Ee ?ξ* (*G* · (*y*$^f{}_\alpha$)) = *Ee ?ξ* (*H* · (*y*$^f{}_\alpha$))
by (*meson FuncE.hyps*(*2*,*3*) *FuncE.prems*(*2*) *M.sat-Leib M.sigma-model-axioms app-struct.asg-upd*
sigma-eval-def sigma-model-def that wff-App wff-Fre)
moreover have *Ee ?ξ G* = *Ee ξ G*
by (*metis* (*mono-tags*, *lifting*) *FuncE.hyps*(*2*) *FuncE.prems*(*2*) *M.asg-upd M.ev-coin fst-conv*
fvs-eq-fst-occ image-eqI that upd-def y(*1*))
moreover have *Ee ?ξ H* = *Ee ξ H*
by (*metis* (*mono-tags*, *lifting*) *FuncE.hyps*(*3*) *FuncE.prems*(*2*) *M.asg-upd M.ev-coin fst-conv*
fvs-eq-fst-occ image-eqI that upd-def y(*2*))
ultimately show *?thesis*
by (*metis FuncE.hyps*(*2*,*3*) *FuncE.prems*(*2*) *M.asg-upd M.ev-app M.ev-var that upd-same wff-Fre*)
qed
have *Ee ξ G* = *Ee ξ H*
using *FuncE.hyps*(*2*,*3*) *FuncE.prems*(*2*) *M.ev-type M.functional-def M.prop-f pointwise* **by** *blast*
thus *?case* **by** (*simp add*: *M.sat-Leib*[*OF FuncE.hyps*(*2*,*3*) *FuncE.prems*(*2*)])
next
case (*BoolE* Φ *A B*)
interpret *M*: *bkk-model Dm Ap Ee vl* **by** (*rule BoolE.prems*(*1*))
have *vl* (*Ee ξ A*) ⟷ *vl* (*Ee ξ B*)
using *BoolE* **by** *fast*
hence *Ee ξ A* = *Ee ξ B*
by (*simp add*: *BoolE.hyps*(*3*,*4*) *BoolE.prems*(*2*) *M.ev-type M.prop-b*)
thus *?case*
by (*simp add*: *M.sat-Leib*[*OF BoolE.hyps*(*3*,*4*) *BoolE.prems*(*2*)])
next case (*Desc α A* Φ)
interpret *M*: *bkk-model Dm Ap Ee vl* **by** (*rule Desc.prems*(*1*))
define *y* **where** *y* = *fresh* (*fvs A*)
have *y*: *y* ∉ *fvs A* **unfolding** *y-def* **by** (*simp add*: *fresh-notin*)
let *?f* = *Ee ξ* (*Leib α* · *A*)
have *sing*: *vl* (*Ap ?f b*) ⟷ *b* = *Ee ξ A* **if** *b*: *Dm α b* **for** *b*
proof −
let *?ξ* = *ξ*(*y*$_\alpha$:= *b*)
have *c*: *Ee ?ξ* (*Leib α* · *A*) = *?f*
using *y*
by (*safe intro*!: *M.ev-coin*[*OF wff-App*[*OF wff-Leib Desc.hyps*]
M.asg-upd[*OF Desc.prems*(*2*) *b*] *Desc.prems*(*2*)])
(*auto simp add*: *Forall-def ImpB-def Leib-def upd-def fvs-eq-fst-occ image-iff*)

```
    have cA: Ee ?ξ A = Ee ξ A
     by (metis (mono-tags, lifting) Desc.hyps Desc.prems(2) M.asg-upd M.ev-coin
fst-conv
              fvs-eq-fst-occ image-eqI that upd-def y)
    have Ap ?f b = Ee ?ξ (A ≐α (y^f α))
     by (simp add: M.ev-app[OF wff-App[OF wff-Leib Desc.hyps]
          wff-Fre M.asg-upd[OF Desc.prems(2) b]]
          M.ev-var[OF M.asg-upd[OF Desc.prems(2) b]] c)
    thus ?thesis
      by (metis Desc.hyps that cA Desc.prems(2) wff-Fre M.sat-Leib M.ev-var
app-struct.asg-upd
              upd-same M.app-struct-axioms)
  qed
  have Ee ξ (Iota α · (Leib α · A)) = Ee ξ A
   using M.vl-iota[OF Desc.prems(2) M.ev-type[OF wff-App[OF wff-Leib Desc.hyps]
Desc.prems(2)]
                M.ev-type[OF Desc.hyps Desc.prems(2)]] sing
    by (simp add: M.ev-app[OF wff-Iota wff-App[OF wff-Leib Desc.hyps]
       Desc.prems(2)])
  thus ?case
   by (simp add: M.sat-Leib[OF wff-App[OF wff-Iota wff-App[OF wff-Leib Desc.hyps]]
                Desc.hyps Desc.prems(2)])
next
  case (EqR α A Φ)
  interpret M: bkk-model Dm Ap Ee vl by (rule EqR.prems(1))
  show ?case
   by (simp add: M.ev-app[OF wff-App[OF wff-Eq EqR.hyps] EqR.hyps EqR.prems(2)]
       M.ev-app[OF wff-Eq EqR.hyps EqR.prems(2)]
       M.vl-eq[OF EqR.prems(2) M.ev-type[OF EqR.hyps EqR.prems(2)]
                M.ev-type[OF EqR.hyps EqR.prems(2)]])
next case (EqL Φ C α D)
  interpret M: bkk-model Dm Ap Ee vl by (rule EqL.prems(1))
  have wCD: wff_o(C =α D) using bprov-wff[OF EqL.hyps(1)] EqL.prems(3)
    by blast
  have wC: wff_α(C) and wD: wff_α(D) using wCD
    by (auto dest: wff-unique)
  have vl (Ee ξ (C =α D)) using EqL.IH[OF EqL.prems] .
  hence Ee ξ C = Ee ξ D
    by (simp add: M.ev-app[OF wff-App[OF wff-Eq wC] wD EqL.prems(2)]
       M.ev-app[OF wff-Eq wC EqL.prems(2)]
        M.vl-eq[OF EqL.prems(2) M.ev-type[OF wC EqL.prems(2)] M.ev-type[OF
wD EqL.prems(2)]])
  thus ?case by (simp add: M.sat-Leib[OF wC wD EqL.prems(2)])
qed
```

4.2.5 The canonical construction is a BKK model

Every Σ-Henkin general model in the sense of Section 2 — the frame-based canonical construction — is a BKK model: take $E := den$ and $v := (\lambda a.\ a$

$= Tv$). The four evaluation conditions are the denotation lemmas, and the L-properties are the *gm*-conditions.

sublocale *general-model* $\subseteq$ *bkkA*: *app-struct Dm Ap*
  by *unfold-locales* (*use gm-nonempty gm-appTy* **in** *blast*)+

sublocale *general-model* $\subseteq$ *bkk*: *bkk-model Dm Ap* $\lambda\xi\ A.\ (\!|A|\!)_\xi\ \lambda a.\ a = Tv$
  by (*unfold-locales*;
    auto simp: *bkkA.functional-def general-model-axioms*[*unfolded general-model-def*]
          bkkA.asg-def frame-sig.den.simps(*8*) *resp-def*
      intro: *beq-den den-coincidence den-dom*)

4.2.6 Derived rules for the defined quantifiers

$NK(\Pi E)$ and $NK(\Pi I)$ for the folded quantifier $\mathbf{\Pi}_\sigma$.

lemma *PiE-Forall*: $\Phi \vdash \mathbf{\Pi}_\alpha\ b \Longrightarrow \mathit{wff}_\alpha(A) \Longrightarrow \Phi \vdash (\mathbf{\Lambda}_\alpha\ b) \cdot A$
  unfolding *Forall-def* **by** (*rule bprov.PiE*)
lemma *PiI-Forall*: $\Phi \vdash (\mathbf{\Lambda}_\alpha\ b) \cdot (w^p{}_\alpha) \Longrightarrow \mathit{wff}\alpha \Rightarrow \mathrm{o}(\mathbf{\Lambda}_\alpha\ b) \Longrightarrow w \notin \mathit{pars}\ (\mathbf{\Lambda}_\alpha\ b)$
    $\Longrightarrow (\forall D \in \Phi.\ w \notin \mathit{pars}\ D) \Longrightarrow \Phi \vdash \mathbf{\Pi}_\alpha\ b$
  unfolding *Forall-def* **by** (*rule bprov.PiI*)

Existential elimination at a fresh eigen-parameter, for the defined quantifier $\exists\ = \neg\mathbf{\Pi}\neg$: the derived counterpart of the paper-style step "obtain a witness".

lemma *ExE*: **assumes** *ex*: $\Phi \vdash \exists_\sigma\ b$ **and** *step*: $\Phi \cup \{b\langle w^p{}_\sigma\rangle\} \vdash C$
    and *wb*: $\mathit{wff}_\sigma \Rightarrow \mathrm{o}(\mathbf{\Lambda}_\sigma\ b)$
  and *wC*: $\mathit{wff}_\mathrm{o}(C)$ **and** *wpb*: $w \notin \mathit{pars}\ b$ **and** *wpC*: $w \notin \mathit{pars}\ C$
    and *wpΦ*: $\forall D \in \Phi.\ w \notin \mathit{pars}\ D$ **and** *fp*: *freep* Φ
  shows $\Phi \vdash C$
proof −
  have *wNb*: $\mathit{wff}_\sigma \Rightarrow \mathrm{o}(\mathbf{\Lambda}_\sigma\ (\neg\ b))$
    by (*auto intro*!: *wff-opn*[*OF wb*] *wff-Fre wff-AbsI*)
  have *fp1*: *freep* $(\Phi \cup \{\neg\ C\})$ **using** *freep-add*[*OF fp*] **by** *simp*
  have *fp2*: *freep* $(\Phi \cup \{\neg\ C\} \cup \{b\langle w^p{}_\sigma\rangle\})$
    using *freep-add*[*OF freep-add*[*OF fp*]] **by** *simp*
  have *s1*: $\Phi \cup \{\neg\ C\} \cup \{b\langle w^p{}_\sigma\rangle\} \vdash C$
    by (*rule bprov-weaken*[*OF step - fp2*]) *auto*
  have *s2*: $\Phi \cup \{\neg\ C\} \cup \{b\langle w^p{}_\sigma\rangle\} \vdash \neg\ C$ **by** (*auto intro*: *bprov.Hyp*)
  have *bq*: $(\mathbf{\Lambda}_\sigma\ (\neg\ b)) \cdot (w^p{}_\sigma) \approx_\mathrm{o} \neg\ (b\langle w^p{}_\sigma\rangle)$
    using *beq.beta*[*OF wNb wff-Par*] **by** *simp*
  have *s5*: $\Phi \cup \{\neg\ C\} \vdash \mathbf{\Pi}_\sigma\ (\neg\ b)$
    using *wpb wpC wpΦ*
    by (*safe intro*!:
      PiI-Forall[*OF bprov.Beta*[*OF beq.sym*[*OF bq*] *bprov.NegI*[*OF*
        bprov.NegE[*OF s2 s1 wff-FalseB*] *wff-opn*[*OF wb wff-Par*]]] *wNb*]) *auto*
  have *s6*: $\Phi \cup \{\neg\ C\} \vdash \neg\ (\mathbf{\Pi}_\sigma\ (\neg\ b))$ **by** (*rule bprov-weaken*[*OF ex - fp1*]) *auto*
  show *?thesis* **by** (*rule bprov.Contr*[*OF bprov.NegE*[*OF s6 s5 wff-FalseB*] *wC*])
qed

Universal instantiation directly at the β-reduced instance.

lemma *PiE-open*: $\Phi \vdash \mathbf{\Pi}_\alpha\ b \Longrightarrow \mathit{wff}_\alpha \Rightarrow {}_{\mathrm{o}}(\mathbf{\Lambda}_\alpha\ b) \Longrightarrow \mathit{wff}_\alpha(A) \Longrightarrow \Phi \vdash b\langle A\rangle$
by (*metis PiE-Forall beq.beta bprov.Beta*)

Soundness, repackaged in the validity and satisfaction notation: the two forms used by the closing summary.

theorem *soundness-sat*:
$\Phi \vdash C \Longrightarrow \Phi \models(\prime u)\ C$
by (*simp add*: *bkk-consequence-def rel-truth-def soundness-bkk*)

theorem *soundness-valid*: $\vdash A \Longrightarrow \models(\prime u)\ A$
unfolding *bkk-valid-def rel-truth-def*
by (*auto intro*: *soundness-bkk*[*of* {} *A*])

5 Completeness

Henkin completeness via the model-existence / abstract-consistency method of BKK Section 6 (Corollary 7.7), with the term model realised as a term evaluation (BKK Definition 3.35) — then strengthened to arbitrary infinite value carriers, signatures with infinitely many parameters, and open formulas.

5.1 Model existence and completeness (BKK Section 6, Corollary 7.7)

We build a Henkin term model from a maximal consistent, saturated set of sentences, following BKK's model-existence route (BKK Section 6). This file develops *NK*-consistency and its closure properties (BKK Lemma 7.5), the maximal saturated extension (BKK Lemma 6.32), the term model with its truth lemma (BKK Theorem 6.33), and the completeness theorem itself (BKK Corollary 7.7).

5.1.1 Consistency (BKK Definition 7.4)

A set of sentences is *NK-consistent* (BKK Definition 7.4) if falsity is not derivable from it.

definition *con* :: $'p\ tm\ set \Rightarrow bool$ **where** $con\ \Phi \longleftrightarrow \neg\ (\Phi \vdash \bot)$
lemma *con-I*: $(\Phi \vdash \bot \Longrightarrow \mathit{False}) \Longrightarrow con\ \Phi$ **by** (*auto simp*: *con-def*)

Subsets of a consistent set are consistent (using weakening).

lemma *con-mono*: $con\ \Psi \Longrightarrow \Phi \subseteq \Psi \Longrightarrow \mathit{freep}\ \Psi \Longrightarrow con\ \Phi$ **unfolding** *con-def*
using *bprov-weaken* **by** *blast*

Consistency is of finite character (compactness, BKK Definition 6.1): a set is consistent as soon as all its finite subsets are.

lemma *con-compact*:
(⋀Φ0::'p::*infinite tm set*. *finite* Φ0 ⟹ Φ0 ⊆ Φ ⟹ *con* Φ0) ⟹ *con* Φ
using *bprov-finite* **unfolding** *con-def* **by** *blast*

The central step of a maximal-consistent extension (the ∇_{sat} case of BKK Lemma 7.5; property ∇_{sat} is BKK Definition 6.5): from a consistent set, adding a proposition or its negation keeps it consistent.

lemma *con-split*: **assumes** *con* Φ **and** *wff*$_{\mathrm{O}}$(*A*)
shows *con* (*insert A* Φ) ∨ *con* (*insert* (¬ *A*) Φ)
proof (*rule ccontr*)
assume ¬*?thesis*
hence Φ ∪ {*A*} ⊢ ⊥ **and** Φ ∪ {¬ *A*} ⊢ ⊥ **by** (*auto simp*: *con-def*)
from ‹Φ ∪ {*A*} ⊢ ⊥› **have** Φ ⊢ ¬ *A* **using** *assms*(*2*)
by (*rule bprov.NegI*)
moreover from ‹Φ ∪ {¬ *A*} ⊢ ⊥› **have** Φ ⊢ *A* **using** *assms*(*2*)
by (*rule bprov.Contr*)
ultimately have Φ ⊢ ⊥ **using** *wff-FalseB* **by** (*rule bprov.NegE*)
thus *False* **using** *assms*(*1*) **by** (*simp add*: *con-def*)
qed

A consistent set does not contain both a proposition and its negation.

lemma *con-not-both*: *con* Φ ⟹ *A* ∈ Φ ⟹ ¬ *A* ∈ Φ ⟹ *wff*$_{\mathrm{O}}$(*A*) ⟹ *False*
by (*metis con-def wff-FalseB NegE Hyp*)

5.1.2 Some admissible rules

lemma *freep-un*: *freep* Φ ⟹ *freep* (Φ ∪ {*A*})
by (*metis freep-add Un-insert-right sup-bot.right-neutral*)

Double-negation elimination (derivable from the classical rule $NK(Contr)$).

lemma *dneg*: Φ ⊢ ¬ (*Neg* · *X*) ⟹ *wff*$_{\mathrm{O}}$(*X*) ⟹ *freep* Φ ⟹ Φ ⊢ *X*
by (*metis* (*no-types*, *lifting*) *Contr Hyp NegE Un-insert-right bprov-weaken*
freep-add insertI1 subset-insertI sup-bot.right-neutral wff-FalseB)

Excluded middle and implication introduction (derivable in the classical calculus).

lemma *bprov-em*: **assumes** *wA*: *wff*$_{\mathrm{O}}$(*A*) **shows** Φ ⊢ (¬ *A*) ∨ *A*
proof −
let *?D* = (¬ *A*) ∨ *A*
have *wnA*: *wff*$_{\mathrm{O}}$(¬ *A*) **using** *wA* **by** (*rule wff-Not*)
have *wD*: *wff*$_{\mathrm{O}}$(*?D*) **using** *wnA wA* **by** (*rule wff-Or*)
have Φ ∪ {¬ *?D*} ⊢ ¬ *A*
proof (*rule bprov.NegI*[*OF - wA*])
have (Φ ∪ {¬ *?D*}) ∪ {*A*} ⊢ *A* **by** (*auto intro*: *bprov.Hyp*)
hence (Φ ∪ {¬ *?D*}) ∪ {*A*} ⊢ *?D* **using** *wnA* **by** (*rule bprov.DisIR*)
moreover have (Φ ∪ {¬ *?D*}) ∪ {*A*} ⊢ ¬ *?D*
by (*auto intro*: *bprov.Hyp*)
ultimately show (Φ ∪ {¬ *?D*}) ∪ {*A*} ⊢ ⊥ **using** *wff-FalseB*
by (*metis bprov.NegE*)

qed
hence $\Phi \cup \{\neg\ ?D\} \vdash ?D$ **using** *wA* **by** (*rule bprov.DisIL*)
moreover have $\Phi \cup \{\neg\ ?D\} \vdash \neg\ ?D$ **by** (*auto intro*: *bprov.Hyp*)
ultimately have $\Phi \cup \{\neg\ ?D\} \vdash \bot$ **using** *wff-FalseB* **by** (*metis bprov.NegE*)
thus *?thesis* **using** *wD* **by** (*rule bprov.Contr*)
qed
lemma *bprov-ImpE*: **assumes** *AB*: $\Phi \vdash A \supset B$ **and** *A*: $\Phi \vdash A$
and *fp*: *freep* Φ
and *wA*: $wff_{\mathrm{o}}(A)$ **and** *wB*: $wff_{\mathrm{o}}(B)$ **shows** $\Phi \vdash B$
proof −
have *wnA*: $wff_{\mathrm{o}}(\neg\ A)$ **using** *wA* **by** (*rule wff-Not*)
have *disj*: $\Phi \vdash (\neg\ A) \vee B$ **using** *AB* **by** (*simp add*: *ImpB-def*)
have *b1*: $\Phi \cup \{\neg\ A\} \vdash B$
by (*meson A Hyp NegE bprov-weaken fp freep-un inf-sup-ord*(*4*) *insertCI sup-ge1 wB*)
have *b2*: $\Phi \cup \{B\} \vdash B$ **by** (*auto intro*: *bprov.Hyp*)
show *?thesis* **by** (*rule bprov.DisE*[*OF disj b1 b2 wnA wB*])
qed
lemma *bprov-TrueB*: $\Phi \vdash \top$ **by** (*simp add*: *Hyp TrueB-def bprov.intros*(*3*))

5.1.3 Witnessing a false universal (BKK property $\nabla_\exists$, Definition 6.5; the $\nabla_\exists$ case of Lemma 7.5)

If $\neg\Pi_\alpha\ G$ is consistent with Φ, then it stays consistent when we add a witness $\neg\ (G\ c)$ for a fresh parameter c. This is the key step for making the extension saturated.

lemma *con-witness*:
assumes *con*: *con* (*insert* $(\neg\ ((Pi\ \alpha) \cdot G))\ \Phi$)
and *wG*: $wff_{\alpha \Rightarrow \mathrm{o}}(G)$ **and** *fp*: *freep* Φ
and *c*: $c \notin usedp\ \Phi\ c \notin pars\ G$
shows *con* (*insert* $(\neg\ (G \cdot (c^p\ \alpha)))$ (*insert* $(\neg\ ((Pi\ \alpha) \cdot G))\ \Phi$))
proof (*rule con-I*)
define Γ **where** $\Gamma = \Phi \cup \{\neg\ ((Pi\ \alpha) \cdot G)\}$
have *fpG*: *freep* Γ **unfolding** Γ-*def* **by** (*rule freep-un*[*OF fp*])
assume *insert* $(\neg\ (G \cdot (c^p\ \alpha)))$ (*insert* $(\neg\ ((Pi\ \alpha) \cdot G))\ \Phi \vdash \bot$
hence $\Gamma \cup \{\neg\ (G \cdot (c^p\ \alpha))\} \vdash \bot$ **by** (*simp add*: Γ-*def insert-commute*)
hence $\Gamma \vdash \neg\ (Neg \cdot (G \cdot (c^p\ \alpha)))$
using *wff-Not*[*OF wff-App*[*OF wG wff-Par*]] **by** (*rule bprov.NegI*)
hence $\Gamma \vdash G \cdot (c^p\ \alpha)$
using *wff-App*[*OF wG wff-Par*] *fpG* **by** (*rule dneg*)
moreover have $c \notin pars\ G$ **and** $\forall\ D \in \Gamma.\ c \notin pars\ D$ **using** *c*
by (*auto simp*: Γ-*def usedp-def*)
ultimately have *piG*: $\Gamma \vdash (Pi\ \alpha) \cdot G$ **using** *wG*
by (*blast intro*: *bprov.PiI*)
have $\Gamma \vdash \neg\ ((Pi\ \alpha) \cdot G)$ **by** (*auto simp*: Γ-*def intro*: *bprov.Hyp*)
from *this piG* **have** $\Gamma \vdash \bot$ **using** *wff-FalseB* **by** (*rule bprov.NegE*)
thus *False* **using** *con* **by** (*simp add*: *con-def* Γ-*def*)
qed

5.1.4 Maximal consistent, saturated extension (BKK's abstract extension lemma 6.32)

We enumerate the (countably many) sentences and, step by step, decide each one or its negation (*con-split*), immediately adding a Henkin witness for a decided negated universal (*con-witness*). The union of the chain is a maximal consistent, saturated set.

definition *freshc* :: *'p tm set* ⇒ *'p tm* ⇒ *'p* **where**
freshc S G ≡ *SOME c. c* ∉ *usedp S* ∧ *c* ∉ *pars G*

lemma *freshc-fresh*: *freep S* ⟹ *freshc S G* ∉ *usedp S* ∧ *freshc S G* ∉ *pars G*
by (*metis* (*lifting*) *finite-pars freep-fresh freshc-def someI-ex*)

fun *wit* :: *'p tm set* ⇒ *'p tm* ⇒ *'p tm set* **where**
‹*wit S* (¬*App* (*Pi* α) *G*) = {¬ (*G* · ((*freshc S G*)^{p}α))}›
| ‹*wit* - - = {}›

lemma *wit-cases*: *wit S A* = {} ∨ (∃α *G*. *A* = ¬ ((*Pi* α) · *G*) ∧
wit S A = {¬ (*G* · ((*freshc S G*)^{p}α))})
by (*induct S A rule*: *wit.induct*) *auto*
lemma *finite-wit*: *finite* (*wit S A*)
by (*induct S A rule*: *wit.induct*) *auto*
definition *step* :: *'p tm set* ⇒ *'p tm* ⇒ *'p tm set* **where**
step S A ≡ *if cwff* ∘ *A*
then (*if con* (*insert A S*) *then insert A S* ∪ *wit S A else insert* (¬ *A*) *S*)
else S

primrec *ext* :: *'p*::{*countable*,*infinite*} *tm set* ⇒ *nat* ⇒ *'p tm set* **where**
ext Φ *0* = Φ
| *ext* Φ (*Suc n*) = *step* (*ext* Φ *n*) (*from-nat n*)

definition *Hset* :: *'p*::{*countable*,*infinite*} *tm set* ⇒ *'p tm set* **where**
Hset Φ = (⋃*n*. *ext* Φ *n*)

The chain is increasing and each stage stays finite-in-parameters and consistent.

lemma *ext-mono*: *ext* Φ *n* ⊆ *ext* Φ (*Suc n*)
by (*induct n*) (*auto simp*: *step-def*)
lemma *ext-mono'*: *m* ≤ *n* ⟹ *ext* Φ *m* ⊆ *ext* Φ *n*
using *ext-mono lift-Suc-mono-le* **by** *blast*
lemma *freep-un-finite*: *freep S* ⟹ *finite T* ⟹ *freep* (*S* ∪ *T*)
proof −
assume *freep S finite T*
hence *finite* (*usedp T*) **by** (*auto simp*: *usedp-def*)
moreover have − *usedp* (*S* ∪ *T*) = − *usedp S* − *usedp T*
by (*auto simp*: *usedp-def*)
ultimately show *?thesis* **using** ‹*freep S*›
by (*metis freep-def Diff-infinite-finite*)
qed

lemma *freep-step*: *freep S* $\Longrightarrow$ *freep* (*step S A*)
by (*simp add*: *finite-wit freep-add freep-un-finite step-def*)
lemma *freep-ext*: *freep* Φ $\Longrightarrow$ *freep* (*ext* Φ *n*)
by (*induction n*) (*auto simp*: *freep-step*)
lemma *con-step*: **assumes** *con S* **and** *freep S* **shows** *con* (*step S A*)
proof (*cases cwff* o *A*)
case *False*
with *assms*(*1*) **show** *?thesis* **by** (*auto simp*: *step-def*)
next case *True*
show *?thesis*
proof (*cases con* (*insert A S*))
case *False*
hence *con* (*insert* ($\neg$ *A*) *S*)
using *con-split*[*OF assms*(*1*)] *cwff-wff*[*OF True*] **by** *blast*
thus *?thesis* **using** *False True* **by** (*simp add*: *step-def*)
next
case *True*
have *con* (*insert A S* $\cup$ *wit S A*) **using** *wit-cases*[*of S A*]
proof
assume *wit S A* = {} **thus** *?thesis* **using** *True* **by** *simp*
next
assume $\exists\,\alpha\ G.\ A = \neg\ ((Pi\ \alpha) \cdot G) \wedge wit\ S\ A = \{\neg\ (G \cdot ((freshc\ S\ G)^p{}_\alpha))\}$
then obtain α *G* **where** *AG*: *A* = $\neg$ ((*Pi* α) $\cdot$ *G*)
and *w*: *wit S A* = {$\neg$ (*G* $\cdot$ ((*freshc S G*)$^p{}_\alpha$))} **by** *blast*
have *wG*: *wff*$_\alpha$$\Rightarrow$$_{\mathrm{o}}$(*G*) **using** *cwff-wff*[*OF* ‹*cwff* o *A*›] *AG*
by (*auto dest*: *wff-unique*)
have *fr*: *freshc S G* $\notin$ *usedp S freshc S G* $\notin$ *pars G*
using *freshc-fresh*[*OF assms*(*2*)] **by** *auto*
have *con* (*insert* ($\neg$ (*G* $\cdot$ ((*freshc S G*)$^p{}_\alpha$))) (*insert A S*))
by (*metis AG True assms*(*2*) *con-witness fr*(*1*,*2*) *wG*)
thus *?thesis* **using** *w* **by** *simp*
qed
thus *?thesis* **using** *True* ‹*cwff* o *A*› **by** (*simp add*: *step-def*)
qed
qed
lemma *con-ext*: *con* Φ $\Longrightarrow$ *freep* Φ $\Longrightarrow$ *con* (*ext* Φ *n*)
by (*induction n*) (*auto simp*: *con-step freep-ext*)

Hset Φ extends Φ and, by compactness, is consistent.

lemma *Phi-sub-Hset*: $\Phi \subseteq$ *Hset* Φ **unfolding** *Hset-def* **using** *ext.simps*(*1*)
by *blast*
lemma *finite-sub-ext*: *finite F* $\Longrightarrow$ *F* $\subseteq$ *Hset* Φ $\Longrightarrow$ $\exists\,N.\ F \subseteq ext\ \Phi\ N$
proof (*induction F rule*: *finite-induct*)
case *empty* **thus** *?case* **by** *blast*
next
case (*insert x F*)
then obtain *N* **where** *N*: *F* $\subseteq$ *ext* Φ *N* **by** *auto*
from *insert.prems* **obtain** *M* **where** *M*: *x* $\in$ *ext* Φ *M*
by (*auto simp*: *Hset-def*)

```
  have insert x F ⊆ ext Φ (max N M)
    using N M ext-mono'[of N max N M Φ] ext-mono'[of M max N M Φ] by auto
  thus ?case by blast
qed
lemma con-Hset: fixes Φ :: 'p::{countable,infinite} tm set
  assumes con Φ and freep Φ
  shows con (Hset Φ)
  by (metis assms(1,2) con-compact con-ext con-mono finite-sub-ext freep-ext)
```

Crucially, *freep* (*Hset* Φ) fails (a maximal set uses every parameter), so we never weaken *to Hset*. Instead we use that every *finite* subset of *Hset* is consistent — finite contexts are always *freep*, which is all the proof rules need.

```
lemma Hset-finite-con: fixes Φ :: 'p::{countable,infinite} tm set
  assumes con Φ and freep Φ and finite F and F ⊆ Hset Φ
  shows con F
  by (meson assms(1,2,3,4) con-ext con-mono finite-sub-ext freep-ext)
```

Hset decides every sentence — BKK call this *saturated*, property $\sim\nabla_{sat}$ (BKK Definition 6.24) — and has the Henkin witness property $\sim\nabla_{\exists}$ (BKK Definition 6.19). We call the former *maximality* and reserve *saturation* for the witness property; the lemma names below follow this convention.

```
lemma Hset-maximal: fixes Φ :: 'p::{countable,infinite} tm set
  assumes c: cwff o A shows A ∈ Hset Φ ∨ ¬ A ∈ Hset Φ
proof −
  have A ∈ ext Φ (Suc (to-nat A)) ∨ ¬ A ∈ ext Φ (Suc (to-nat A))
    using c by (auto simp: step-def)
  thus ?thesis unfolding Hset-def by blast
qed
lemma Hset-saturated: fixes Φ :: 'p::{countable,infinite} tm set
  assumes conΦ: con Φ and fp: freep Φ
  and cA: cwff o (¬ ((Pi α) · G)) and inH: ¬ ((Pi α) · G) ∈ Hset Φ
shows ∃ c. ¬ (G · (c^p α)) ∈ Hset Φ
proof −
  note wA = cwff-wff[OF cA] and fvsA = cwff-closed[OF cA]
  let ?A = ¬ ((Pi α) · G)
  let ?S = ext Φ (to-nat ?A)
  have step: ext Φ (Suc (to-nat ?A)) = step ?S ?A
    by simp
  have con (insert ?A ?S)
  proof (rule ccontr)
    assume ¬ con (insert ?A ?S)
    hence ¬ ?A ∈ ext Φ (Suc (to-nat ?A)) using cA step
      by (auto simp: step-def)
    hence ¬ ?A ∈ Hset Φ unfolding Hset-def by blast
    thus False using con-not-both[OF con-Hset[OF conΦ fp] inH - wA]
      by blast
  qed
  hence wit ?S ?A ⊆ ext Φ (Suc (to-nat ?A)) using cA step
```

by (*auto simp*: *step-def*)
moreover have *wit ?S ?A* = {¬ (*G* · ((*freshc ?S G*)^p α))}
by *simp*
ultimately have ¬ (*G* · ((*freshc ?S G*)^p α)) ∈ *Hset* Φ
unfolding *Hset-def* **by** *blast*
thus *?thesis* **by** *blast*
qed

5.1.5 Leibniz equality is reflexive (BKK property ∇_r, Lemma 6.25)

lemma *leib-refl*: **assumes** *fp*: *freep* Φ **and** *wa*: *wff*$_\alpha$(*A*)
shows Φ ⊢ *A* $\doteq_\alpha$ *A* **by** (*simp add*: *EqL EqR wa*)

Leibniz substitution (the substitutivity built into BKK's Leibniz equality, Section 2.2; cf. the ∇-properties of BKK Lemma 6.12): equals may replace equals in any predicate. Instantiating *P* with suitable predicates yields symmetry, transitivity and congruence.

lemma *leib-subst*:
assumes *AB*: Φ ⊢ *A* $\doteq_\alpha$ *B* **and** *fp*: *freep* Φ **and** *wa*: *wff*$_\alpha$(*A*) **and** *wb*: *wff*$_\alpha$(*B*)
and *wP*: *wff*$_{\alpha\Rightarrow o}$(*P*) **and** *PA*: Φ ⊢ *P* · *A*
shows Φ ⊢ *P* · *B*
proof −
let *?body* = (*Bnd 0*) · *A* ⊃ (*Bnd 0*) · *B*
have Φ ⊢ **Π**$_{\alpha\Rightarrow o}$ *?body*
using *Leib-beq*[*OF wa wb*] *AB* **by** (*rule bprov.Beta*)
hence Φ ⊢ (*Pi* (α ⇒ o)) · (**Λ**α ⇒ o *?body*) **by** (*simp add*: *Forall-def*)
hence *P*: Φ ⊢ (**Λ**α ⇒ o *?body*) · *P* **using** *wP* **by** (*rule bprov.PiE*)
have *wAbs*: *wff*$_{(\alpha\Rightarrow o)\Rightarrow o}$(**Λ**α ⇒ o *?body*)
by (*auto simp*: *opn-lc*[*OF wff-lc*[*OF wa*]] *opn-lc*[*OF wff-lc*[*OF wb*]]
intro!: *wff-App wff-Fre wa wb wff-AbsI*)
have (**Λ**α ⇒ o *?body*) · *P* ≈$_o$ *P* · *A* ⊃ *P* · *B*
proof −
have (**Λ**α ⇒ o *?body*) · *P* ≈$_o$ *?body*⟨*P*⟩
by (*rule beq.beta*[*OF wAbs wP*])
thus *?thesis*
by (*simp add*: *opn-lc*[*OF wff-lc*[*OF wa*]] *opn-lc*[*OF wff-lc*[*OF wb*]])
qed
from *bprov.Beta*[*OF this P*] **have** Φ ⊢ *P* · *A* ⊃ *P* · *B*.
moreover have *wff*$_o$(*P* · *A*) **by** (*rule wff-App*[*OF wP wa*])
moreover have *wff*$_o$(*P* · *B*) **by** (*rule wff-App*[*OF wP wb*])
ultimately show *?thesis* **using** *PA fp* **by** (*metis bprov-ImpE*)
qed
lemma *leib-sym*:
assumes *AB*: Φ ⊢ *A* $\doteq_\alpha$ *B* **and** *fp*: *freep* Φ **and** *wa*: *wff*$_\alpha$(*A*) **and** *wb*: *wff*$_\alpha$(*B*)
shows Φ ⊢ *B* $\doteq_\alpha$ *A*
proof −
have *lcA*: *lc A* **using** *wa* **by** (*rule wff-lc*)
let *?P* = **Λ**$_\alpha$ (((*Leib* α) · (*Bnd 0*)) · *A*) — the predicate λ*x*. *x* $\doteq$ *A*

have *wP*: $wff_{\alpha \Rightarrow o}(?P)$ **by** (*auto simp*: *opn-lc*[*OF lcA*] *intro*!: *wff-Fre wa wff-AbsI*)
have *PbA*: $?P \cdot A \approx_o (A \doteq_\alpha A)$
using *beq.beta*[*OF wP wa*] **by** (*simp add*: *opn-lc*[*OF lcA*])
have *PbB*: $?P \cdot B \approx_o (B \doteq_\alpha A)$
using *beq.beta*[*OF wP wb*] **by** (*simp add*: *opn-lc*[*OF lcA*])
have $\Phi \vdash A \doteq_\alpha A$ **using** *fp wa* **by** (*rule leib-refl*)
hence $\Phi \vdash ?P \cdot A$
using *beq.sym*[*OF PbA*] **by** (*rule bprov.Beta*[*rotated*])
hence $\Phi \vdash ?P \cdot B$ **using** *leib-subst*[*OF AB fp wa wb wP*] **by** *auto*
thus *?thesis* **by** (*rule bprov.Beta*[*OF PbB*])
qed
lemma *leib-trans*:
assumes *AB*: $\Phi \vdash A \doteq_\alpha B$ **and** *BC*: $\Phi \vdash B \doteq_\alpha C$ **and** *fp*: *freep* Φ
and *wa*: $wff_\alpha(A)$ **and** *wb*: $wff_\alpha(B)$ **and** *wc*: $wff_\alpha(C)$
shows $\Phi \vdash A \doteq_\alpha C$
proof −
have *lcA*: *lc A* **using** *wa* **by** (*rule wff-lc*)
let $?P = \mathbf{\Lambda}_\alpha\ (((Leib\ \alpha) \cdot A) \cdot (Bnd\ 0))$ — the predicate $\lambda x.\ A \doteq x$
have *wP*: $wff_{\alpha \Rightarrow o}(?P)$
by (*auto simp*: *opn-lc*[*OF lcA*] *intro*!: *wff-Fre wa wff-AbsI*)
have *PbB*: $?P \cdot B \approx_o (A \doteq_\alpha B)$
using *beq.beta*[*OF wP wb*] **by** (*simp add*: *opn-lc*[*OF lcA*])
have *PbC*: $?P \cdot C \approx_o (A \doteq_\alpha C)$
using *beq.beta*[*OF wP wc*] **by** (*simp add*: *opn-lc*[*OF lcA*])
from *AB* **have** $\Phi \vdash ?P \cdot B$ **by** (*rule bprov.Beta*[*OF beq.sym*[*OF PbB*]])
hence $\Phi \vdash ?P \cdot C$ **using** *leib-subst*[*OF BC fp wb wc wP*] **by** *auto*
thus *?thesis* **by** (*rule bprov.Beta*[*OF PbC*])
qed
lemma *leib-cong2*:
assumes *AA*: $\Phi \vdash A \doteq_\alpha A'$ **and** *fp*: *freep* Φ
and *wa*: $wff_\alpha(A)$ **and** *wa'*: $wff_\alpha(A')$ **and** *wC*: $wff_{\alpha \Rightarrow \beta}(C)$
shows $\Phi \vdash (C \cdot A) \doteq_\beta (C \cdot A')$
proof −
have *lcC*: *lc C* **using** *wC* **by** (*rule wff-lc*)
have *lcA*: *lc A* **using** *wa* **by** (*rule wff-lc*)
let $?P = \mathbf{\Lambda}_\alpha\ (C \cdot A \doteq_\beta C \cdot (Bnd\ 0))$ — $\lambda x.\ (C\ A) \doteq (C\ x)$
have *wP*: $wff_{\alpha \Rightarrow o}(?P)$
by (*auto simp*: *opn-lc*[*OF lcC*] *opn-lc*[*OF lcA*]
intro!: *wff-AbsI wff-App wff-Fre wC wa wff-App*[*OF wC wa*])
have *PbA*: $?P \cdot A \approx_o (C \cdot A \doteq_\beta C \cdot A)$
using *beq.beta*[*OF wP wa*] **by** (*simp add*: *opn-lc*[*OF lcC*] *opn-lc*[*OF lcA*])
have *PbA'*: $?P \cdot A' \approx_o (C \cdot A \doteq_\beta C \cdot A')$
using *beq.beta*[*OF wP wa'*] **by** (*simp add*: *opn-lc*[*OF lcC*] *opn-lc*[*OF lcA*])
have $\Phi \vdash C \cdot A \doteq_\beta C \cdot A$ **using** *fp wff-App*[*OF wC wa*]
by (*rule leib-refl*)
hence $\Phi \vdash ?P \cdot A$ **by** (*rule bprov.Beta*[*OF beq.sym*[*OF PbA*]])
hence $\Phi \vdash ?P \cdot A'$ **using** *leib-subst*[*OF AA fp wa wa' wP*] **by** *auto*
thus *?thesis* **by** (*rule bprov.Beta*[*OF PbA'*])
qed

lemma *leib-cong1*:
  assumes *CC*: $\Phi \vdash C \doteq_{\alpha\Rightarrow\beta} C'$ **and** *fp*: *freep* Φ
    and *wC*: $wff_{\alpha\Rightarrow\beta}(C)$ **and** *wC'*: $wff_{\alpha\Rightarrow\beta}(C')$ **and** *wa*: $wff_\alpha(A)$
   shows $\Phi \vdash (C \cdot A) \doteq_\beta (C' \cdot A)$
proof −
  have *lcC*: *lc C* **using** *wC* **by** (*rule wff-lc*)
  have *lcA*: *lc A* **using** *wa* **by** (*rule wff-lc*)
  let *?P* = $\mathbf{\Lambda}_\alpha \Rightarrow_\beta (C \cdot A \doteq_\beta (Bnd\ 0) \cdot A)$ — $\lambda f.\ (C\ A) \doteq (f\ A)$
  have *wP*: $wff_{(\alpha\Rightarrow\beta)\Rightarrow o}(?P)$
    by (*auto simp*: *opn-lc*[*OF lcC*] *opn-lc*[*OF lcA*]
          intro!: *wff-AbsI wff-App wff-Fre wC wa wff-App*[*OF wC wa*])
  have *PbC*: $?P \cdot C \approx_o (C \cdot A \doteq_\beta C \cdot A)$
    using *beq.beta*[*OF wP wC*] **by** (*simp add*: *opn-lc*[*OF lcC*] *opn-lc*[*OF lcA*])
  have *PbC'*: $?P \cdot C' \approx_o (C \cdot A \doteq_\beta C' \cdot A)$
    using *beq.beta*[*OF wP wC'*] **by** (*simp add*: *opn-lc*[*OF lcC*] *opn-lc*[*OF lcA*])
  have $\Phi \vdash C \cdot A \doteq_\beta C \cdot A$
    using *fp wff-App*[*OF wC wa*] **by** (*rule leib-refl*)
  hence $\Phi \vdash ?P \cdot C$ **by** (*rule bprov.Beta*[*OF beq.sym*[*OF PbC*]])
  hence $\Phi \vdash ?P \cdot C'$ **using** *leib-subst*[*OF CC fp wC wC' wP*] **by** *auto*
  thus *?thesis* **by** (*rule bprov.Beta*[*OF PbC'*])
qed

5.1.6 Deductive closure of the Hintikka set

Since *Hset* is maximal and each of its finite subsets is consistent, every sentence provable from a finite subset already belongs to *Hset* (deductive closure — a consequence of the maximality of the extension, BKK Lemma 6.32).

lemma *Hset-deduct*: **fixes** Φ :: *'p*::{*countable*,*infinite*} *tm set*
  assumes *con*Φ: *con* Φ **and** *fp*Φ: *freep* Φ
    and *finF*: *finite F* **and** *subF*: $F \subseteq Hset\ \Phi$ **and** *FS*: $F \vdash S$
    and *cS*: *cwff* o *S* **shows** $S \in Hset\ \Phi$
proof (*rule ccontr*)
  assume $S \notin Hset\ \Phi$
  hence *negS*: $\neg\ S \in Hset\ \Phi$ **using** *Hset-maximal*[*OF cS*] **by** *blast*
  have *fin'*: *finite* (*insert* ($\neg$ *S*) *F*) **using** *finF* **by** *simp*
  have *sub'*: *insert* ($\neg$ *S*) *F* $\subseteq$ *Hset* Φ **using** *negS subF* **by** *auto*
  have *insert* ($\neg$ *S*) *F* $\vdash \bot$
    by (*meson FS Hyp NegE bprov-weaken fin' freep-finite insertCI subset-insertI wff-FalseB*)
  moreover have *con* (*insert* ($\neg$ *S*) *F*)
    by (*rule Hset-finite-con*[*OF con*Φ *fp*Φ *fin' sub'*])
  ultimately show *False* **by** (*simp add*: *con-def*)
qed

In particular, Leibniz equality is reflexive on *Hset* — BKK's property ∇_r for saturated Hintikka sets (Lemma 6.25; Lemma 6.23 gives the negative form $\sim\nabla_{=r}$); symmetry, transitivity and congruence follow below.

lemma *Hset-leib-refl*: **fixes** Φ :: *'p*::{*countable*,*infinite*} *tm set*
assumes *con*Φ: *con* Φ **and** *fp*Φ: *freep* Φ **and** *ca*: *cwff* α *A*
shows $(A \doteq_\alpha A) \in$ *Hset* Φ
proof (*rule Hset-deduct*[*OF con*Φ *fp*Φ, *of* {}])
note *wa* = *cwff-wff*[*OF ca*] **and** *cA* = *cwff-closed*[*OF ca*]
show *finite* {} **by** *simp*
show {} $\subseteq$ *Hset* Φ **by** *simp*
show {} $\vdash A \doteq_\alpha A$
by (*rule leib-refl*[*OF freep-finite*[*OF finite.emptyI*] *wa*])
show *cwff* o $(A \doteq_\alpha A)$
by (*rule cwffI*[*OF wff-LeibE*[*OF wa wa*]]) (*simp add*: *Leib-def cA*)
qed
lemma *Hset-leib-sym*: **fixes** Φ :: *'p*::{*countable*,*infinite*} *tm set*
assumes *con*Φ: *con* Φ **and** *fp*Φ: *freep* Φ
and *ca*: *cwff* α *A* **and** *cb*: *cwff* α *B* **and** *AB*: $(A \doteq_\alpha B) \in$ *Hset* Φ
shows $(B \doteq_\alpha A) \in$ *Hset* Φ
proof (*rule Hset-deduct*[*OF con*Φ *fp*Φ])
note *wa* = *cwff-wff*[*OF ca*] **and** *cA* = *cwff-closed*[*OF ca*] **and**
wb = *cwff-wff*[*OF cb*] **and** *cB* = *cwff-closed*[*OF cb*]
show *finite* $\{A \doteq_\alpha B\}$ **by** *simp*
show $\{A \doteq_\alpha B\} \subseteq$ *Hset* Φ **using** *AB* **by** *simp*
show $\{A \doteq_\alpha B\} \vdash B \doteq_\alpha A$
by (*simp add*: *Hyp freep-finite leib-sym wa wb*)
show *cwff* o $(B \doteq_\alpha A)$
by (*rule cwffI*[*OF wff-LeibE*[*OF wb wa*]]) (*simp add*: *Leib-def cA cB*)
qed
lemma *Hset-leib-trans*: **fixes** Φ :: *'p*::{*countable*,*infinite*} *tm set*
assumes *con*Φ: *con* Φ **and** *fp*Φ: *freep* Φ
and *ca*: *cwff* α *A* **and** *wb*: $wff_\alpha(B)$ **and** *cc*: *cwff* α *C*
and *AB*: $(A \doteq_\alpha B) \in$ *Hset* Φ **and** *BC*: $(B \doteq_\alpha C) \in$ *Hset* Φ
shows $(A \doteq_\alpha C) \in$ *Hset* Φ
proof (*rule Hset-deduct*[*OF con*Φ *fp*Φ])
note *wa* = *cwff-wff*[*OF ca*] **and** *cA* = *cwff-closed*[*OF ca*] **and**
wc = *cwff-wff*[*OF cc*] **and** *cC* = *cwff-closed*[*OF cc*]
let *?F* = $\{A \doteq_\alpha B, B \doteq_\alpha C\}$
show *finite ?F* **by** *simp*
show *?F* $\subseteq$ *Hset* Φ **using** *AB BC* **by** *simp*
show *?F* $\vdash A \doteq_\alpha C$
by (*meson Hyp* ‹*finite* $\{A \doteq_\alpha B, B \doteq_\alpha C\}$› *freep-finite insertCI*
leib-trans wa wb wc)
show *cwff* o $(A \doteq_\alpha C)$
by (*rule cwffI*[*OF wff-LeibE*[*OF wa wc*]]) (*simp add*: *Leib-def cA cC*)
qed
lemma *Hset-leib-cong*: **fixes** Φ :: *'p*::{*countable*,*infinite*} *tm set*
assumes *con*Φ: *con* Φ **and** *fp*Φ: *freep* Φ
and *cc*: *cwff* $(\alpha \Rightarrow \beta)$ *C* **and** *cc'*: *cwff* $(\alpha \Rightarrow \beta)$ *C'* **and** *ca*: *cwff* α *A*
and *ca'*: *cwff* α *A'*
and *CC*: $(C \doteq_{\alpha \Rightarrow \beta} C') \in$ *Hset* Φ **and** *AA*: $(A \doteq_\alpha A') \in$ *Hset* Φ
shows $((C \cdot A) \doteq_\beta (C' \cdot A')) \in$ *Hset* Φ

proof (*rule Hset-deduct*[*OF con*Φ *fp*Φ])
  note wC = *cwff-wff*[*OF cc*] **and** cC = *cwff-closed*[*OF cc*] **and**
    wC' = *cwff-wff*[*OF cc'*] **and** cC' = *cwff-closed*[*OF cc'*] **and**
    wA = *cwff-wff*[*OF ca*] **and** cA = *cwff-closed*[*OF ca*] **and**
    wA' = *cwff-wff*[*OF ca'*] **and** cA' = *cwff-closed*[*OF ca'*]
  let *?F* = $\{C \doteq_{\alpha\Rightarrow\beta} C',\ A \doteq_{\alpha} A'\}$
  have *fp*: *freep ?F* **using** *freep-finite*[*of ?F*] **by** *simp*
  show *finite ?F* **by** *simp*
  show *?F* $\subseteq$ *Hset* Φ **using** *CC AA* **by** *simp*
  show *?F* $\vdash$ $(C \cdot A) \doteq_{\beta} (C' \cdot A')$
    by (*meson Hyp fp insertCI leib-cong1 leib-cong2 leib-trans wA wA' wC wC'*
        wff-App[*OF wC' wA*] *wff-App*[*OF wC' wA'*] *wff-App*[*OF wC wA*])
  show *cwff* o $((C \cdot A) \doteq_{\beta} (C' \cdot A'))$
    by (*rule cwffI*[*OF wff-LeibE*[*OF wff-App*[*OF wC wA*] *wff-App*[*OF wC' wA'*]]])
      (*simp add*: *Leib-def cC cA cC' cA'*)
qed

Leibniz-equal propositions have the same truth (membership transfers along $\sim$ at type o); proved by Leibniz substitution with the identity predicate.

lemma *Hset-leib-mp*: **fixes** Φ :: *'p*::{*countable*,*infinite*} *tm set*
  assumes *con*Φ: *con* Φ **and** *fp*Φ: *freep* Φ
    and *ca*: *cwff* o *A* **and** *cb*: *cwff* o *B* **and** *AB*: $(A \doteq_{\mathrm{o}} B) \in$ *Hset* Φ
    and *A*: $A \in$ *Hset* Φ
   shows $B \in$ *Hset* Φ
proof (*rule Hset-deduct*[*OF con*Φ *fp*Φ - - - *cb*])
  note wA = *cwff-wff*[*OF ca*] **and** cA = *cwff-closed*[*OF ca*] **and**
    wB = *cwff-wff*[*OF cb*] **and** cB = *cwff-closed*[*OF cb*]
  let *?F* = $\{A \doteq_{\mathrm{o}} B,\ A\}$
  let *?P* = $\mathbf{\Lambda}_{\mathrm{o}}$ (*Bnd 0*) — the identity predicate
  have *wP*: *wff*$_{\mathrm{o}\Rightarrow\mathrm{o}}$(*?P*) **by** (*rule wff-AbsI*) (*simp add*: *wff-Fre*)
  have *fp*: *freep ?F* **using** *freep-finite*[*of ?F*] **by** *simp*
  have *PA*: *?P* $\cdot$ *A* $\approx_{\mathrm{o}}$ *A* **using** *beq.beta*[*OF wP wA*] **by** *simp*
  have *PB*: *?P* $\cdot$ *B* $\approx_{\mathrm{o}}$ *B* **using** *beq.beta*[*OF wP wB*] **by** *simp*
  show *finite ?F* **by** *simp*
  show *?F* $\subseteq$ *Hset* Φ **using** *AB A* **by** *simp*
  have *s1*: *?F* $\vdash$ *A* $\doteq_{\mathrm{o}}$ *B* **by** (*auto intro*: *bprov.Hyp*)
  have *s2*: *?F* $\vdash$ *?P* $\cdot$ *A*
    by (*rule bprov.Beta*[*OF beq.sym*[*OF PA*]]) (*auto intro*: *bprov.Hyp*)
  have *?F* $\vdash$ *?P* $\cdot$ *B* **by** (*rule leib-subst*[*OF s1 fp wA wB wP s2*])
  thus *?F* $\vdash$ *B* **by** (*rule bprov.Beta*[*OF PB*])
qed

5.2 The term model (BKK Section 6, the Hintikka lemma)

Fix a consistent, parameter-rich set Φ; its Hintikka extension H is maximal, consistent and saturated. The *term model* has as its domain the closed well-formed terms quotiented by provable Leibniz equality $A \sim B \equiv (A \doteq B) \in H$; this quotient is what forces property q.

locale *hintikka-model* = **fixes** Φ :: *'p*::{*countable*,*infinite*} *tm set*
  assumes *con*Φ: *con* Φ **and** *fp*Φ: *freep* Φ
begin

The $\sim$-equivalence classes of closed well-formed terms (*cwff* from Section 2): $A \sim B \equiv (A \doteq_\sigma B) \in H$ — the Leibniz quotient of BKK's model-existence proof (BKK Theorem 6.33); the $\sim\nabla$-properties of Leibniz equality in H are BKK Lemma 6.23.

definition *cls* :: *'p tm* $\Rightarrow$ *'p tm set* **where**
  cls A $\equiv$ $\{B.\ \exists \sigma.\ cwff\ \sigma\ A \wedge cwff\ \sigma\ B \wedge (A \doteq_\sigma B) \in Hset\ \Phi\}$
lemma *cls-self*: *cwff* σ *A* $\Longrightarrow$ *A* $\in$ *cls A*
  using *Hset-leib-refl*[*OF con*Φ *fp*Φ] **by** (*auto simp*: *cls-def cwff-def*)
lemma *leib-sym'*:
  cwff σ *A* $\Longrightarrow$ *cwff* σ *B* $\Longrightarrow$ $(A \doteq_\sigma B) \in Hset\ \Phi \Longrightarrow (B \doteq_\sigma A) \in Hset\ \Phi$
  using *Hset-leib-sym*[*OF con*Φ *fp*Φ] **by** (*auto simp*: *cwff-def*)
lemma *leib-trans'*:
  cwff σ *A* $\Longrightarrow$ *cwff* σ *B* $\Longrightarrow$ *cwff* σ *C* $\Longrightarrow$ $(A \doteq_\sigma B) \in Hset\ \Phi$
    $\Longrightarrow (B \doteq_\sigma C) \in Hset\ \Phi \Longrightarrow (A \doteq_\sigma C) \in Hset\ \Phi$
  using *Hset-leib-trans*[*OF con*Φ *fp*Φ] **by** (*auto simp*: *cwff-def*)

The quotient is faithful: two classes coincide exactly when the terms are Leibniz-equal in H. This is what will give property q in the model.

lemma *cls-eq-iff*:
  assumes *wA*: *cwff* σ *A* **and** *wB*: *cwff* σ *B*
    shows (*cls A* = *cls B*) $\longleftrightarrow$ $(A \doteq_\sigma B) \in Hset\ \Phi$
  by (*smt* (*verit*, *best*) *Collect-cong cls-def cls-self cwff-unique*
      leib-sym' leib-trans' mem-Collect-eq wA wB)

lemma *cls-rep*:
  assumes *cwff* σ *A*
  shows *cwff* σ (*SOME B*. *B* $\in$ *cls A*) $\wedge$ $(A \doteq_\sigma$ (*SOME B*. *B* $\in$ *cls A*)$) \in Hset\ \Phi$
proof −
  have (*SOME B*. *B* $\in$ *cls A*) $\in$ *cls A* **using** *cls-self*[*OF assms*]
    by (*metis someI*)
  then obtain τ **where** *t*: *cwff* τ *A cwff* τ (*SOME B*. *B* $\in$ *cls A*)
    $(A \doteq_\tau$ (*SOME B*. *B* $\in$ *cls A*)$) \in Hset\ \Phi$ **by** (*auto simp*: *cls-def*)
  have $\tau = \sigma$ **using** *cwff-unique*[*OF t*(*1*) *assms*]**.**
  thus *?thesis* **using** *t*(*2*,*3*) **by** *simp*
qed

The applicative structure of the term model (BKK Definition 3.1): the domain of type σ consists of the classes of closed terms of type σ, and application is term application.

definition *Dm* :: *ty* $\Rightarrow$ *'p tm set* $\Rightarrow$ *bool* **where** *Dm* σ *X* $\equiv \exists$ *A*. *cwff* σ *A* $\wedge$ *X* = *cls A*
definition *Ap* :: *'p tm set* $\Rightarrow$ *'p tm set* $\Rightarrow$ *'p tm set* **where**
  Ap X Y $\equiv$ *cls* ((*SOME A*. *A* $\in$ *X*) $\cdot$ (*SOME B*. *B* $\in$ *Y*))
lemma *Ap-cls*:
  assumes *wA*: *cwff* ($\alpha \Rightarrow \beta$) *A* **and** *wB*: *cwff* α *B*

shows *Ap* (*cls A*) (*cls B*) = *cls* (*A* · *B*)
proof −
have *a*: *cwff* (*α* ⇒ *β*) (*SOME A′*. *A′* ∈ *cls A*) (*A* $\doteq_{\alpha\Rightarrow\beta}$ (*SOME A′*. *A′* ∈ *cls A*)) ∈ *Hset* Φ
using *cls-rep*[*OF wA*] **by** *auto*
have *b*: *cwff α* (*SOME B′*. *B′* ∈ *cls B*) (*B* $\doteq_{\alpha}$ (*SOME B′*. *B′* ∈ *cls B*)) ∈ *Hset* Φ
using *cls-rep*[*OF wB*] **by** *auto*
have *wAB*: *cwff β* (*A* · *B*) **using** *wA wB*
by (*auto simp*: *cwff-def intro*: *wff-App*)
have ((*A* · *B*) $\doteq_{\beta}$ ((*SOME A′*. *A′* ∈ *cls A*) · (*SOME B′*. *B′* ∈ *cls B*))) ∈ *Hset* Φ
using *Hset-leib-cong*[*OF con*Φ *fp*Φ] *wA a*(*1*) *wB b*(*1*) *a*(*2*) *b*(*2*) **by** (*auto simp*: *cwff-def*)
moreover have *cwff β* ((*SOME A′*. *A′* ∈ *cls A*) · (*SOME B′*. *B′* ∈ *cls B*))
using *a*(*1*) *b*(*1*) *cwff-App* **by** *blast*
ultimately have *cls* (*A* · *B*) = *cls* ((*SOME A′*. *A′* ∈ *cls A*) · (*SOME B′*. *B′* ∈ *cls B*))
using *cls-eq-iff wAB* **by** *blast*
thus *?thesis* **by** (*simp add*: *Ap-def*)
qed
lemma *Dm-cls*: *cwff σ A* ⟹ *Dm σ* (*cls A*) **by** (*auto simp*: *Dm-def*)
lemma *Ap-dom*: *Dm* (*α* ⇒ *β*) *X* ⟹ *Dm α Y* ⟹ *Dm β* (*Ap X Y*)
by (*auto simp*: *Dm-def Ap-cls cwff-def intro*: *wff-App*)

Truth and falsity behave (the analogue of BKK Lemma 3.43 / property b): ⊤ ∈ *H*, ⊥ ∉ *H*, so *cls* ⊤ ≠ *cls* ⊥.

lemma *Hset-TrueB*: ⊤ ∈ *Hset* Φ
using *Hset-deduct bprov-TrueB con*Φ *cwff-TrueB fp*Φ **by** *blast*
lemma *Hset-not-FalseB*: ⊥ ∉ *Hset* Φ
by (*metis hintikka-model-axioms Hyp con-def con-Hset hintikka-model-def*)
lemma *TF*: *cls* ⊤ ≠ *cls* ⊥
using *Hset-TrueB Hset-leib-mp Hset-not-FalseB cls-eq-iff con*Φ
*cwff-FalseB cwff-TrueB fp*Φ **by** *blast*

5.2.1 Satisfaction bridges: membership in *H* as truth (BKK Lemma 6.21, Lemmas 6.25–6.26)

H never contains both *φ* and ¬*φ* (BKK's property $\sim\nabla_c$, Definition 6.19, here for arbitrary sentences via Lemma 6.10).

lemma *Hset-notboth*: *wff*$_{\mathrm{o}}$(*φ*) ⟹ *φ* ∈ *Hset* Φ ⟹ ¬ *φ* ∉ *Hset* Φ
using *con*Φ *con-Hset con-not-both fp*Φ **by** *auto*

Boolean extensionality inside *H* (via the rule *NK*(*b*); the saturated-sets lemma for property b, BKK Lemma 6.26): a member of *H* is Leibniz-equal to ⊤, a non-member to ⊥.

lemma *Hset-eq-TrueB*:
assumes *p*: *φ* ∈ *Hset* Φ **and** *w*: *cwff* o *φ*
shows (*φ* $\doteq_{\mathrm{o}}$ ⊤) ∈ *Hset* Φ

proof (*rule Hset-deduct*[*OF con*Φ *fp*Φ, *of* {φ}])
show *finite* {φ} **by** *simp*
show {φ} ⊆ *Hset* Φ **using** *p* **by** *simp*
show {φ} ⊢ $\varphi \doteq_o \top$
by (*metis BoolE Hyp bprov-TrueB cwff-def insertCI insert-is-Un w wff-TrueB*)
show *cwff* o ($\varphi \doteq_o \top$)
by (*metis cwff-App cwff-TrueB cwff-def fvs-defs*(*5*) *w wff-Leib*)
qed

lemma *Hset-eq-FalseB*: **assumes** *np*: ¬ φ ∈ *Hset* Φ **and** *w*: *cwff* o φ
shows ($\varphi \doteq_o \bot$) ∈ *Hset* Φ
proof (*rule Hset-deduct*[*OF con*Φ *fp*Φ, *of* {¬ φ}])
show *finite* {¬ φ} **by** *simp*
show {¬ φ} ⊆ *Hset* Φ **using** *np* **by** *simp*
show {¬ φ} ⊢ $\varphi \doteq_o \bot$
by (*metis BoolE Hyp NegE TrueB-def Un-empty-right*
Un-insert-right bprov-TrueB cwff-wff insertCI w wff-FalseB)
show *cwff* o ($\varphi \doteq_o \bot$)
by (*metis w cwff-def cwff-FalseB cwff-App wff-Leib fvs-defs*(*5*))
qed

Sentences with the same truth value in H are Leibniz-equal in H (the general form of the $NK(b)$ bridge, cf. BKK Lemma 6.26).

lemma *Hset-iff-eq*:
assumes *wp*: *cwff* o φ **and** *wq*: *cwff* o ψ
and *iff*: φ ∈ *Hset* Φ ⟷ ψ ∈ *Hset* Φ
shows ($\varphi \doteq_o \psi$) ∈ *Hset* Φ
by (*metis Hset-eq-FalseB Hset-eq-TrueB Hset-maximal cls-eq-iff*
cwff-FalseB cwff-TrueB iff wp wq)

THE satisfaction bridge: a sentence's class is the class of ⊤ exactly when it is in H (the v-valuation of the term model in the proof of BKK Theorem 6.33).

lemma *cls-TrueB-iff*: *cwff* o φ ⟹ *cls* φ = *cls* ⊤ ⟷ φ ∈ *Hset* Φ
by (*metis Hset-eq-FalseB Hset-eq-TrueB Hset-maximal TF cls-eq-iff cwff-FalseB cwff-TrueB*)

Property b at the class level: the boolean domain has exactly the classes of ⊤ and ⊥ (BKK Definition 3.46).

lemma *cls-bool*: *cwff* o φ ⟹ *cls* φ = *cls* ⊤ ∨ *cls* φ = *cls* ⊥
by (*metis TF cls-eq-iff cwff-FalseB cls-TrueB-iff Hset-iff-eq*)
lemma *cls-FalseB-iff*: *cwff* o φ ⟹ *cls* φ = *cls* ⊥ ⟷ $\varphi \notin$ *Hset* Φ
by (*metis cls-eq-iff cwff-FalseB TF Hset-iff-eq cls-TrueB-iff*)

5.2.2 The evaluation and logical conditions of the term model

The β-condition (BKK Definition 3.18(4)) inside H: a β-redex is Leibniz-equal to its reduct, via the rule $NK(\beta)$ applied to reflexivity.

lemma *Hset-beta*:
assumes *wAbs*: *cwff* $(\sigma \Rightarrow \tau)$ $(\mathbf{\Lambda}_\sigma\ b)$ **and** *wa*: *cwff* σ *a*
shows $((\mathbf{\Lambda}_\sigma\ b) \cdot a \doteq_\tau b\langle a\rangle) \in \mathit{Hset}\ \Phi$
proof (*rule Hset-deduct*[*OF con*Φ *fp*Φ, *of* {}])
let *?A* $= (\mathbf{\Lambda}_\sigma\ b) \cdot a$ **let** *?B* $= b\langle a\rangle$
have *wA*: *wff*$_\tau$(*?A*)
using *cwff-wff*[*OF wAbs*] *cwff-wff*[*OF wa*] **by** (*auto intro*: *wff-App*)
have *step*: (*?B* $\doteq_\tau$ *?B*) $\approx_{\mathrm{o}}$ (*?A* $\doteq_\tau$ *?B*)
by (*meson appL appR beq.sym beta cwff-wff wAbs wa wff-Leib wff-opn*)
show {} $\vdash$ *?A* $\doteq_\tau$ *?B*
by (*meson Beta cwff-def finite.emptyI freep-finite leib-refl local.step wAbs wa wff-opn*)
show *finite* {} **by** *simp*
show {} $\subseteq$ *Hset* Φ **by** *simp*
show *cwff* o (*?A* $\doteq_\tau$ *?B*)
by (*meson cwff-App cwff-def cwff-opn fvs-defs*(*5*) *wAbs wa wff-Leib*)
qed

The $L_\neg$ condition (BKK Figure 2) at the class level, from maximality and consistency of H.

lemma *cls-neg*: *cwff* o $\varphi \Longrightarrow$ *cls* $(\neg\ \varphi)$ = (*if cls* φ = *cls* $\top$ *then cls* $\bot$ *else cls* $\top$)
by (*meson Hset-maximal Hset-notboth cwff-App cwff-Neg cwff-def*
cls-FalseB-iff cls-TrueB-iff)

The $L_\vee$ condition: H treats disjunction disjunctively (BKK $\nabla_\vee$, Definition 6.19).

lemma *Hset-dis-iff*:
assumes *wp*: *cwff* o φ **and** *wq*: *cwff* o ψ
shows $\varphi \vee \psi \in \mathit{Hset}\ \Phi \longleftrightarrow \varphi \in \mathit{Hset}\ \Phi \vee \psi \in \mathit{Hset}\ \Phi$
proof
assume *d*: $\varphi \vee \psi \in \mathit{Hset}\ \Phi$
show $\varphi \in \mathit{Hset}\ \Phi \vee \psi \in \mathit{Hset}\ \Phi$
proof (*rule ccontr*)
assume $\neg\ (\varphi \in \mathit{Hset}\ \Phi \vee \psi \in \mathit{Hset}\ \Phi)$
hence *np*: $\neg\ \varphi \in \mathit{Hset}\ \Phi$ **and** *nq*: $\neg\ \psi \in \mathit{Hset}\ \Phi$
using *Hset-maximal*[*OF wp*] *Hset-maximal*[*OF wq*] **by** *blast+*
let *?F* $= \{\varphi \vee \psi, \neg\ \varphi, \neg\ \psi\}$
have *?F* $\vdash \varphi \vee \psi$ **by** (*auto intro*: *bprov.Hyp*)
moreover have *?F* $\cup \{\varphi\} \vdash \bot$
by (*metis Hyp NegE insertCI insert-is-Un sup.commute wff-FalseB*)
moreover have *?F* $\cup \{\psi\} \vdash \bot$
by (*metis Hyp NegE Un-insert-right insertCI insert-subset sup-ge1 wff-FalseB*)
ultimately have *?F* $\vdash \bot$
using *cwff-wff*[*OF wp*] *cwff-wff*[*OF wq*] **by** (*metis bprov.DisE*)
moreover have *con ?F*
by (*meson Hset-finite-con con*Φ *d empty-subsetI finite.emptyI finite.insertI*
fpΦ *insert-subsetI np nq*)
ultimately show *False* **by** (*simp add*: *con-def*)
qed

next
  show $\varphi \in \mathit{Hset}\ \Phi \vee \psi \in \mathit{Hset}\ \Phi \Longrightarrow \varphi \vee \psi \in \mathit{Hset}\ \Phi$
    by (*meson DisIL DisIR Hset-deduct Hyp bprov-finite con*Φ *cwff-App*
        cwff-Dis cwff-def fpΦ *wp wq*)
qed

lemma *cls-dis*:
  assumes *wp*: *cwff* o φ **and** *wq*: *cwff* o ψ
  shows *cls* $(\varphi \vee \psi) = (\mathit{if}\ \mathit{cls}\ \varphi = \mathit{cls}\ \top \vee \mathit{cls}\ \psi = \mathit{cls}\ \top\ \mathit{then}\ \mathit{cls}\ \top\ \mathit{else}\ \mathit{cls}\ \bot)$
proof −
  have *wd*: *cwff* o $(\varphi \vee \psi)$ **using** *wp wq* **by** (*auto simp*: *cwff-def*)
  show *?thesis* **by** (*simp add*: *Hset-dis-iff cls-FalseB-iff cls-TrueB-iff wd wp wq*)
qed

The $L^{\sigma}{}_{\forall}$ condition: H treats the quantifier universally — $NK(\Pi E)$ gives one direction, the witness property $\sim\nabla_{\exists}$ (BKK Definition 6.19) together with maximality the other.

lemma *Hset-pi-iff*:
  assumes *wf*: *cwff* $(\sigma \Rightarrow \mathrm{o})$ *f*
    shows $(\mathit{Pi}\ \sigma) \cdot f \in \mathit{Hset}\ \Phi \longleftrightarrow (\forall a.\ \mathit{cwff}\ \sigma\ a \longrightarrow f \cdot a \in \mathit{Hset}\ \Phi)$
proof
  assume *pf*: $(\mathit{Pi}\ \sigma) \cdot f \in \mathit{Hset}\ \Phi$
  have *step*: $f \cdot a \in \mathit{Hset}\ \Phi$ **if** *wa*: *cwff* σ *a* **for** *a*
    by (*meson Hset-maximal Hyp NegE PiE con*Φ *con-Hset con-def*
          cwff-App cwff-def fpΦ *local.wf pf wa wff-FalseB*)
  thus $\forall a.$ *cwff* σ *a* $\longrightarrow f \cdot a \in \mathit{Hset}\ \Phi$ **by** *blast*
next
  assume *all*: $\forall a.$ *cwff* σ *a* $\longrightarrow f \cdot a \in \mathit{Hset}\ \Phi$
  show $(\mathit{Pi}\ \sigma) \cdot f \in \mathit{Hset}\ \Phi$
  proof (*rule ccontr*)
    assume *npf*: $(\mathit{Pi}\ \sigma) \cdot f \notin \mathit{Hset}\ \Phi$
    have *wPf*: $\mathit{wff}_{\mathrm{o}}((\mathit{Pi}\ \sigma) \cdot f)$
      using *cwff-wff*[*OF wf*] **by** (*auto intro*: *wff-App wff-Pi*)
    have *cPf*: *fvs* $((\mathit{Pi}\ \sigma) \cdot f) = \{\}$
      using *cwff-closed*[*OF wf*] **by** *simp*
    have *n*: $\neg\ ((\mathit{Pi}\ \sigma) \cdot f) \in \mathit{Hset}\ \Phi$
       using *Hset-maximal*[*OF cwffI*[*OF wPf cPf*]] *npf* **by** *blast*
    have *wn*: $\mathit{wff}_{\mathrm{o}}(\neg\ ((\mathit{Pi}\ \sigma) \cdot f))$ **using** *wPf* **by** (*rule wff-Not*)
    have *cn*: *fvs* $(\neg\ ((\mathit{Pi}\ \sigma) \cdot f)) = \{\}$ **using** *cPf* **by** *simp*
    obtain *c* **where** *nc*: $\neg\ (f \cdot (c^{p}{}_{\sigma})) \in \mathit{Hset}\ \Phi$
      using *Hset-saturated*[*OF con*Φ *fp*Φ *cwffI*[*OF wn cn*] *n*] **by** *blast*
    have *inc*: $f \cdot (c^{p}{}_{\sigma}) \in \mathit{Hset}\ \Phi$
      using *all*[*rule-format, of* $c^{p}{}_{\sigma}$] *cwff-Par*[*of* σ *c*] **by** *simp*
    show *False*
     using *Hset-notboth*[*OF wff-App*[*OF cwff-wff*[*OF wf*] *wff-Par*] *inc*] *nc* **by** *simp*
  qed
qed

lemma *cls-pi*:

assumes *wf*: *cwff* (*σ*⇒o) *f*
shows *cls* ((*Pi σ*) · *f*) = (*if* (∀ *a*. *cwff σ a* ⟶ *Ap* (*cls f*) (*cls a*) = *cls* ⊤)
then cls ⊤ *else cls* ⊥)
proof −
have *wPf*: *cwff* o ((*Pi σ*) · *f*)
using *wf* **by** (*auto simp*: *cwff-def intro*: *wff-App wff-Pi*)
have *e*: (*Ap* (*cls f*) (*cls a*) = *cls* ⊤) = (*f* · *a* ∈ *Hset* Φ) **if** *a*: *cwff σ a* **for** *a*
using *Ap-cls*[*OF wf a*] *cls-TrueB-iff*[*OF cwff-App*[*OF wf a*]] **by** *simp*
have *inner*: (∀ *a*. *cwff σ a* ⟶ *Ap* (*cls f*) (*cls a*) = *cls* ⊤)
= (∀ *a*. *cwff σ a* ⟶ *App f a* ∈ *Hset* Φ) **using** *e* **by** *blast*
show *?thesis*
using *cls-TrueB-iff*[*OF wPf*] *cls-FalseB-iff*[*OF wPf*] *Hset-pi-iff*[*OF wf*] *inner*
by (*cases* ∀ *a*. *cwff σ a* ⟶ *f* · *a* ∈ *Hset* Φ) *auto*
qed

The β-condition transfers membership: a redex of type o is in H exactly when its reduct is (via the Leibniz bridge).

lemma *Hset-beta-iff*:
assumes *cwff* (*σ*⇒o) (**Λ**$_\sigma$ *b*) **and** *cwff σ a*
shows (**Λ**$_\sigma$ *b*) · *a* ∈ *Hset* Φ ⟷ *b*⟨*a*⟩ ∈ *Hset* Φ
by (*meson Hset-beta Hset-leib-mp Hset-leib-sym con*Φ *cwff-App cwff-opn fp*Φ *assms*)

Type uniqueness of the classes (BKK: the domains are disjoint by type).

lemma *cls-type*:
cwff σ s ⟹ *cwff τ t* ⟹ *cls s* = *cls t* ⟹ *σ* = *τ*
by (*smt* (*verit*, *ccfv-SIG*) *cls-def cls-self cwff-unique mem-Collect-eq*)

Property q (BKK Definition 3.46): the Leibniz combinator itself is the identity relation of the term model, by ⟦*cwff ?σ ?A*; *cwff ?σ ?B*⟧ ⟹ (*cls ?A* = *cls ?B*) = (*?A* ≐$_{?\sigma}$ *?B* ∈ *Hset* Φ).

lemma *cwff-Leib*: *cwff* (*σ*⇒*σ*⇒o) (*Leib σ*) **by** (*simp add*: *cwff-def*)

Property f (functionality, BKK Definition 3.46) via the rule *NK*(*f*): two functions that agree on every class are Leibniz-equal in H. Saturation enters through *cwff* (*?σ* ⇒ o) *?f* ⟹ (*Pi ?σ* · *?f* ∈ *Hset* Φ) = (∀ *a*. *cwff ?σ a* ⟶ *?f* · *a* ∈ *Hset* Φ) to establish the Π-premise of *NK*(*f*).

lemma *cls-ext*:
assumes *wg*: *cwff* (*σ*⇒*τ*) *g* **and** *wh*: *cwff* (*σ*⇒*τ*) *h*
and *ag*: ⋀*a*. *cwff σ a* ⟹ *Ap* (*cls g*) (*cls a*) = *Ap* (*cls h*) (*cls a*)
shows *cls g* = *cls h*
proof −
have *lcg*: *lc g* **and** *lch*: *lc h* **using** *wg wh*
by (*auto intro*: *cwff-lc*)
let *?body* = *g* · (*Bnd 0*) ≐$_\tau$ *h* · (*Bnd 0*)
have *opnb*: *?body*⟨*u*⟩ = (*g* · *u* ≐$_\tau$ *h* · *u*) **for** *u* **using** *lcg lch*
by *simp*
— the abstracted body is a closed well-formed predicate

```
  have wB: cwff (σ⇒o) (Λσ ?body)
  proof −
    have wff_o(?body⟨x^f σ⟩) for x unfolding opnb
      using cwff-wff[OF wg] cwff-wff[OF wh]
      by (auto intro: wff-App wff-Fre)
    hence wff_{σ⇒o}(Λσ ?body) by (rule wff-AbsI)
    moreover have fvs (Λσ ?body) = {}
      using cwff-closed[OF wg] cwff-closed[OF wh]
      by (simp add: Leib-def)
    ultimately show ?thesis by (simp add: cwff-def)
  qed
  — pointwise agreement puts every instance of the body in H
  have inst: (Λσ ?body) · a ∈ Hset Φ if a: cwff σ a for a
    by (smt (verit, ccfv-threshold) Ap-cls Hset-beta-iff ag
        cls-eq-iff cwff-App opnb that wB wg wh)
  — hence the Π-sentence is in H, and NK(f) yields the equation
  have PiH: (Pi σ) · (Λσ ?body) ∈ Hset Φ
    using Hset-pi-iff[OF wB] inst by blast
  have (g ≐_{σ⇒τ} h) ∈ Hset Φ
  proof (rule Hset-deduct[OF conΦ fpΦ, of {(Pi σ) · (Λσ
      ?body)}])
    show finite {(Pi σ) · (Λσ ?body)} by simp
    show {(Pi σ) · (Λσ ?body)} ⊆ Hset Φ using PiH by simp
    have {(Pi σ) · (Λσ ?body)} ⊢ Πσ ?body
      by (auto intro: bprov.Hyp simp: Forall-def)
    thus {(Pi σ) · (Λσ ?body)} ⊢ g ≐_{σ⇒τ} h
      using FuncE cwff-wff wg wh by blast
    show cwff o (g ≐_{σ⇒τ} h) by (metis cwff-App wg wh cwff-Leib)
  qed
  thus ?thesis using cls-eq-iff[OF wg wh] by simp
qed
```

The description condition of the term model (beyond BKK; cf. Andrews 1972): a class that provably behaves as the singleton of a is mapped by *Iota* σ to the class of a. Functional extensionality $NK(f)$ reduces the pointwise hypothesis to a Leibniz equation with the literal singleton $(Leib\ \sigma) \cdot a$, which the axiom $NK(\iota)$ describes.

```
lemma cls-desc:
  assumes wf: cwff (σ⇒o) f and wa: cwff σ a
      and sing: ⋀b. cwff σ b ⟹ (Ap (cls f) (cls b) = cls ⊤) = (cls b = cls a)
    shows cls ((Iota σ) · f) = cls a
proof −
  have lcf: lc f and lca: lc a using wf wa
    by (auto intro: cwff-lc)
  let ?body = f · (Bnd 0) ≐_o ((Leib σ) · a) · (Bnd 0)
  have opnb: ?body⟨u⟩ = (f · u ≐_o ((Leib σ) · a) · u) for u
    using lcf lca by simp
  have wB: cwff (σ⇒o) (Λσ ?body)
  proof −
```

```
  have wff_o(?body⟨x^f σ⟩) for x unfolding opnb
    using cwff-wff[OF wf] cwff-wff[OF wa]
    by (auto intro: wff-App wff-Fre)
  hence wff_{σ⇒o}(Λ_σ ?body) by (rule wff-AbsI)
  moreover have fvs (Λ_σ ?body) = {} using cwff-def local.wf wa
    by auto
  ultimately show ?thesis by (simp add: cwff-def)
 qed
 — the pointwise singleton facts land in H
 have inst: (Λ_σ ?body) · b ∈ Hset Φ if b: cwff σ b for b
  by (metis (no-types, opaque-lifting) Ap-cls Hset-beta-iff Hset-iff-eq cls-TrueB-iff
        cls-eq-iff cwff-App cwff-Leib local.wf opnb sing that wB wa)
 have PiH: (Pi σ) · (Λ_σ ?body) ∈ Hset Φ
  using Hset-pi-iff[OF wB] inst by blast
 — NK(f) reduces to the literal singleton, NK(ι) describes it
 have (((Iota σ) · f) ≐_σ a) ∈ Hset Φ
 proof (rule Hset-deduct[OF conΦ fpΦ, of {(Pi σ) · (Λ_σ ?body)}])
  show finite {(Pi σ) · (Λ_σ ?body)} by simp
  show {(Pi σ) · (Λ_σ ?body)} ⊆ Hset Φ using PiH by simp
  let ?F = {(Pi σ) · (Λ_σ ?body)}
  have fpF: freep ?F by (rule freep-finite) simp
  have PiD: ?F ⊢ Π_σ ?body
    by (auto intro: bprov.Hyp simp: Forall-def)
  show ?F ⊢ (Iota σ) · f ≐_σ a
    by (rule leib-trans[OF
      leib-cong2[OF bprov.FuncE[OF PiD cwff-wff[OF wf]
            cwff-wff[OF cwff-App[OF cwff-Leib wa]]] fpF
          cwff-wff[OF wf] cwff-wff[OF cwff-App[OF
          cwff-Leib wa]]
        wff-Iota] bprov.Desc[OF cwff-wff[OF wa]] fpF
        wff-App[OF wff-Iota cwff-wff[OF wf]]
      wff-App[OF wff-Iota cwff-wff[OF cwff-App[OF cwff-Leib
        wa]]] cwff-wff[OF wa]])
  show cwff o ((Iota σ) · f ≐_σ a)
    by (metis wa cwff-App local.wf cwff-Leib cwff-Iota)
 qed
 thus ?thesis
  using cls-eq-iff[OF cwff-App[OF cwff-Iota wf] wa] by simp
qed
```

Primitive equality in H (BKK Remark 7.9): reflexivity is $\nabla_{=_r}$ via $NK(=_r)$, and primitive and Leibniz equality coincide in H — $\rightarrow$ via $NK(=_l)$ ($\nabla_{=_{\doteq}}$), $\leftarrow$ by Leibniz substitution into $\lambda x.\ a =_\sigma x$ from reflexivity.

```
lemma Hset-peq-iff:
 assumes wa: cwff σ a and wb: cwff σ b
  shows (a =_σ b) ∈ Hset Φ ⟷ (a ≐_σ b) ∈ Hset Φ
proof
 assume p: (a =_σ b) ∈ Hset Φ show (a ≐_σ b) ∈ Hset Φ
 proof (rule Hset-deduct[OF conΦ fpΦ, of {a =_σ b}])
```

```
    show finite {a =σ b} by simp
    show {a =σ b} ⊆ Hset Φ using p by simp
    show {a =σ b} ⊢ a ≐σ b
      by (auto intro: bprov.EqL bprov.Hyp)
    show cwff o (a ≐σ b)
      using cwff-App cwff-Leib wa wb by blast
  qed
next
  assume l: (a ≐σ b) ∈ Hset Φ show (a =σ b) ∈ Hset Φ
  proof (rule Hset-deduct[OF conΦ fpΦ, of {a ≐σ b}])
    show finite {a ≐σ b} by simp
    show {a ≐σ b} ⊆ Hset Φ using l by simp
    let ?F = {a ≐σ b}  let ?P = Λσ (a =σ Bnd 0)
    have lca: lc a using wa by (rule cwff-lc)
    have wP: wff_{σ⇒o}(?P)
      by (rule wff-AbsI)
         (auto simp: opn-lc[OF lca] intro: cwff-wff[OF wa] wff-Fre)
    have fpF: freep ?F by (rule freep-finite) simp
    have hyp: ?F ⊢ a ≐σ b by (auto intro: bprov.Hyp)
    have PA: ?P · a ≈o (a =σ a)
      using beq.beta[OF wP cwff-wff[OF wa]] by (simp add: opn-lc[OF lca])
    have PB: ?P · b ≈o (a =σ b)
      using beq.beta[OF wP cwff-wff[OF wb]] by (simp add: opn-lc[OF lca])
    have ?F ⊢ a =σ a by (rule bprov.EqR[OF cwff-wff[OF wa]])
    hence ?F ⊢ ?P · a by (rule bprov.Beta[OF beq.sym[OF PA]])
    hence ?F ⊢ ?P · b
      by (rule leib-subst[OF hyp fpF cwff-wff[OF wa] cwff-wff[OF wb] wP])
    thus ?F ⊢ a =σ b by (rule bprov.Beta[OF PB])
    show cwff o (a =σ b)
      using cwff-wff[OF wa] cwff-wff[OF wb] cwff-closed[OF wa] cwff-closed[OF
wb]
      by (auto simp: cwff-def)
  qed
qed

end
```

5.2.3 Model existence (BKK Lemma 7.5 / Theorem 7.6)

The Hintikka set induces a term evaluation: the class map *cls* together with the term-model domains and application satisfies every *valuation* condition. Via the bridge *valuation* $\subseteq$ *bkk-model* of Section 2, every *NK*-consistent set of sentences therefore has a model in the class $\mathcal{M}_{\beta f b}$ — BKK's model-existence theorem.

```
sublocale hintikka-model ⊆ V: valuation Dm Ap cls
proof
  show cwff (σ⇒τ) (Λσ b) ⟹ cwff σ a ⟹ Ap (cls (Λσ b)) (cls a) = cls (b⟨a⟩)
    for σ τ b a by (metis (no-types, lifting) Ap-cls Hset-beta cls-eq-iff cwff-App
```

cwff-opn)
show *cwff* (*σ⇒τ*) *g* ⟹ *cwff* (*σ⇒τ*) *h* ⟹ (⋀*a*. *cwff σ a*
⟹ *Ap* (*cls g*) (*cls a*) = *Ap* (*cls h*) (*cls a*)) ⟹ *cls g* = *cls h*
for *σ τ g h* **by** (*rule cls-ext*)
show *cwff σ a* ⟹ *cwff σ b* ⟹ (*Ap* (*Ap* (*cls* (*Eq σ*)) (*cls a*)) (*cls b*)
= *cls* ⊤) = (*cls a* = *cls b*) **for** *σ a b*
proof −
assume *wa*: *cwff σ a* **and** *wb*: *cwff σ b*
have *2*: *Ap* (*cls* ((*Eq σ*) · *a*)) (*cls b*) = *cls* (*a* =*σ b*)
using *Ap-cls*[*OF cwff-App*[*OF cwff-Eq wa*] *wb*] **by** *simp*
have *wab*: *cwff* o (*a* =*σ b*)
using *cwff-App*[*OF cwff-App*[*OF cwff-Eq wa*] *wb*] **by** *simp*
show (*Ap* (*Ap* (*cls* (*Eq σ*)) (*cls a*)) (*cls b*) = *cls* ⊤) = (*cls a* = *cls b*)
using *Ap-cls*[*OF cwff-Eq wa*] *2 cls-TrueB-iff*[*OF wab*]
Hset-peq-iff[*OF wa wb*] *cls-eq-iff*[*OF wa wb*] **by** *simp*
qed
qed(*auto simp*: *Ap-cls Dm-def cls-type TF cls-desc cls-pi cls-dis cls-neg cls-bool*)

5.2.4 The truth lemma

context *hintikka-model*
begin

THE TRUTH LEMMA (the satisfaction claim of BKK Theorem 6.33; the underlying Hintikka properties are BKK Lemma 6.21): under any domain-respecting assignment a sentence denotes its own class, and it denotes the truth value *cls* ⊤ of the term model exactly when it belongs to *H*.

theorem *truth-lemma*:
cwff o *φ* ⟹ *V.vresp ξ* ⟹ *V.Ev ξ φ* = *cls* ⊤ ⟷ *φ* ∈ *Hset* Φ
using *cls-TrueB-iff msub-closed cwff-closed V.Ev-def* **by** *metis*

end

5.2.5 Completeness (BKK Corollary 7.7)

$NK_{\beta f b}$ with $NK(\iota)$ is complete for the class $\mathcal{M}_{\beta f b}$ of Σ-models with description (the description-enriched $\mathcal{M}_{\beta f b}$, cf. Andrews 1972): a sentence that is valid in every such model of a sufficiently Σ-pure (*freep*, BKK Definition 6.3) set of sentences Φ (it suffices to assume validity over the models carried by *'p tm set* — in particular the term model) is derivable from Φ. Unlike BKK, who allow signatures of any infinite cardinality $\aleph_s$ (BKK Remark 3.16), we fix a countable type of parameter names (*'p* :: {*countable*, *infinite*}); the enumeration of sentences in the extension lemma rests on it. The proof is by contraposition, BKK's argument: if $\Phi \vdash A$ fails then $\Phi \cup \{\neg A\}$ is *NK*-consistent (*NK*(*Contr*)), extends to a maximal saturated set (BKK's abstract extension lemma 6.32), and its term model — a general model by model existence — satisfies Φ but refutes A, contradicting validity.

```
lemma fp0: freep ({} :: 'p::{countable,infinite} tm set)
  by (rule freep-finite[OF finite.emptyI])

theorem completeness:
  fixes Φ :: 'p::{countable,infinite} tm set and A :: 'p tm
  assumes c: cwff ∘ A and fp: freep Φ
      and valid: ⊨('p tm set) A
      and sen: ⋀B. B ∈ Φ ⟹ cwff ∘ B
  shows Φ ⊢ A
proof (rule ccontr)
  assume nd: ¬ Φ ⊢ A
  note wA = cwff-wff[OF c] and cA = cwff-closed[OF c]
  let ?Ψ = insert (¬ A) Φ
  have conΨ: con ?Ψ using bprov.simps con-def nd wA by fastforce
  have fpΨ: freep ?Ψ by (rule freep-add[OF fp])
  interpret H: hintikka-model ?Ψ by unfold-locales (rule conΨ fpΨ)+
  — a domain-respecting assignment for the term model
  define ξ :: nat ⇒ ty ⇒ 'p tm set where
    ξ ≡ λn τ. H.cls (undefined^p_τ)
  have r: H.V.vresp ξ
    by (auto simp: ξ-def H.V.vresp-def intro: H.Dm-cls cwff-Par)
  have ra: app-struct.asg H.Dm ξ using r
    by (simp add: H.V.bkkA.asg-def H.V.vresp-def)
  have bm: bkk-model H.Dm H.Ap H.V.Ev (λa. a = H.cls ⊤)
    by intro-locales
  — the term model satisfies Φ by the truth lemma
  have sat: ∀B∈Φ. H.V.Ev ξ B = H.cls ⊤
    using H.truth-lemma[OF - r] sen Phi-sub-Hset[of ?Ψ]
    by (auto simp: cwff-def)
  — validity at the term model forces A ∈ H
  have H.V.Ev ξ A = H.cls ⊤
    using valid unfolding bkk-valid-def rel-truth-def using bm ra sat by blast
  hence AH: A ∈ Hset ?Ψ using H.truth-lemma[OF - r] wA cA
    by (simp add: cwff-def)
  — but ¬A ∈ Ψ ⊆ H — contradiction with consistency of H
  have ¬ A ∈ Hset ?Ψ using Phi-sub-Hset[of ?Ψ] by auto
  thus False using H.Hset-notboth[OF wA AH] by simp
qed
```

Soundness (BKK Theorem 7.3) and completeness (BKK Corollary 7.7) are statements about the *same* class of models, so together they characterise derivability semantically: a sentence is derivable from Φ exactly when it holds in every model of Φ in the class $\mathcal{M}_{\beta fb}$ over the term carrier.

The semantic characterisation of derivability from the empty hypothesis set (BKK Corollary 7.7 at Φ = {}, combined with Theorem 7.3): a sentence is derivable iff it is valid in every Σ-model of the class $\mathcal{M}_{\beta fb}$ (over the carrier of the term model; the general hypothesis-set form is *completeness*). The soundness direction is *soundness-bkk*; the completeness direction only

shrinks the quantification to the canonical models via the sublocale bridge.

theorem *derivable-iff-valid*:
fixes $A :: 'p::\{countable,infinite\}\ tm$
assumes *cwff* o A
shows $\vdash A \longleftrightarrow \models('p\ tm\ set)\ A$
by (*metis* (*mono-tags*, *lifting*) *assms bkk-valid-def completeness empty-iff fp0 rel-truth-def soundness-bkk*)

5.2.6 Further derived Leibniz rules

Leibniz modus ponens: transport a theorem along a proven Leibniz equation (via $NK(\beta)$ and Leibniz substitution into the identity predicate).

lemma *leib-mp*:
assumes *AB*: $\Phi \vdash A \doteq_o B$ **and** *A*: $\Phi \vdash A$ **and** *fp*: *freep* Φ
and *wA*: $wff_o(A)$ **and** *wB*: $wff_o(B)$
shows $\Phi \vdash B$
proof −
let *?P* = $\mathbf{\Lambda}_o\ (Bnd\ 0) :: 'p\ tm$
have *wP*: $wff_{o \Rightarrow o}(?P)$ **by** (*rule wff-AbsI*) (*simp add*: *wff-Fre*)
have *bA*: $?P \cdot A \approx_o A$ **using** *beq.beta*[*OF wP wA*] **by** *simp*
have *bB*: $?P \cdot B \approx_o B$ **using** *beq.beta*[*OF wP wB*] **by** *simp*
have $\Phi \vdash ?P \cdot A$ **by** (*rule bprov.Beta*[*OF beq.sym*[*OF bA*] *A*])
hence $\Phi \vdash ?P \cdot B$ **by** (*rule leib-subst*[*OF AB fp wA wB wP*])
thus *?thesis* **by** (*rule bprov.Beta*[*OF bB*])
qed

From Leibniz to primitive equality (BKK Remark 7.9; by Leibniz substitution into $\mathbf{\Lambda}x.\ A =_\alpha x$ from $NK(=_r)$-reflexivity; the converse is the rule $NK(=_l)$).

lemma *leib-to-peq*:
assumes *AB*: $\Phi \vdash A \doteq_\alpha B$ **and** *fp*: *freep* Φ
and *wA*: $wff_\alpha(A)$ **and** *wB*: $wff_\alpha(B)$
shows $\Phi \vdash A =_\alpha B$
proof −
let *?P* = $\mathbf{\Lambda}_\alpha\ (A =_\alpha Bnd\ 0)$
have *wP*: $wff_{\alpha \Rightarrow o}(?P)$
by (*rule wff-AbsI*) (*auto simp*: *opn-lc*[*OF wff-lc*[*OF wA*]] *intro*!: *wA wff-Fre*)
have *bA*: $?P \cdot A \approx_o (A =_\alpha A)$
using *beq.beta*[*OF wP wA*] **by** (*simp add*: *opn-lc*[*OF wff-lc*[*OF wA*]])
have *bB*: $?P \cdot B \approx_o (A =_\alpha B)$
using *beq.beta*[*OF wP wB*] **by** (*simp add*: *opn-lc*[*OF wff-lc*[*OF wA*]])
have $\Phi \vdash A =_\alpha A$ **by** (*rule bprov.EqR*[*OF wA*])
hence $\Phi \vdash ?P \cdot A$ **by** (*rule bprov.Beta*[*OF beq.sym*[*OF bA*]])
hence $\Phi \vdash ?P \cdot B$ **by** (*rule leib-subst*[*OF AB fp wA wB wP*])
thus *?thesis* **by** (*rule bprov.Beta*[*OF bB*])
qed

The diagonal contradiction: a proposition Leibniz-equal to its own negation refutes the context (by excluded middle and Leibniz modus ponens).

lemma *leib-neg-contra*:
  assumes *E*: $\Phi \vdash A \doteq_{\mathrm{o}} (\neg\ A)$ **and** *fp*: *freep* Φ **and** *wA*: $\mathit{wff}_{\mathrm{o}}(A)$
  shows $\Phi \vdash \bot$
proof −
  have *br1*: $\Phi \cup \{\neg\ A\} \vdash \bot$
  proof −
    have *fp1*: *freep* $(\Phi \cup \{\neg\ A\})$ **using** *freep-add*[*OF fp*] **by** *simp*
    have *hy*: $\Phi \cup \{\neg\ A\} \vdash \neg\ A$ **by** (*auto intro*: *bprov.Hyp*)
    have *e*: $\Phi \cup \{\neg\ A\} \vdash A \doteq_{\mathrm{o}} (\neg\ A)$
      by (*rule bprov-weaken*[*OF E - fp1*]) *auto*
    have $\Phi \cup \{\neg\ A\} \vdash (\neg\ A) \doteq_{\mathrm{o}} A$
      by (*rule leib-sym*[*OF e fp1 wA wff-Not*[*OF wA*]])
    hence $\Phi \cup \{\neg\ A\} \vdash A$
      by (*rule leib-mp*[*OF - hy fp1 wff-Not*[*OF wA*] *wA*])
    thus *?thesis* **by** (*rule bprov.NegE*[*OF hy - wff-FalseB*])
  qed
  have *br2*: $\Phi \cup \{A\} \vdash \bot$
  proof −
    have *fp2*: *freep* $(\Phi \cup \{A\})$ **using** *freep-add*[*OF fp*] **by** *simp*
    have *hy*: $\Phi \cup \{A\} \vdash A$ **by** (*auto intro*: *bprov.Hyp*)
    have *e*: $\Phi \cup \{A\} \vdash A \doteq_{\mathrm{o}} (\neg\ A)$
      by (*rule bprov-weaken*[*OF E - fp2*]) *auto*
    have $\Phi \cup \{A\} \vdash \neg\ A$
      by (*rule leib-mp*[*OF e hy fp2 wA wff-Not*[*OF wA*]])
    thus *?thesis* **by** (*rule bprov.NegE*[*OF - hy wff-FalseB*])
  qed
  show *?thesis*
    by (*rule bprov.DisE*[*OF bprov-em*[*OF wA*] *br1 br2 wff-Not*[*OF wA*] *wA*])
qed

Application of a primitive equation, and β-reduction on the right of $\doteq$.

lemma *peq-app*: $\Phi \vdash C =_{\alpha \Rightarrow \beta} D \Longrightarrow \mathit{freep}\ \Phi \Longrightarrow \mathit{wff}_{\alpha \Rightarrow \beta}(C) \Longrightarrow \mathit{wff}_{\alpha \Rightarrow \beta}(D)$
  $\Longrightarrow \mathit{wff}_{\alpha}(A) \Longrightarrow \Phi \vdash (C \cdot A) \doteq_{\beta} (D \cdot A)$
  by (*rule leib-cong1*[*OF bprov.EqL*])
lemma *leib-reduce-right*: $\Phi \vdash A \doteq_{\mathrm{o}} B \Longrightarrow B \approx_{\mathrm{o}} B' \Longrightarrow \mathit{wff}_{\mathrm{o}}(A) \Longrightarrow \Phi \vdash A \doteq_{\mathrm{o}} B'$
  by (*metis beq.appR bprov.Beta wff-App wff-Leib*)

The same three steps for *primitive* equality (via $NK(=_l)$ in and *leib-to-peq* out): application to an argument, β-reduction on the right, and the diagonal contradiction $A = \neg\ A \Longrightarrow \bot$.

5.3 Completeness at every signature and carrier

This part strengthens the completeness theorem from the term-model carrier to *arbitrary* infinite value carriers, by an explicit model-embedding construction: every Σ-model of the class $\mathcal{M}_{\beta f b}$ whose total domain embeds injectively into a carrier $'u$ has an isomorphic copy on $'u$, and satisfaction is invariant

under the embedding. Since the term model has a countable total domain (the language is countable), it embeds into every infinite carrier, so validity over any infinite carrier suffices for derivability. Completeness is then extended further, to open formulas and to signatures with infinitely many parameters, and the development closes with the main theorems stated in self-contained notation.

5.3.1 A countermodel for every underivable sentence

The completeness construction, packaged as an explicit countermodel: if a sentence is not derivable, the Hintikka term model refutes it. The total domain of the term model is countable — the domains are $\sim$-classes of closed wffs.

lemma *refuting-term-model*:
  fixes A :: $'p$::{*countable*,*infinite*} *tm*
  assumes *c*: *cwff* ∘ A **and** *nd*: $\neg\vdash A$
  obtains *Dm Ap* **and** *Ee* :: (*nat* ⇒ *ty* ⇒ $'p$ *tm set*) ⇒ $'p$ *tm* ⇒ $'p$ *tm set* **and** *vl* ξ
  where *bkk-model Dm Ap Ee vl countable* {x. ∃τ. *Dm* τ x}
      app-struct.asg Dm ξ ¬ *vl* (*Ee* ξ A)
proof −
  note wA = *cwff-wff*[*OF c*] **and** cA = *cwff-closed*[*OF c*]
  have *con*Ψ: *con* {¬ A} **using** *bprov.simps con-def nd wA* **by** *fastforce*
  have *fp*Ψ: *freep* {¬ A} **by** (*intro freep-finite*) *simp*
  interpret H: *hintikka-model* {¬ A}
    by *unfold-locales* (*rule con*Ψ *fp*Ψ)+
  define ξ :: *nat* ⇒ *ty* ⇒ $'p$ *tm set* **where**
    $\xi \equiv \lambda n\ \tau.\ H.cls\ (undefined^p{}_\tau)$
  have *r*: $H.V.vresp\ \xi$
    by (*auto simp*: ξ-*def* $H.V.vresp$-*def intro*: H.*Dm-cls cwff-Par*)
  have *ra*: *app-struct.asg* $H.Dm\ \xi$
    using *r* **by** (*simp add*: $H.V.bkkA.asg$-*def* $H.V.vresp$-*def*)
  have *bm*: *bkk-model* $H.Dm$ $H.Ap$ $H.V.Ev$ ($\lambda a.\ a = H.cls\ \top$)
    by *intro-locales*
  — the model refutes A: $\neg A \in H$, so $A \notin H$, so the truth lemma denies A
  have $\neg A \in Hset$ {¬ A} **using** *Phi-sub-Hset*[*of* {¬ A}] **by** *auto*
  hence $A \notin Hset$ {¬ A} **using** H.*Hset-notboth*[*OF wA*] **by** *blast*
  hence *ref*: $H.V.Ev\ \xi\ A \neq H.cls\ \top$ **using** H.*truth-lemma*[*OF c r*] **by** *simp*
  — the total domain is contained in the countable range of *cls*
  have {x. ∃τ. $H.Dm\ \tau\ x$} ⊆ *range* $H.cls$ **by** (*auto simp*: H.*Dm-def*)
  hence *cnt*: *countable* {x. ∃τ. $H.Dm\ \tau\ x$}
    by (*rule countable-subset*) *simp*
  show *?thesis* **by** (*rule that*[*OF bm cnt ra*]) (*use ref* **in** *simp*)
qed

5.3.2 Model embedding: satisfaction is carrier-independent

A Σ-model over a carrier $'v$ whose total domain maps injectively into a carrier $'u$ has an isomorphic copy over $'u$; every refutation transfers. All model conditions are pointwise, so the transfer needs no induction on terms.

lemma *bkk-model-embed*:
  fixes $Dm :: ty \Rightarrow 'v \Rightarrow bool$ **and** $i :: 'v \Rightarrow 'u$
    and $A :: 'p\ tm$
  assumes M: *bkk-model Dm Ap Ee vl*
    and *inj*: *inj-on i* $\{x.\ \exists \tau.\ Dm\ \tau\ x\}$
    and wA: $wff_{\mathrm{O}}(A)$
    and *xi*: *app-struct.asg Dm* ξ **and** nA: $\neg\ vl\ (Ee\ \xi\ A)$
  obtains $Dm'\ Ap'$ **and** $Ee' :: (nat \Rightarrow ty \Rightarrow 'u) \Rightarrow 'p\ tm \Rightarrow 'u$ **and** $vl'\ \xi'$
  where *bkk-model* $Dm'\ Ap'\ Ee'\ vl'$ *app-struct.asg* $Dm'\ \xi'\ \neg\ vl'\ (Ee'\ \xi'\ A)$
proof −
  interpret M: *bkk-model Dm Ap Ee vl* **by** (*rule M*)
  define D **where** $D = \{x.\ \exists \tau.\ Dm\ \tau\ x\}$
  have *DmD*: $Dm\ \tau\ x \Longrightarrow x \in D$ **for** $\tau\ x$ **by** (*auto simp*: *D-def*)
  have *iinj*: $x \in D \Longrightarrow y \in D \Longrightarrow i\ x = i\ y \Longrightarrow x = y$ **for** $x\ y$
    using *inj* **by** (*auto simp*: *D-def inj-on-def*)
  define *bk* **where** $bk = (\lambda u.\ SOME\ x.\ x \in D \wedge i\ x = u)$
  have *cancel*: $x \in D \Longrightarrow bk\ (i\ x) = x$ **for** x
    unfolding *bk-def* **by** (*rule some-equality*) (*auto intro*: *iinj*)
  define Dm' **where** $Dm' = (\lambda \tau\ u.\ \exists x.\ Dm\ \tau\ x \wedge u = i\ x)$
  have $Dm'I$: $Dm\ \tau\ x \Longrightarrow Dm'\ \tau\ (i\ x)$ **for** $\tau\ x$ **by** (*auto simp*: Dm'*-def*)
  define *pull* $:: (nat \Rightarrow ty \Rightarrow 'u) \Rightarrow nat \Rightarrow ty \Rightarrow 'v$ **where**
    pull $\equiv \lambda \xi'\ n\ \tau.$ *if* $\exists x.\ Dm\ \tau\ x \wedge i\ x = \xi'\ n\ \tau$
                  then SOME $x.\ Dm\ \tau\ x \wedge i\ x = \xi'\ n\ \tau$
                  else SOME $x.\ Dm\ \tau\ x$
  have *pull-asg*: *M.asg* (*pull* ξ') **for** ξ'
    unfolding *M.asg-def pull-def*
    by (*auto intro*: *someI-ex M.as-nonempty someI2-ex*)
  have *pull-agree*: $Dm'\ \tau\ (\xi'\ n\ \tau) \Longrightarrow i\ (pull\ \xi'\ n\ \tau) = \xi'\ n\ \tau$
    for $\xi'\ n\ \tau$
    unfolding *pull-def* Dm'*-def* **by** (*auto intro*: *someI2-ex*)
  have *pull-push*: *pull* $(\lambda n\ \tau.\ i\ (\xi\ n\ \tau)) = \xi$
  proof (*intro ext*)
    fix $n\ \tau$
    have d: $Dm\ \tau\ (\xi\ n\ \tau)$ **using** *xi* **by** (*simp add*: *M.asg-def*)
    have $\exists x.\ Dm\ \tau\ x \wedge i\ x = i\ (\xi\ n\ \tau)$ **using** d **by** *blast*
    moreover have $\bigwedge x.\ Dm\ \tau\ x \Longrightarrow i\ x = i\ (\xi\ n\ \tau) \Longrightarrow x = \xi\ n\ \tau$
      by (*auto intro*: *iinj DmD d*)
    ultimately show *pull* $(\lambda n\ \tau.\ i\ (\xi\ n\ \tau))\ n\ \tau = \xi\ n\ \tau$
      unfolding *pull-def* **by** *auto*
  qed
  define Ap' **where** $Ap' \equiv \lambda u\ v.\ i\ (Ap\ (bk\ u)\ (bk\ v))$
  define Ee' **where** $Ee' \equiv \lambda \xi'\ B.\ i\ (Ee\ (pull\ \xi')\ B)$
  define vl' **where** $vl' \equiv \lambda u.\ vl\ (bk\ u)$
  have $Ap'I$: $x \in D \Longrightarrow y \in D \Longrightarrow Ap'\ (i\ x)\ (i\ y) = i\ (Ap\ x\ y)$ **for** $x\ y$

```
    by (simp add: Ap'-def cancel)
  have vl'I: x ∈ D ⟹ vl' (i x) ⟷ vl x for x
    by (simp add: vl'-def cancel)
  have EeD: wff_τ(B) ⟹ Ee (pull ξ') B ∈ D for τ B ξ'
    by (auto intro: DmD M.ev-type[OF - pull-asg])
  have ApD: Dm (α ⇒ β) f ⟹ Dm α a ⟹ Ap f a ∈ D for α β f a
    by (auto intro: DmD M.as-appTy)
— the image is an applicative structure; interpreting it makes the specialised asg
equation available
  have AS': app-struct Dm' Ap'
  proof (unfold-locales, goal-cases)
    case (1 α) show ?case using M.as-nonempty Dm'I by (metis Dm'-def)
  next
    case (2 α β f a) thus ?case
      by (auto simp: Dm'-def Ap'I DmD intro: M.as-appTy)
  qed
  interpret A': app-struct Dm' Ap' by (rule AS')
  — the image is a model: every condition transfers pointwise
  have BM: bkk-model Dm' Ap' Ee' vl'
  proof (unfold-locales, goal-cases)
    case 1 thus ?case
      by (auto simp: Ee'-def intro: Dm'I M.ev-type[OF - pull-asg])
  next
    case (2 ξ' n σ)
    hence Dm' σ (ξ' n σ) by (simp add: A'.asg-def)
    thus ?case by (simp add: Ee'-def M.ev-var[OF pull-asg] pull-agree)
  next
    case 3 thus ?case
      by (simp add: Ee'-def M.ev-app[OF - - pull-asg]
          Ap'I[OF EeD EeD])
  next
    case (4 τ B ξ' ξ'')
    have pull ξ' n σ = pull ξ'' n σ if (n, σ) ∈ occ B for n σ
      using 4(4)[OF that] by (auto simp: pull-def)
    thus ?case unfolding Ee'-def
      by (intro arg-cong[of - - i] M.ev-coin[OF 4(1) pull-asg pull-asg])
  next
    case 5 thus ?case
      by (simp add: Ee'-def M.ev-beta[OF - pull-asg])
  next
    case (6 ξ' a')
    then obtain a where a: Dm ∘ a a' = i a by (auto simp: Dm'-def)
    have vl' (Ap' (Ee' ξ' Neg) a') = vl (Ap (Ee (pull ξ') Neg) a)
      by (simp add: a Ee'-def Ap'I[OF EeD[OF wff-Neg] DmD[OF a(1)]]
          vl'I[OF ApD[OF M.ev-type[OF wff-Neg pull-asg] a(1)]])
    thus ?case
      by (simp add: a M.vl-neg[OF pull-asg a(1)] vl'I[OF DmD[OF a(1)]])
  next
    case (7 ξ' a' b')
```

```
  then obtain a b where ab: Dm o a a' = i a Dm o b b' = i b
    by (auto simp: Dm'-def)
  have dsD: Ap (Ee (pull ξ') Dis) a ∈ D
    by (rule ApD[OF M.ev-type[OF wff-Dis pull-asg] ab(1)])
  have ds2: Dm (o ⇒ o) (Ap (Ee (pull ξ') Dis) a)
    by (rule M.as-appTy[OF M.ev-type[OF wff-Dis pull-asg] ab(1)])
  have vl' (Ap' (Ap' (Ee' ξ' Dis) a') b')
        = vl (Ap (Ap (Ee (pull ξ') Dis) a) b)
    by (simp add: ab Ee'-def Ap'I[OF EeD[OF wff-Dis] DmD[OF ab(1)]]
        Ap'I[OF dsD DmD[OF ab(3)]] vl'I[OF ApD[OF ds2 ab(3)]])
  thus ?case
    by (simp add: ab M.vl-dis[OF pull-asg ab(1) ab(3)]
        vl'I[OF DmD[OF ab(1)]] vl'I[OF DmD[OF ab(3)]])
next
  case (8 ξ' σ f')
  then obtain f where f: Dm (σ ⇒ o) f f' = i f
    by (auto simp: Dm'-def)
  have piD: Dm ((σ ⇒ o) ⇒ o) (Ee (pull ξ') (Pi σ))
    by (rule M.ev-type[OF wff-Pi pull-asg])
  have l: vl' (Ap' (Ee' ξ' (Pi σ)) f')
          = vl (Ap (Ee (pull ξ') (Pi σ)) f)
    by (simp add: f Ee'-def Ap'I[OF EeD[OF wff-Pi] DmD[OF f(1)]]
        vl'I[OF ApD[OF piD f(1)]])
  have r: (∀ d'. Dm' σ d' ⟶ vl' (Ap' f' d'))
          = (∀ d. Dm σ d ⟶ vl (Ap f d))
    by (auto simp: Dm'-def f Ap'I[OF DmD[OF f(1)] DmD]
        vl'I[OF ApD[OF f(1)]])
  show ?case using l r M.vl-pi[OF pull-asg f(1)] by simp
next
  case (9 ξ' σ a' b')
  then obtain a b where ab: Dm σ a a' = i a Dm σ b b' = i b
    by (auto simp: Dm'-def)
  have eqD: Ap (Ee (pull ξ') (Eq σ)) a ∈ D
    by (rule ApD[OF M.ev-type[OF wff-Eq pull-asg] ab(1)])
  have eq2: Dm (σ ⇒ o) (Ap (Ee (pull ξ') (Eq σ)) a)
    by (rule M.as-appTy[OF M.ev-type[OF wff-Eq pull-asg] ab(1)])
  have vl' (Ap' (Ap' (Ee' ξ' (Eq σ)) a') b')
        = vl (Ap (Ap (Ee (pull ξ') (Eq σ)) a) b)
    by (simp add: ab Ee'-def Ap'I[OF EeD[OF wff-Eq] DmD[OF ab(1)]]
        Ap'I[OF eqD DmD[OF ab(3)]] vl'I[OF ApD[OF eq2 ab(3)]])
  thus ?case
    using M.vl-eq[OF pull-asg ab(1) ab(3)] iinj[OF DmD DmD] ab
    by (auto simp: DmD)
next
  case (10 ξ' σ f' a')
  then obtain f a where fa: Dm (σ ⇒ o) f f' = i f Dm σ a a' = i a
    by (auto simp: Dm'-def)
  have sing: vl (Ap f b) ⟷ b = a if b: Dm σ b for b
  proof −
```

```
    have vl' (Ap' f' (i b)) ⟷ i b = a'
      using 10(4)[OF Dm'I[OF b]] by simp
    thus ?thesis
      using b fa iinj[OF DmD DmD]
      by (auto simp: Ap'I[OF DmD[OF fa(1)] DmD[OF b]]
          vl'I[OF ApD[OF fa(1) b]] DmD)
  qed
  have Ap (Ee (pull ξ') (Iota σ)) f = a
    by (rule M.vl-iota[OF pull-asg fa(1) fa(3)]) (rule sing)
  thus ?case
    by (simp add: fa Ee'-def Ap'I[OF EeD[OF wff-Iota] DmD[OF fa(1)]])
next
  case 11
  show ?case
    unfolding A'.functional-def
  proof (intro allI impI)
    fix α β f' g'
    assume f': Dm' (α ⇒ β) f' and g': Dm' (α ⇒ β) g'
      and agree: ∀ a'. Dm' α a' ⟶ Ap' f' a' = Ap' g' a'
    obtain f g where fg: Dm (α ⇒ β) f f' = i f Dm (α ⇒ β) g g' = i g
      using f' g' by (auto simp: Dm'-def)
    have Ap f a = Ap g a if a: Dm α a for a
    proof −
      have i (Ap f a) = i (Ap g a)
        using agree[THEN spec, of i a] Dm'I[OF a]
        by (simp add: fg Ap'I[OF DmD[OF fg(1)] DmD[OF a]]
            Ap'I[OF DmD[OF fg(3)] DmD[OF a]])
      thus ?thesis by (rule iinj[OF ApD ApD, OF fg(1) a fg(3) a])
    qed
    hence f = g
      using M.prop-f fg unfolding M.functional-def by blast
    thus f' = g' by (simp add: fg)
  qed
next
  case (12 a' b')
  then obtain a b where ab: Dm ∘ a a' = i a Dm ∘ b b' = i b
    by (auto simp: Dm'-def)
  thus ?case
    using 12(3) M.prop-b[OF ab(1) ab(3)] vl'I[OF DmD[OF ab(1)]]
      vl'I[OF DmD[OF ab(3)]] by simp
qed
— the pushed assignment refutes A in the image model
define ξ' where ξ' ≡ λn τ. i (ξ n τ)
have asg': app-struct.asg Dm' ξ'
  using xi unfolding M.asg-def ξ'-def A'.asg-def
  by (auto intro: Dm'I)
have Ee' ξ' A = i (Ee ξ A) by (simp add: Ee'-def ξ'-def pull-push)
hence ¬ vl' (Ee' ξ' A)
  using nA vl'I[OF DmD[OF M.ev-type[OF wA xi]]] by simp
```

thus *?thesis* **using** *BM asg′ that* **by** *simp*
qed

5.3.3 Completeness at every infinite carrier

The strengthened form of BKK Corollary 7.7: validity over the models of $\mathcal{M}_{\beta f b}$ at *any* infinite value carrier implies derivability. The term carrier plays no special role: it only needs to embed into the given carrier, which its countability guarantees.

theorem *completeness-at-any-carrier*:
fixes *A* :: *'p*::{*countable*,*infinite*} *tm*
assumes *c*: *cwff* ∘ *A*
and *valid*: ⊨(*'u*::*infinite*) *A*
shows ⊢ *A*
proof (*rule ccontr*)
assume *nd*: ¬⊢ *A*
obtain *Dm Ap* **and** *Ee* :: (*nat* ⇒ *ty* ⇒ *'p tm set*) ⇒ *'p tm* ⇒ *'p tm set* **and** *vl ξ*
where *M*: *bkk-model Dm Ap Ee vl* **and** *cnt*: *countable* {*x*. ∃*τ*. *Dm τ x*}
and *xi*: *app-struct.asg Dm ξ* **and** *nA*: ¬ *vl* (*Ee ξ A*)
by (*rule refuting-term-model*[*OF c nd*])
— embed the countable total domain into the infinite carrier *'u*
obtain *f* :: *'p tm set* ⇒ *nat* **where** *f*: *inj-on f* {*x*. ∃*τ*. *Dm τ x*}
using *cnt* **by** (*auto simp*: *countable-def*)
obtain *g* :: *nat* ⇒ *'u* **where** *g*: *inj g*
using *infinite-UNIV infinite-countable-subset* **by** *blast*
have *inj*: *inj-on* (*g* ∘ *f*) {*x*. ∃*τ*. *Dm τ x*}
using *f g* **by** (*rule comp-inj-on*[*OF - inj-on-subset*]) *auto*
obtain *Dm′ Ap′* **and** *Ee′* :: (*nat* ⇒ *ty* ⇒ *'u*) ⇒ *'p tm* ⇒ *'u* **and** *vl′ ξ′*
where *bkk-model Dm′ Ap′ Ee′ vl′ app-struct.asg Dm′ ξ′* ¬ *vl′* (*Ee′ ξ′ A*)
using *bkk-model-embed M inj cwff-wff*[*OF c*] *xi nA* **by** *blast*
thus *False* **using** *valid* **unfolding** *bkk-valid-def rel-truth-def* **by** *blast*
qed

5.3.4 Completeness for open formulas

Completeness does not require closed sentences: assignments interpret the free variables. The bridge is a substitution-value law for *abstract* Σ-evaluations (the abstract form of BKK Lemma 3.20, one variable at a time), proved without any term induction: substitution is expressed through closing, β-conversion and the evaluation conditions. On the syntactic side a free variable is generalised through a fresh parameter by the rule chain $NK(\beta)$–$NK(\Pi I)$–$NK(\Pi E)$–$NK(\beta)$.

lemma *opn-clos-sub*: *opn k v t* = *t* ⟹ *opn k v* (*clos k x σ t*) = *fsub x σ v t*
by (*induction t arbitrary*: *k*) *auto*

lemma *fsub-id*: *fsub x σ* (*x*$^{f}_{\sigma}$) *t* = *t*
by (*induction t*) *auto*

lemma *occ-clos*: (*x*, *σ*) ∉ *occ* (*clos k x σ t*)

by (*induction t arbitrary*: *k*) *auto*

lemma *pars-clos* [*simp*]: *pars* (*clos k x σ t*) = *pars t*
by (*induction t arbitrary*: *k*) *auto*

lemma *finite-occ*: *finite* (*occ t*)
by (*induction t*) *auto*

lemma *occ-fsub-closed*: *occ u* = {} ⟹ *occ* (*fsub x σ u t*) = *occ t* − {(*x*, *σ*)}
by (*induction t*) *auto*

The sharpened β-application law: the fresh-name condition only concerns the typed occurrence (x, σ), not the bare name (a name may occur at several types).

lemma (**in** *sigma-eval*) *ev-abs-app-occ*:
assumes *wb*: *wff*$_{\sigma \Rightarrow \tau}$(**Λ**$_\sigma$ *b*) **and** *xi*: *asg ξ*
and *x*: (*x*, *σ*) ∉ *occ b* **and** *d*: *Dm σ d*
shows *Ap* (*Ee ξ* (**Λ**$_\sigma$ *b*)) *d* = *Ee* (*ξ*(*x*$_\sigma$:= *d*)) (*b*⟨*x*$^f{}_\sigma$⟩)
proof −
have *xi′*: *asg* (*ξ*(*x*$_\sigma$:= *d*)) **by** (*rule asg-upd*[*OF xi d*])
have *Ee* (*ξ*(*x*$_\sigma$:= *d*)) (*b*⟨*x*$^f{}_\sigma$⟩) = *Ee* (*ξ*(*x*$_\sigma$:= *d*)) ((**Λ**$_\sigma$ *b*) · (*x*$^f{}_\sigma$))
by (*rule ev-beta*[*OF beq.sym*[*OF beq.beta*[*OF wb wff-Fre*]] *xi′*])
also have ... = *Ap* (*Ee* (*ξ*(*x*$_\sigma$:= *d*)) (**Λ**$_\sigma$ *b*)) (*Ee* (*ξ*(*x*$_\sigma$:= *d*)) (*x*$^f{}_\sigma$))
by (*rule ev-app*[*OF wb wff-Fre xi′*])
also have ... = *Ap* (*Ee* (*ξ*(*x*$_\sigma$:= *d*)) (**Λ**$_\sigma$ *b*)) *d*
using *ev-var*[*OF xi′*, *of x σ*] **by** (*simp add*: *upd-def*)
also have ... = *Ap* (*Ee ξ* (**Λ**$_\sigma$ *b*)) *d*
using *x* **by** (*intro arg-cong2*[*of - - d d Ap*] *refl*
ev-coin[*OF wb xi′ xi*]) (*auto simp*: *upd-def*)
finally show *?thesis* **..**
qed

The substitution-value law for abstract Σ-evaluations (BKK Lemma 3.20 for a single free variable): substituting *any* well-formed term equals updating the assignment with its value.

lemma (**in** *sigma-eval*) *ev-fsub-one*:
assumes *wA*: *wff*$_\tau$(*A*) **and** *wu*: *wff*$_\sigma$(*u*) **and** *xi*: *asg ξ*
shows *Ee ξ* (*fsub x σ u A*) = *Ee* (*ξ*(*x*$_\sigma$:= *Ee ξ u*)) *A*
proof −
have *opnA*: *opn 0 v A* = *A* **for** *v* **by** (*simp add*: *wff-lc*[*OF wA*])
let *?B* = *clos 0 x σ A*
have *wAbs*: *wff*$_{\sigma \Rightarrow \tau}$(**Λ**$_\sigma$ *?B*)
by (*rule wff-AbsI*) (*simp add*: *opn-clos-sub*[*OF opnA*] *wff-fsub*[*OF wA*
wff-Fre])
have *du*: *Dm σ* (*Ee ξ u*) **by** (*rule ev-type*[*OF wu xi*])
have *Ee ξ* (*fsub x σ u A*) = *Ee ξ* ((**Λ**$_\sigma$ *?B*) · *u*)
unfolding *opn-clos-sub*[*OF opnA*, *symmetric*]
by (*rule ev-beta*[*OF beq.sym*[*OF beq.beta*[*OF wAbs wu*]] *xi*])
also have ... = *Ap* (*Ee ξ* (**Λ**$_\sigma$ *?B*)) (*Ee ξ u*)

```
    by (rule ev-app[OF wAbs wu xi])
  also have ... = Ee (ξ(x_σ := Ee ξ u)) (?B⟨x^f_σ⟩)
    by (rule ev-abs-app-occ[OF wAbs xi occ-clos du])
  also have ?B⟨x^f_σ⟩ = A
    by (simp add: opn-clos-sub[OF opnA] fsub-id)
  finally show ?thesis .
qed
```

Syntactic generalisation: a fresh parameter substituted for a free variable can be quantified away and re-instantiated, recovering the open formula.

```
lemma bprov-generalize-par:
  assumes wA: wff_o(A) and p: p ∉ pars A
    and d: ⊢ fsub x σ (p^p_σ) A
  shows ⊢ A
proof −
  have opnA: opn 0 v A = A for v :: 'a tm
    by (simp add: wff-lc[OF wA])
  let ?B = clos 0 x σ A
  have wAbs: wff_{σ⇒o}(Λ_σ ?B)
    by (rule wff-AbsI)
       (simp add: opn-clos-sub[OF opnA] wff-fsub[OF wA wff-Fre])
  have bq1: (Λ_σ ?B) · (p^p_σ) ≈_o fsub x σ (p^p_σ) A
    using beq.beta[OF wAbs wff-Par] by (simp add: opn-clos-sub[OF opnA])
  have h2: {} ⊢ (Λ_σ ?B) · (p^p_σ)
    by (rule bprov.Beta[OF beq.sym[OF bq1] d])
  have h3: {} ⊢ (Pi σ) · (Λ_σ ?B)
    by (rule bprov.PiI[OF h2 wAbs]) (use p in auto)
  have h4: {} ⊢ (Λ_σ ?B) · (x^f_σ)
    by (rule bprov.PiE[OF h3 wff-Fre])
  have bq2: (Λ_σ ?B) · (x^f_σ) ≈_o A
  proof −
    have (Λ_σ ?B) · (x^f_σ) ≈_o ?B⟨x^f_σ⟩
      by (rule beq.beta[OF wAbs wff-Fre])
    thus ?thesis by (simp add: opn-clos-sub[OF opnA] fsub-id)
  qed
  show ?thesis by (rule bprov.Beta[OF bq2 h4])
qed
```

Completeness for arbitrary (locally closed) well-formed formulas, by induction on the number of typed free occurrences: each occurrence is generalised through a fresh parameter, semantically justified by the substitution-value law.

```
lemma completeness-open-aux:
  fixes A :: 'p::{countable,infinite} tm
  shows card (occ A) ≤ n ⟹ wff_o(A) ⟹ (⊨('p tm set) A) ⟹ ⊢ A
proof (induction n arbitrary: A)
  case (0 A)
  hence occ A = {} by (simp add: finite-occ card-eq-0-iff)
  hence cwff ∘ A using 0(2) by (simp add: cwff-def fvs-eq-fst-occ)
```

```
  thus ?case using derivable-iff-valid 0(3) by blast
next
  case (Suc n A)
  show ?case
  proof (cases occ A = {})
    case True
    hence cwff ∘ A
      using Suc.prems(2) by (simp add: cwff-def fvs-eq-fst-occ)
    thus ?thesis using derivable-iff-valid Suc.prems(3) by blast
  next
    case False
    then obtain x σ where xs: (x, σ) ∈ occ A by auto
    obtain p :: 'p where p: p ∉ pars A
      by (meson ex-new-if-finite finite-pars infinite-UNIV)
    define A1 where A1 = fsub x σ (p^p_σ) A
    have wA1: wff_∘(A1)
      unfolding A1-def by (rule wff-fsub[OF Suc.prems(2) wff-Par])
    have occ1: occ A1 = occ A − {(x, σ)}
      unfolding A1-def by (rule occ-fsub-closed) simp
    have card1: card (occ A1) ≤ n
      using Suc.prems(1) xs finite-occ
      by (simp add: occ1 card-Diff-singleton)
    have valid1: ⊨('p tm set) A1
      unfolding bkk-valid-def rel-truth-def
    proof (intro allI impI)
      fix Dm Ap and Ee :: (nat ⇒ ty ⇒ 'p tm set) ⇒ 'p tm ⇒ 'p tm set
        and vl :: 'p tm set ⇒ bool and ξ
      assume bm: bkk-model Dm Ap Ee vl and xi: app-struct.asg Dm ξ
      interpret M: bkk-model Dm Ap Ee vl by (rule bm)
      have eq: Ee ξ A1 = Ee (ξ(x_σ := Ee ξ (p^p_σ))) A
        unfolding A1-def
        by (rule M.ev-fsub-one[OF Suc.prems(2) wff-Par xi])
      have asg': app-struct.asg Dm (ξ(x_σ := Ee ξ (p^p_σ)))
        by (rule M.asg-upd[OF xi M.ev-type[OF wff-Par xi]])
      have v: ∀ξ. app-struct.asg Dm ξ ⟶ vl (Ee ξ A)
        using Suc.prems(3) bm unfolding bkk-valid-def rel-truth-def by blast
      show vl (Ee ξ A1) using v[rule-format, OF asg']
        by (simp add: eq)
    qed
    have ⊢ A1 by (rule Suc.IH[OF card1 wA1 valid1])
    thus ?thesis
      unfolding A1-def
      by (rule bprov-generalize-par[OF Suc.prems(2) p, of x σ])
  qed
qed

theorem completeness-open:
  fixes A :: 'p::{countable,infinite} tm
  assumes wff_∘(A)
```

```
    and ⊨('p tm set) A
  shows ⊢ A
 using assms(1,2) completeness-open-aux by blast

lemma completeness-open-any-aux:
 fixes A :: 'p::{countable,infinite} tm
 shows card (occ A) ≤ n ⟹ wff_o(A) ⟹ ⊨('u::infinite) A ⟹ ⊢ A
proof (induction n arbitrary: A)
 case (0 A)
 hence occ A = {} by (simp add: finite-occ card-eq-0-iff)
 hence cwff ∘ A using 0(2) by (simp add: cwff-def fvs-eq-fst-occ)
 thus ?case using completeness-at-any-carrier 0(3) by blast
next
 case (Suc n A)
 show ?case
 proof (cases occ A = {})
  case True
  hence cwff ∘ A
   using Suc.prems(2) by (simp add: cwff-def fvs-eq-fst-occ)
  thus ?thesis using completeness-at-any-carrier Suc.prems(3) by blast
 next
  case False
  then obtain x σ where xs: (x, σ) ∈ occ A by auto
  obtain p :: 'p where p: p ∉ pars A
   by (meson ex-new-if-finite finite-pars infinite-UNIV)
  define A1 where A1 = fsub x σ (p^p_σ) A
  have wA1: wff_o(A1)
   unfolding A1-def by (rule wff-fsub[OF Suc.prems(2) wff-Par])
  have occ1: occ A1 = occ A − {(x, σ)}
   unfolding A1-def by (rule occ-fsub-closed) simp
  have card1: card (occ A1) ≤ n
   using Suc.prems(1) xs finite-occ
   by (simp add: occ1 card-Diff-singleton)
  have valid1: ⊨('u) A1
   unfolding bkk-valid-def rel-truth-def
  proof (intro allI impI)
   fix Dm :: ty ⇒ 'u ⇒ bool and Ap
    and Ee :: (nat ⇒ ty ⇒ 'u) ⇒ 'p tm ⇒ 'u and vl ξ
   assume bm: bkk-model Dm Ap Ee vl and xi: app-struct.asg Dm ξ
   interpret M: bkk-model Dm Ap Ee vl by (rule bm)
   have eq: Ee ξ A1 = Ee (ξ(x_σ := Ee ξ (p^p_σ))) A
    unfolding A1-def
    by (rule M.ev-fsub-one[OF Suc.prems(2) wff-Par xi])
   have asg': app-struct.asg Dm (ξ(x_σ := Ee ξ (p^p_σ)))
    by (rule M.asg-upd[OF xi M.ev-type[OF wff-Par xi]])
   have v: ∀ξ. app-struct.asg Dm ξ ⟶ vl (Ee ξ A)
    using Suc.prems(3) bm unfolding bkk-valid-def rel-truth-def by blast
   show vl (Ee ξ A1) using v[rule-format, OF asg']
    by (simp add: eq)
```

```
    qed
    have ⊢ A1 by (rule Suc.IH[OF card1 wA1 valid1])
    thus ?thesis unfolding A1-def
      by (rule bprov-generalize-par[OF Suc.prems(2) p, of x σ])
  qed
qed

theorem completeness-open-at-any-carrier:
  fixes A :: 'p::{countable,infinite} tm
  assumes wff_O(A)
     and ⊨('u::infinite) A
    shows ⊢ A
  using completeness-open-any-aux assms by auto
```

5.3.5 Signature transport

On the semantic side, maps of parameter names need *no* injectivity: any $h :: {}'p \Rightarrow {}'q$ turns a model for $'q$ into a model for $'p$ by evaluating through *prn h* — the value conditions $vl_{\neg}, \ldots, vl_{\iota}$ only inspect *Ee* at the logical constants, which *prn* fixes.

```
lemma prn-occ [simp]: occ (prn f t) = occ t
  by (induction t) auto

lemma bkk-model-reduct:
  fixes Ee :: (nat ⇒ ty ⇒ 'u) ⇒ 'q tm ⇒ 'u and h :: 'p ⇒ 'q
  assumes bkk-model Dm Ap Ee vl
  shows bkk-model Dm Ap (λξ t. Ee ξ (prn h t)) vl
proof −
  interpret bkk-model Dm Ap Ee vl by (rule assms)
  show ?thesis
  proof (unfold-locales, goal-cases)
    case 3 thus ?case using ev-app by fastforce
    next case 4 thus ?case by (metis ev-coin prn-occ wff-prn)
    next case 5 thus ?case using ev-beta by blast
  qed(auto simp: vl-eq vl-pi vl-dis vl-neg vl-iota ev-var ev-type wff-prn prop-f prop-b)
qed

lemma bkk-valid-map:
  fixes h :: 'p ⇒ 'q
  assumes v: ⊨('u) (A :: 'p tm)
    shows ⊨('u) (prn h A :: 'q tm)
  unfolding bkk-valid-def rel-truth-def
  by (metis bkk-model-reduct bkk-valid-def rel-truth-def v)
```

Completeness for every signature with infinitely many parameters: the finitely many parameters of A are relocated into a copy of $\mathbb{N}$ inside $'p$, completeness over $\mathbb{N}$ applies, and *bprov-rename* transports the derivation back. (With only finitely many parameters this route is barred: $NK(\Pi I)$

consumes fresh eigen-parameters, and an injection $\mathbb{N} \Rightarrow 'p$ is exactly what supplies them.)

theorem *completeness-at-any-signature*:
  fixes $A :: 'p$::*infinite tm*
  assumes *wA*: $wff_o(A)$ **and** *v*: $\models('u$::*infinite*$)\ A$
  shows $\vdash A$
proof −
  obtain *g0* :: *nat* ⇒ $'p$ **where** *g0*: *inj g0*
    using *infinite-UNIV infinite-countable-subset* **by** *blast*
  define *B* **where** $B = pars\ A \cup range\ g0$
  have *inj-on* (*inv g0*) (*range g0*) **by** (*rule inj-on-inv-into*) *simp*
  hence *rc*: *countable* (*range g0*) **by** (*rule countableI*)
  have *ri*: *infinite* (*range g0*)
    using *finite-imageD*[*of g0 UNIV*] *g0 infinite-UNIV-nat* **by** *auto*
  have *cB*: *countable B* **and** *iB*: *infinite B*
    unfolding *B-def* **using** *rc ri* **by** (*auto intro*: *countable-finite*)
  define *g* **where** *g* = *from-nat-into B*
  have *g*: *inj g* **and** *cover*: $pars\ A \subseteq range\ g$
    using *bij-betw-from-nat-into*[*OF cB iB*]
    unfolding *g-def bij-betw-def B-def* **by** *auto*
  define *h* :: $'p \Rightarrow nat$ **where** $h = (\lambda p.\ SOME\ n.\ g\ n = p)$
  have *gh*: $g\ (h\ p) = p$ **if** $p \in pars\ A$ **for** *p*
  proof −
    from *cover that* **obtain** *n* **where** *n*: $g\ n = p$ **by** *auto*
    from *someI*[*of* $\lambda n.\ g\ n = p$, *OF n*] **show** *?thesis* **by** (*simp add*: *h-def*)
  qed
  have *wN*: $wff_o(prn\ h\ A)$ **by** (*rule wff-prn*[*OF wA*])
  have *dN*: $\vdash (prn\ h\ A :: nat\ tm)$
    using *bkk-valid-map completeness-open-at-any-carrier v wN* **by** *blast*
  have $\vdash (prn\ g\ (prn\ h\ A) :: 'p\ tm)$
    using *bprov-rename dN g* **by** *fastforce*
  moreover have $prn\ g\ (prn\ h\ A) = A$
    by (*simp add*: *gh prn-cong prn-prn*)
  ultimately show *?thesis* **by** *simp*
qed

6 Consistency

Consistency, in the typical variants: a concrete Σ-standard model over finite domains is exhibited (so the model class $\mathcal{M}_{\beta fb}$ is non-empty), whence by soundness *NK* does not derive ⊥, and no sentence is derivable together with its negation.

6.1 A concrete Σ-standard model over finite domains

The carrier: an individual, the two truth values, and functions as finite graphs. With a one-element domain of individuals every domain D_τ is finite, so the full function spaces of BKK Definition 3.5 are representable by graphs; *en* τ enumerates D_τ without repetition.

```
datatype mval = MI | MB bool | MF (mval × mval) list

fun en :: ty ⇒ mval list where
  en ι = [MI]
| en o = [MB True, MB False]
| en (σ ⇒ τ) = map (λvs. MF (zip (en σ) vs))
                   (List.n-lists (length (en σ)) (en τ))

definition cD :: ty ⇒ mval ⇒ bool where cD τ v ≡ v ∈ set (en τ)

fun cAp :: mval ⇒ mval ⇒ mval where
  cAp (MF G) x = (case map-of G x of Some y ⇒ y | None ⇒ MI)
| cAp v x = MI

definition cLm :: ty ⇒ (mval ⇒ mval) ⇒ mval where
 cLm σ h ≡ MF (zip (en σ) (map h (en σ)))

lemma en-nonempty: en τ ≠ []
proof (induction τ)
  case (Fun σ τ)
  have replicate (length (en σ)) (hd (en τ)) ∈ set (List.n-lists
      (length (en σ)) (en τ))
      using Fun.IH(2) by (auto simp: set-n-lists)
  thus ?case by auto
qed auto

lemma distinct-en: distinct (en τ)
proof (induction τ)
  case (Fun σ τ)
  have inj-on (λvs. MF (zip (en σ) vs)) (set (List.n-lists (length (en σ)) (en τ)))
  proof
    fix vs ws
    assume vs ∈ set (List.n-lists (length (en σ)) (en τ))
       and ws ∈ set (List.n-lists (length (en σ)) (en τ))
    hence vs = map snd (zip (en σ) vs) and
          ws = map snd (zip (en σ) ws)
      by (auto simp: set-n-lists)
    moreover assume MF (zip (en σ) vs) = MF (zip (en σ) ws)
    ultimately show vs = ws by simp
  qed
  thus ?case
    by (simp add: distinct-map distinct-n-lists Fun.IH(2))
qed simp-all
```

lemma *cD-fun*: *cD* (*σ* ⇒ *τ*) *v* ⟷ (∃ *vs*.
length vs = *length* (*en σ*) ∧ (∀ *x* ∈ *set vs*. *cD τ x*) ∧ *v* = *MF* (*zip* (*en σ*) *vs*))
by (*auto simp*: *cD-def set-n-lists subset-iff*)

lemma *cAp-cLm* [*simp*]: *cD σ a* ⟹ *cAp* (*cLm σ h*) *a* = *h a*
by (*simp add*: *cLm-def cD-def map-of-zip-map*)

lemma *cLm-dom*: (⋀*d*. *cD σ d* ⟹ *cD τ* (*h d*)) ⟹ *cD* (*σ* ⇒ *τ*) (*cLm σ h*)
unfolding *cD-fun cLm-def*
by (*rule exI*[*of* - *map h* (*en σ*)]) (*auto simp*: *cD-def*)

lemma *cAp-dom*: *cD* (*σ* ⇒ *τ*) *f* ⟹ *cD σ a* ⟹ *cD τ* (*cAp f a*)
proof −
assume *cD* (*σ* ⇒ *τ*) *f* **and** *a*: *cD σ a*
then obtain *vs* **where** *vs*: *length vs* = *length* (*en σ*) ∀ *x* ∈ *set vs*. *cD τ x*
and *f*: *f* = *MF* (*zip* (*en σ*) *vs*) **unfolding** *cD-fun* **by** *blast*
obtain *b* **where** *map-of* (*zip* (*en σ*) *vs*) *a* = *Some b* **using** *a vs*(*1*)
unfolding *cD-def* **by** (*metis map-of-zip-is-Some*)
moreover from *this* **have** *b* ∈ *set vs*
by (*metis map-of-SomeD set-zip-rightD*)
ultimately show *?thesis* **using** *vs*(*2*) *f* **by** *simp*
qed

lemma *cAp-ext*:
assumes *f*: *cD* (*σ* ⇒ *τ*) *f* **and** *g*: *cD* (*σ* ⇒ *τ*) *g*
and *ag*: ⋀*a*. *cD σ a* ⟹ *cAp f a* = *cAp g a*
shows *f* = *g*
proof −
obtain *vs* **where** *vs*: *length vs* = *length* (*en σ*) **and** *fv*: *f* = *MF* (*zip* (*en σ*) *vs*)
using *f* **unfolding** *cD-fun* **by** *blast*
obtain *ws* **where** *ws*: *length ws* = *length* (*en σ*) **and** *gw*: *g* = *MF* (*zip* (*en σ*)
ws)
using *g* **unfolding** *cD-fun* **by** *blast*
have *vs* ! *i* = *ws* ! *i* **if** *i*: *i* < *length* (*en σ*) **for** *i*
proof −
have *d*: *cD σ* (*en σ* ! *i*) **using** *i* **by** (*simp add*: *cD-def*)
have *iv*: *i* < *length vs* **and** *iw*: *i* < *length ws* **using** *i vs ws*
by *simp-all*
have *cAp f* (*en σ* ! *i*) = *vs* ! *i*
using *map-of-zip-nth*[*OF vs*[*symmetric*] *distinct-en iv*] *fv*
by *simp*
moreover have *cAp g* (*en σ* ! *i*) = *ws* ! *i*
using *map-of-zip-nth*[*OF ws*[*symmetric*] *distinct-en iw*] *gw*
by *simp*
ultimately show *?thesis* **using** *ag*[*OF d*] **by** *simp*
qed
hence *vs* = *ws* **using** *vs ws* **by** (*intro nth-equalityI*) *auto*
thus *?thesis* **using** *fv gw* **by** *simp*

qed

The denotations of the logical constants, and a canonical parameter and assignment interpretation.

definition *cNg* :: *mval* **where** *cNg* ≡ *cLm* o (λ*a*. *MB* (*a* = *MB False*))
definition *cDs* :: *mval* **where**
cDs ≡ *cLm* o (λ*a*. *cLm* o (λ*b*. *MB* (*a* = *MB True* ∨ *b* = *MB True*)))
definition *cPi* :: *ty* ⇒ *mval* **where**
cPi σ ≡ *cLm* (σ ⇒ o) (λ*f*. *MB* (∀ *d*. *cD* σ *d* ⟶ *cAp f d* = *MB True*))
definition *cEv* :: *ty* ⇒ *mval* **where**
cEv σ ≡ *cLm* σ (λ*a*. *cLm* σ (λ*b*. *MB* (*a* = *b*)))
definition *cIv* :: *ty* ⇒ *mval* **where**
cIv σ ≡ *cLm* (σ ⇒ o)
(λ*f*. *if* ∃ *a*. *cD* σ *a* ∧ (∀ *b*. *cD* σ *b* ⟶ (*cAp f b* = *MB True*) = (*b* = *a*))
then SOME a. *cD* σ *a* ∧ (∀ *b*. *cD* σ *b* ⟶ (*cAp f b* = *MB True*) = (*b* = *a*))
else hd (*en* σ))

abbreviation *cJv* :: *'p* ⇒ *ty* ⇒ *mval* **where** *cJv p* σ ≡ *hd* (*en* σ)
abbreviation *cXi* :: *nat* ⇒ *ty* ⇒ *mval* **where** *cXi n* σ ≡ *hd* (*en* σ)

lemma *cD-bool*: *cD* o *v* ⟷ *v* = *MB True* ∨ *v* = *MB False*
by (*simp add*: *cD-def*)

lemma *hd-en-dom*: *cD* σ (*hd* (*en* σ)) **by** (*simp add*: *cD-def en-nonempty*)

lemma *cIv-desc*:
assumes *f*: *cD* (σ ⇒ o) *f* **and** *a*: *cD* σ *a*
and *sing*: ∀ *b*. *cD* σ *b* ⟶ (*cAp f b* = *MB True*) = (*b* = *a*)
shows *cAp* (*cIv* σ) *f* = *a*
proof −
let *?P* = λ*x*. *cD* σ *x* ∧ (∀ *b*. *cD* σ *b* ⟶ (*cAp f b* = *MB True*) = (*b* = *x*))
have *ex*: ∃ *x*. *?P x* **using** *a sing* **by** *blast*
have *cAp* (*cIv* σ) *f* = (*SOME x*. *?P x*)
unfolding *cIv-def* **using** *cAp-cLm*[*OF f*] *ex* **by** *simp*
moreover have (*SOME x*. *?P x*) = *a*
proof (*rule someI2-ex*[*OF ex*])
fix *x* **assume** *x*: *?P x*
have *cAp f a* = *MB True* **using** *a sing* **by** *blast*
thus *x* = *a* **using** *x a* **by** *blast*
qed
ultimately show *?thesis* **by** *simp*
qed

theorem *concrete-standard-model*:
standard-model cD cAp cLm (*MB True*) (*MB False*) *cNg cDs cIv cEv cPi* (*cJv* :: *'p* ⇒ *ty* ⇒ *mval*)
proof
show ‹*cD* ((σ ⇒ o) ⇒ σ) (*cIv* σ)› **for** σ
unfolding *cIv-def*

by (*rule cLm-dom*) (*auto simp*: *hd-en-dom intro*: *someI2-ex*)
next
fix *a b*
assume *13*: ‹*cD* ∘ *a*› ‹*cD* ∘ *b*›
have *inner*: *cAp cDs d* = *cLm* ∘ (λ*b*. *MB* (*d* = *MB True* ∨ *b* = *MB True*))
if *cD* ∘ *d* **for** *d* **unfolding** *cDs-def* **by** (*rule cAp-cLm*[*OF that*])
have *ds*: *cAp* (*cAp cDs d*) *e* = *MB* (*d* = *MB True* ∨ *e* = *MB True*)
if *d*: *cD* ∘ *d* **and** *e*: *cD* ∘ *e* **for** *d e* **unfolding** *inner*[*OF d*]
by (*rule cAp-cLm*[*OF e*])
show ‹(*cAp* (*cAp cDs a*) *b* = *MB True*) = (*a* = *MB True* ∨ *b* = *MB True*)›
unfolding *ds*[*OF 13*(*1*) *13*(*2*)] **by** *simp*
qed(*auto simp*: *cAp-ext cDs-def cIv-desc cEv-def cPi-def cNg-def cD-bool hd-en-dom*
intro!: *cLm-dom cAp-dom*)

6.2 Consistency of *NK*

The Σ-model predicate of the canonical construction, exported from the sublocale chain *standard-model* ⊆ *general-model* ⊆ *bkk-model*.

lemma (**in** *general-model*) *bkk-model-pred*:
bkk-model Dm Ap (λξ *A*. (|*A*|)$_\xi$) (λ*a*. *a* = *Tv*) **by** *intro-locales*

Consistency of *NK* (the deep embedding): ⊥ is not derivable — by soundness (BKK Theorem 7.3) it would have to be v-true in the concrete model, contradicting BKK Lemma 3.43.

theorem *nk-consistent*: ¬ (⊢ (⊥ :: *'p tm*))
proof
assume *d*: ⊢ (⊥ :: *'p tm*)
interpret *standard-model cD cAp cLm MB True MB False cNg cDs cIv cEv cPi*
cJv :: *'p* ⇒ *ty* ⇒ *mval*
by (*rule concrete-standard-model*)
have *xi*: *bkkA.asg cXi* **by** (*simp add*: *bkkA.asg-def hd-en-dom*)
have *den* (⊥ :: *'p tm*) *cXi* = *MB True*
using *soundness-bkk*[*OF d bkk-model-pred xi*] **by** *simp*
thus *False* **using** *bkk.vl-TF*[*OF xi*] **by** *simp*
qed

No sentence is derivable together with its negation.

theorem *nk-not-both*: **assumes** ⊢ (*A* :: *'p tm*) **shows** ¬⊢ ¬ *A*
using *bprov.NegE*[*OF - assms wff-FalseB*] *nk-consistent* **by** *blast*

In the terminology of the completeness development: the empty set is consistent (BKK Definition 7.4).

corollary *con-empty*: *con* ({} :: *'p tm set*)
unfolding *con-def* **by** (*rule nk-consistent*)

7 Example: Cantor's theorem

The surjective and the injective Cantor theorem, at every type σ: there is no surjection from σ onto $\sigma \Rightarrow \mathrm{o}$, and no injection from $\sigma \Rightarrow \mathrm{o}$ into σ. Both are derived *inside the calculus*: genuine *NK*-derivations via the diagonal predicate (for the injective version via the description operator $NK(\iota)$, following Andrews 1972). The Cantor sentences are stated as in the *Stanford Encyclopedia of Philosophy* entry on Church's type theory (Benzmüller and Andrews 2024), generalised from ι to every type σ.

7.1 The defined existential quantifier, and the Cantor sentences

The Cantor sentences are stated literally, as in the cited encyclopedia entry: no surjection $\mathcal{G} : \sigma \Rightarrow \sigma \Rightarrow \mathrm{o}$ onto $\sigma \Rightarrow \mathrm{o}$, and no injection $\mathcal{I} : (\sigma \Rightarrow \mathrm{o}) \Rightarrow \sigma$. The following lemmas record their locally-nameless normal forms (by computation) and the closedness facts used by the derivations. The right-hand sides deliberately show the machine-level de Bruijn normal form (indices *Bnd 0*, *Bnd* (*Suc 0*) and so on); the named-binder left-hand sides are the human-facing statements.

lemma *surj-norm*:
$(\exists\, \mathcal{G}_\sigma \Rightarrow \sigma \Rightarrow \mathrm{o}.\ \mathbf{\Pi}\mathcal{F}_\sigma \Rightarrow \mathrm{o}.\ \exists\, \mathcal{X}_\sigma.\ ((\mathcal{G}^f\sigma \Rightarrow \sigma \Rightarrow \mathrm{o} \cdot \mathcal{X}^f\sigma) =_\sigma \Rightarrow \mathrm{o}\ \mathcal{F}^f\sigma \Rightarrow \mathrm{o}))$
$= \exists_\sigma \Rightarrow \sigma \Rightarrow \mathrm{o}\ (\mathbf{\Pi}_\sigma \Rightarrow \mathrm{o}\ (\exists_\sigma$
$((Bnd\ (Suc\ (Suc\ 0)) \cdot Bnd\ 0) =_\sigma \Rightarrow \mathrm{o}\ Bnd\ (Suc\ 0))))$
by (*simp add*: *ExN-def AllN-def* $\mathcal{G}$*-def* $\mathcal{F}$*-def* $\mathcal{X}$*-def*)

lemma *inj-norm*:
$(\exists\, \mathcal{I}_{(\sigma \Rightarrow \mathrm{o}) \Rightarrow \sigma}.\ \mathbf{\Pi}\mathcal{F}_\sigma \Rightarrow \mathrm{o}.\ \mathbf{\Pi}\mathcal{H}_\sigma \Rightarrow \mathrm{o}.$
$(((\mathcal{I}^f{}_{(\sigma \Rightarrow \mathrm{o}) \Rightarrow \sigma} \cdot \mathcal{F}^f\sigma \Rightarrow \mathrm{o}) =_\sigma (\mathcal{I}^f{}_{(\sigma \Rightarrow \mathrm{o}) \Rightarrow \sigma} \cdot \mathcal{H}^f\sigma \Rightarrow \mathrm{o}))$
$\supset (\mathcal{F}^f\sigma \Rightarrow \mathrm{o} =_\sigma \Rightarrow \mathrm{o}\ \mathcal{H}^f\sigma \Rightarrow \mathrm{o})))$
$= \exists_{(\sigma \Rightarrow \mathrm{o}) \Rightarrow \sigma}\ (\mathbf{\Pi}_\sigma \Rightarrow \mathrm{o}\ (\mathbf{\Pi}_\sigma \Rightarrow \mathrm{o}$
$(((Bnd\ (Suc\ (Suc\ 0)) \cdot Bnd\ (Suc\ 0)) =_\sigma (Bnd\ (Suc\ (Suc\ 0)) \cdot Bnd$
$0))$
$\supset (Bnd\ (Suc\ 0) =_\sigma \Rightarrow \mathrm{o}\ Bnd\ 0))))$
by (*simp add*: *ExN-def AllN-def* $\mathcal{I}$*-def* $\mathcal{F}$*-def* $\mathcal{H}$*-def*)

lemma *wff-surj*:
$wff_{\mathrm{o}}(\exists\, \mathcal{G}_\sigma \Rightarrow \sigma \Rightarrow \mathrm{o}.\ \mathbf{\Pi}\mathcal{F}_\sigma \Rightarrow \mathrm{o}.\ \exists\, \mathcal{X}_\sigma.$
$((\mathcal{G}^f\sigma \Rightarrow \sigma \Rightarrow \mathrm{o} \cdot \mathcal{X}^f\sigma) =_\sigma \Rightarrow \mathrm{o}\ \mathcal{F}^f\sigma \Rightarrow \mathrm{o}))$
unfolding *surj-norm*
by (*intro wff-Not wff-Forall*)
(*auto del*: *wff-Not wff-PEq wff-Forall*
intro!: *wff-Not wff-Forall wff-PEq wff-Eq wff-App wff-Fre*)

lemma *wff-inj*:
$wff_{\mathrm{o}}(\exists\, \mathcal{I}_{(\sigma \Rightarrow \mathrm{o}) \Rightarrow \sigma}.\ \mathbf{\Pi}\mathcal{F}_\sigma \Rightarrow \mathrm{o}.\ \mathbf{\Pi}\mathcal{H}_\sigma \Rightarrow \mathrm{o}.$

$$(((\mathcal{I}^f{}_{(\sigma \Rightarrow o) \Rightarrow \sigma} \cdot \mathcal{F}^f\sigma \Rightarrow o) =_\sigma (\mathcal{I}^f{}_{(\sigma \Rightarrow o) \Rightarrow \sigma} \cdot \mathcal{H}^f\sigma \Rightarrow o)) \supset (\mathcal{F}^f\sigma \Rightarrow o =_\sigma \Rightarrow o\ \mathcal{H}^f\sigma \Rightarrow o)))$$

unfolding *inj-norm*
by (*intro wff-Not wff-Forall*)
(*auto del*: *wff-Not wff-Forall wff-ImpB wff-PEq*
intro!: *wff-Not wff-Forall wff-ImpB wff-PEq wff-Eq wff-App wff-Fre*)

lemma *fvs-surj* [*simp*]:
$$fvs\ (\neg\ (\exists\,\mathcal{G}\sigma \Rightarrow \sigma \Rightarrow o.\ \mathbf{\Pi}\mathcal{F}\sigma \Rightarrow o.\ \exists\,\mathcal{X}_\sigma.\ ((\mathcal{G}^f\sigma \Rightarrow \sigma \Rightarrow o \cdot \mathcal{X}^f{}_\sigma) =_\sigma \Rightarrow o\ \mathcal{F}^f\sigma \Rightarrow o))) = \{\}$$
by (*simp add*: *surj-norm*)

lemma *fvs-inj* [*simp*]:
$$fvs\ (\neg\ (\exists\,\mathcal{I}_{(\sigma \Rightarrow o) \Rightarrow \sigma}.\ \mathbf{\Pi}\mathcal{F}\sigma \Rightarrow o.\ \mathbf{\Pi}\mathcal{H}\sigma \Rightarrow o.\ (((\mathcal{I}^f{}_{(\sigma \Rightarrow o) \Rightarrow \sigma} \cdot \mathcal{F}^f\sigma \Rightarrow o) =_\sigma (\mathcal{I}^f{}_{(\sigma \Rightarrow o) \Rightarrow \sigma} \cdot \mathcal{H}^f\sigma \Rightarrow o)) \supset (\mathcal{F}^f\sigma \Rightarrow o =_\sigma \Rightarrow o\ \mathcal{H}^f\sigma \Rightarrow o)))) = \{\}$$
by (*simp add*: *inj-norm*)

lemma *pars-surj* [*simp*]:
$$pars\ (\exists\,\mathcal{G}\sigma \Rightarrow \sigma \Rightarrow o.\ \mathbf{\Pi}\mathcal{F}\sigma \Rightarrow o.\ \exists\,\mathcal{X}_\sigma.\ ((\mathcal{G}^f\sigma \Rightarrow \sigma \Rightarrow o \cdot \mathcal{X}^f{}_\sigma) =_\sigma \Rightarrow o\ \mathcal{F}^f\sigma \Rightarrow o)) = \{\}$$
by (*simp add*: *surj-norm Forall-def*)

lemma *pars-inj* [*simp*]:
$$pars\ (\exists\,\mathcal{I}_{(\sigma \Rightarrow o) \Rightarrow \sigma}.\ \mathbf{\Pi}\mathcal{F}\sigma \Rightarrow o.\ \mathbf{\Pi}\mathcal{H}\sigma \Rightarrow o.\ (((\mathcal{I}^f{}_{(\sigma \Rightarrow o) \Rightarrow \sigma} \cdot \mathcal{F}^f\sigma \Rightarrow o) =_\sigma (\mathcal{I}^f{}_{(\sigma \Rightarrow o) \Rightarrow \sigma} \cdot \mathcal{H}^f\sigma \Rightarrow o)) \supset (\mathcal{F}^f\sigma \Rightarrow o =_\sigma \Rightarrow o\ \mathcal{H}^f\sigma \Rightarrow o))) = \{\}$$
by (*simp add*: *inj-norm*)

Typing of the terms occurring in the derivations, once and for all.

lemma *wff-surj-body*:
$$wff_{(\sigma \Rightarrow \sigma \Rightarrow o) \Rightarrow o}(\mathbf{\Lambda}\sigma \Rightarrow \sigma \Rightarrow o\ (\mathbf{\Pi}\sigma \Rightarrow o\ (\exists_\sigma\ ((Bnd\ (Suc\ (Suc\ 0)) \cdot Bnd\ 0) =_\sigma \Rightarrow o\ Bnd\ (Suc\ 0)))))$$
by (*rule wff-AbsI*)
(*auto del*: *wff-Not wff-Forall wff-PEq*
intro!: *wff-Not wff-Forall wff-PEq wff-Eq wff-App wff-Fre*)

lemma *wff-inst-body*:
$$wff_{(\sigma \Rightarrow o) \Rightarrow o}(\mathbf{\Lambda}\sigma \Rightarrow o\ (\exists_\sigma\ ((Par\ g\ (\sigma \Rightarrow \sigma \Rightarrow o) \cdot Bnd\ 0) =_\sigma \Rightarrow o\ Bnd\ (Suc\ 0))))$$
by (*rule wff-AbsI*)
(*auto del*: *wff-Not wff-Forall wff-PEq*
intro!: *wff-Not wff-Forall wff-PEq wff-Eq wff-App wff-Par wff-Fre*)

lemma *wff-diag*:
$$wff_{\sigma \Rightarrow o}(\mathbf{\Lambda}_\sigma\ (\neg\ ((Par\ g\ (\sigma \Rightarrow \sigma \Rightarrow o) \cdot Bnd\ 0) \cdot Bnd\ 0)))$$
by (*rule wff-AbsI*) (*auto del*: *wff-Not intro*!: *wff-Not wff-App wff-Par wff-Fre*)

lemma *wff-wit-body*:
assumes $wff_\sigma \Rightarrow {}_{\mathrm{o}}(D)$ **and** *lc D*
shows $wff_\sigma \Rightarrow {}_{\mathrm{o}}(\mathbf{\Lambda}_\sigma\ ((Par\ g\ (\sigma \Rightarrow \sigma \Rightarrow {}_{\mathrm{o}})) \cdot Bnd\ 0) =_\sigma \Rightarrow {}_{\mathrm{o}}\ D))$
using *assms*
by (*intro wff-AbsI*) (*auto del*: *wff-PEq intro*!: *wff-PEq wff-Eq wff-App wff-Par wff-Fre*)

7.2 The surjective Cantor theorem in *NK*

theorem *nk-surjective-cantor*: **fixes** σ :: *ty* **shows**
$\vdash (\neg\ (\exists\, \mathcal{G}_\sigma \Rightarrow \sigma \Rightarrow {}_{\mathrm{o}}.\ \mathbf{\Pi}\mathcal{F}_\sigma \Rightarrow {}_{\mathrm{o}}.\ \exists\, \mathcal{X}_\sigma.$
$((\mathcal{G}^f{}_\sigma \Rightarrow \sigma \Rightarrow {}_{\mathrm{o}} \cdot \mathcal{X}^f{}_\sigma) =_\sigma \Rightarrow {}_{\mathrm{o}}\ \mathcal{F}^f{}_\sigma \Rightarrow {}_{\mathrm{o}})) :: {'p}{::}infinite\ tm)$
(**is** $\vdash \neg\ ?S$)
proof −
— two distinct parameters: the assumed surjection *g*, the inner witness *a*
obtain *g a* :: $'p$ **where** *ag*: $a \neq g$
by (*metis* (*full-types*) *ex-new-if-finite finite.emptyI*
finite.insertI infinite-UNIV insert-iff)
let $?gP = g^p{}_\sigma \Rightarrow \sigma \Rightarrow {}_{\mathrm{o}} :: {'p}\ tm$ **and** $?aP = a^p{}_\sigma :: {'p}\ tm$
— the diagonal $?F = \mathbf{\Lambda}\mathcal{X}.\ \neg\ (g \cdot \mathcal{X} \cdot \mathcal{X})$; the two $\exists$-assumptions; the diagonal instance $?\varphi = g \cdot a \cdot a$
let $?F = \mathbf{\Lambda}_\sigma\ (\neg\ ((?gP \cdot Bnd\ 0) \cdot Bnd\ 0))$
let $?Sg = \mathbf{\Pi}_\sigma \Rightarrow {}_{\mathrm{o}}\ (\exists\, {}_\sigma\ ((?gP \cdot Bnd\ 0) =_\sigma \Rightarrow {}_{\mathrm{o}}\ Bnd\ (Suc\ 0)))$
let $?E = (?gP \cdot ?aP) =_\sigma \Rightarrow {}_{\mathrm{o}}\ ?F$
let $?\varphi = (?gP \cdot ?aP) \cdot ?aP$
— bookkeeping, once and for all: typing and parameter-freshness
have *wF*: $wff_\sigma \Rightarrow {}_{\mathrm{o}}(?F)$ **by** (*rule wff-diag*)
have *lcF*: *lc ?F* **by** (*rule wff-lc*[*OF wF*])
have *fp1*: *freep* {*?S*, *?Sg*} **and** *fp2*: *freep* {*?S*, *?Sg*, *?E*}
by (*intro freep-finite*, *simp*)+
— the derivation, innermost context first
have *1*: {*?S*, *?Sg*, *?E*} ⊢ *?E* — *NK*(*Hyp*)
by (*auto intro*: *bprov.Hyp*)
have *2*: {*?S*, *?Sg*, *?E*} ⊢ $?\varphi \doteq_{\mathrm{o}} (?F \cdot ?aP)$
— apply both sides of (1) to *a* — $NK(=_l)$, Leibniz $NK(\Pi E)$
by (*rule peq-app*[*OF 1 fp2*]) (*auto intro*!: *wff-App wff-Par wF*)
have *3*: $?F \cdot ?aP \approx_{\mathrm{o}} \neg\ ?\varphi$ — β-reduce the diagonal
using *beq.beta*[*OF wF wff-Par*] **by** *simp*
have *4*: {*?S*, *?Sg*, *?E*} ⊢ $?\varphi \doteq_{\mathrm{o}} (\neg\ ?\varphi)$ — $NK(\beta)$ on (2) by (3)
by (*rule leib-reduce-right*[*OF 2 3*]) (*auto intro*!: *wff-App wff-Par*)
have *5*: {*?S*, *?Sg*, *?E*} ⊢ ⊥
— $?\varphi \doteq \neg\ ?\varphi$ is contradictory — $NK(\Pi E)$, $NK(\neg E)$, tertium non datur
by (*rule leib-neg-contra*[*OF 4 fp2*]) (*auto intro*!: *wff-App wff-Par*)
have *6*: {*?S*, *?Sg*} ⊢ *?Sg* — *NK*(*Hyp*)
by (*auto intro*: *bprov.Hyp*)
have *7*: {*?S*, *?Sg*} ⊢ $\exists\, {}_\sigma\ ((?gP \cdot Bnd\ 0) =_\sigma \Rightarrow {}_{\mathrm{o}}\ ?F)$
— $NK(\Pi E)$: instantiate the surjectivity of *g* at the diagonal *?F*
using *PiE-open*[*OF 6 wff-inst-body wF*]
by (*simp add*: *opn-lc*[*OF lcF*])

have *8*: $\{?S,\ ?Sg\} \vdash \bot$ — $NK(\exists\, E)$, discharging the witness a
by (*rule ExE*[**where** $w = a$, *OF 7 -*
wff-wit-body[*OF wF lcF*] *wff-FalseB*])
(*use 5 ag* **in** ‹*auto simp*: *opn-lc*[*OF lcF*] *insert-commute*
intro!: *freep-finite*›)
have *9*: $\{?S\} \vdash \exists_\sigma \Rightarrow \sigma \Rightarrow \mathrm{o}\ (\mathbf{\Pi}_\sigma \Rightarrow \mathrm{o}\ (\exists_\sigma$
$((Bnd\ (Suc\ (Suc\ 0)) \cdot Bnd\ 0) =_\sigma \Rightarrow \mathrm{o}\ Bnd\ (Suc\ 0))))$
— $NK(Hyp)$, in locally-nameless normal form
by (*subst surj-norm*[*symmetric*]) (*auto intro*: *bprov.Hyp*)
have *10*: $\{?S\} \vdash \bot$ — $NK(\exists\, E)$, discharging the witness g
by (*rule ExE*[**where** $w = g$, *OF 9 - wff-surj-body wff-FalseB*])
(*use 8* **in** ‹*auto simp*: *insert-commute intro*!: *freep-finite*›)
show *?thesis* — $NK(\neg I)$ discharges the assumed surjection
by (*rule bprov.NegI*[*OF - wff-surj*]) (*use 10* **in** *simp*)
qed

7.3 The injective Cantor theorem in *NK*

Typing of the description-based diagonal predicate.

lemma *wff-desc-diag*:
$wff_\sigma \Rightarrow \mathrm{o}(\mathbf{\Lambda}_\sigma\ (\neg\ (((Iota\ (\sigma \Rightarrow \mathrm{o})) \cdot (\mathbf{\Lambda}_\sigma \Rightarrow \mathrm{o}\ (((Par\ i\ ((\sigma \Rightarrow \mathrm{o}) \Rightarrow \sigma)) \cdot$
$Bnd\ 0) \doteq_\sigma Bnd\ (Suc\ 0)))) \cdot Bnd\ 0)))$
proof (*rule wff-AbsI*)
fix x
have *inner*: $wff_{(\sigma \Rightarrow \mathrm{o}) \Rightarrow \mathrm{o}}(\mathbf{\Lambda}_\sigma \Rightarrow \mathrm{o}\ (((Par\ i\ ((\sigma \Rightarrow \mathrm{o}) \Rightarrow \sigma)) \cdot Bnd\ 0)$
$\doteq_\sigma x^f{}_\sigma))$
by (*rule wff-AbsI*)
(*auto del*: *wff-PEq wff-LeibE wff-Leib*
intro!: *wff-LeibE wff-Leib wff-PEq wff-Eq wff-App wff-Par wff-Fre*)
show $wff_{\mathrm{o}}((\neg\ (((Iota\ (\sigma \Rightarrow \mathrm{o})) \cdot$
$(\mathbf{\Lambda}_\sigma \Rightarrow \mathrm{o}\ (((Par\ i\ ((\sigma \Rightarrow \mathrm{o}) \Rightarrow \sigma)) \cdot Bnd\ 0) \doteq_\sigma Bnd\ (Suc\ 0)))) \cdot Bnd$
$0))\langle x^f{}_\sigma\rangle)$
using *inner* **by** (*auto del*: *wff-Not intro*!: *wff-Not wff-App wff-Iota wff-Fre*)
qed

theorem *nk-injective-cantor*: **fixes** σ :: *ty* **shows**
$\vdash (\neg\ (\exists\, \mathcal{I}_{(\sigma \Rightarrow \mathrm{o}) \Rightarrow \sigma}.\ \mathbf{\Pi}\mathcal{F}_\sigma \Rightarrow \mathrm{o}.\ \mathbf{\Pi}\mathcal{H}_\sigma \Rightarrow \mathrm{o}.$
$(((\mathcal{I}^f{}_{(\sigma \Rightarrow \mathrm{o}) \Rightarrow \sigma} \cdot \mathcal{F}^f{}_\sigma \Rightarrow \mathrm{o}) =_\sigma (\mathcal{I}^f{}_{(\sigma \Rightarrow \mathrm{o}) \Rightarrow \sigma} \cdot \mathcal{H}^f{}_\sigma \Rightarrow \mathrm{o}))$
$\supset (\mathcal{F}^f{}_\sigma \Rightarrow \mathrm{o} =_\sigma \Rightarrow \mathrm{o}\ \mathcal{H}^f{}_\sigma \Rightarrow \mathrm{o}))) :: {'p}{::}infinite\ tm)$
(**is** $\vdash \neg\ ?S$)
proof −
— The same narrative as for the surjective theorem, with the diagonal formed through the description operator: steps (1)-(2) instantiate the assumed injectivity; step (3) shows that, by injectivity, the singleton predicate $\mathbf{\Lambda}H.\ i \cdot H \doteq a$ of the image point $a = i \cdot ?G$ is the Leibniz singleton of the diagonal $?G = \mathbf{\Lambda}x.\ \neg\ (\boldsymbol{\iota}(\mathbf{\Lambda}H.$ $i \cdot H \doteq x) \cdot x)$, so $NK(\iota)$ describes it to $?G$ itself (step (4)) — description inverts i; steps (5)-(6) derive the diagonal contradiction $?\psi \doteq \neg\ ?\psi$; steps (7)-(8) and

$NK(\neg I)$ discharge the assumption.
obtain $i\ h :: 'p$ **where** *hi*: $h \neq i$
by (*metis* (*full-types*) *ex-new-if-finite finite.emptyI finite.insertI infinite-UNIV insert-iff*)
let $?\tau = (\sigma \Rightarrow o) \Rightarrow \sigma$
let $?iP = i^p{}_{?\tau} :: 'p\ tm$ **and** $?hP = h^p\sigma \Rightarrow o :: 'p\ tm$
let $?G = \mathbf{\Lambda}_\sigma\ (\neg\ (((Iota\ (\sigma \Rightarrow o)) \cdot (\mathbf{\Lambda}_\sigma \Rightarrow o\ ((?iP \cdot Bnd\ 0) \doteq_\sigma Bnd\ (Suc\ 0)))) \cdot Bnd\ 0))$
let $?a = ?iP \cdot ?G$
let $?PX = \mathbf{\Lambda}\sigma \Rightarrow o\ ((?iP \cdot Bnd\ 0) \doteq_\sigma ?a)$
let $?LG = (Leib\ (\sigma \Rightarrow o)) \cdot ?G$
let $?\psi = ((Iota\ (\sigma \Rightarrow o)) \cdot ?PX) \cdot ?a$
let $?IBb = \mathbf{\Pi}_\sigma \Rightarrow o\ (\mathbf{\Pi}_\sigma \Rightarrow o$
$(((Bnd\ (Suc\ (Suc\ 0)) \cdot Bnd\ (Suc\ 0)) =_\sigma (Bnd\ (Suc\ (Suc\ 0)) \cdot Bnd\ 0))$
$\supset (Bnd\ (Suc\ 0) =_\sigma \Rightarrow o\ Bnd\ 0)))$
let $?IB = \mathbf{\Pi}_\sigma \Rightarrow o\ (\mathbf{\Pi}_\sigma \Rightarrow o$
$(((?iP \cdot Bnd\ (Suc\ 0)) =_\sigma (?iP \cdot Bnd\ 0))$
$\supset (Bnd\ (Suc\ 0) =_\sigma \Rightarrow o\ Bnd\ 0)))$
let $?\Gamma = \{?S,\ ?IB\}$
have *wG*: $wff_\sigma \Rightarrow o(?G)$
using *wff-desc-diag*[**where** $\sigma = \sigma$ **and** $i = i$] **by** *simp*
— bookkeeping, once and for all: typing, local closure, freshness
have *lcG*: *lc ?G* **by** (*rule wff-lc*[*OF wG*])
have *wa*: $wff_\sigma(?a)$ **by** (*auto intro*!: *wff-App wff-Par wG*)
have *lca*: *lc ?a* **by** (*rule wff-lc*[*OF wa*])
have *wPX*: $wff_{(\sigma \Rightarrow o) \Rightarrow o}(?PX)$
by (*rule wff-AbsI*)
(*auto del*: *wff-LeibE simp*: *opn-lc*[*OF lca*]
intro!: *wff-LeibE wff-App wff-Par wff-Fre wa wG*)
have *wLG*: $wff_{(\sigma \Rightarrow o) \Rightarrow o}(?LG)$
by (*auto del*: *wff-Leib intro*!: *wff-App wff-Leib wG*)
have *wIo*: $wff_\sigma \Rightarrow o((Iota\ (\sigma \Rightarrow o)) \cdot ?PX)$
by (*auto intro*!: *wff-App wff-Iota wPX*)
have *w*ψ: $wff_o(?\psi)$ **by** (*rule wff-App*[*OF wIo wa*])
have *wIBb*: $wff_{((\sigma \Rightarrow o) \Rightarrow \sigma) \Rightarrow o}(\mathbf{\Lambda}_{(\sigma \Rightarrow o) \Rightarrow \sigma}\ ?IBb :: 'p\ tm)$
by (*rule wff-AbsI*)
(*auto del*: *wff-Forall wff-ImpB wff-LeibE wff-Leib wff-PEq*
intro!: *wff-Forall wff-ImpB wff-LeibE wff-Leib wff-PEq wff-Eq wff-App wff-Fre*)
have *fp*Γ: *freep* $?\Gamma$ **by** (*intro freep-finite*) *simp*
— the derivation
have *1*: $?\Gamma \vdash ?IB$ — $NK(Hyp)$
by (*auto intro*: *bprov.Hyp*)
have *2*: $?\Gamma \vdash ((?iP \cdot F) =_\sigma (?iP \cdot H)) \supset (F =_\sigma \Rightarrow o\ H)$
if *wF*: $wff_\sigma \Rightarrow o(F)$ **and** *wH*: $wff_\sigma \Rightarrow o(H)$
and *lcF*: *lc F* **and** *lcH*: *lc H* **for** *F H*
— instantiate (1) at closed *F*, *H*: twice $NK(\Pi E)$, each followed by $NK(\beta)$
proof −

let *?B1* = $\mathbf{\Pi}_{\sigma \Rightarrow o}$ (((*?iP* · *Bnd* (*Suc 0*)) $=_\sigma$ (*?iP* · *Bnd 0*))
⊃ (*Bnd* (*Suc 0*) $=_{\sigma \Rightarrow o}$ *Bnd 0*))
have *wB1*: *wff*$_{(\sigma \Rightarrow o) \Rightarrow o}$($\mathbf{\Lambda}_{\sigma \Rightarrow o}$ *?B1*)
by (*rule wff-AbsI*)
(*auto del*: *wff-ImpB intro*!: *wff-ImpB wff-Eq wff-App wff-Par wff-Fre*)
have *bq1*: ($\mathbf{\Lambda}_{\sigma \Rightarrow o}$ *?B1*) · *F* $\approx_o$
$\mathbf{\Pi}_{\sigma \Rightarrow o}$ (((*?iP* · *F*) $=_\sigma$ (*?iP* · *Bnd 0*)) ⊃ (*F* $=_{\sigma \Rightarrow o}$ *Bnd 0*))
using *beq.beta*[*OF wB1 wF*] **by** (*simp add*: *opn-lc*[*OF lcF*])
have *wB2*: *wff*$_{(\sigma \Rightarrow o) \Rightarrow o}$($\mathbf{\Lambda}_{\sigma \Rightarrow o}$
(((*?iP* · *F*) $=_\sigma$ (*?iP* · *Bnd 0*)) ⊃ (*F* $=_{\sigma \Rightarrow o}$ *Bnd 0*)))
by (*rule wff-AbsI*)
(*auto simp*: *opn-lc*[*OF lcF*] *del*: *wff-ImpB*
intro!: *wff-ImpB wff-Eq wff-App wff-Par wff-Fre wF*)
have *bq2*: ($\mathbf{\Lambda}_{\sigma \Rightarrow o}$
(((*?iP* · *F*) $=_\sigma$ (*?iP* · *Bnd 0*)) ⊃ (*F* $=_{\sigma \Rightarrow o}$ *Bnd 0*))) · *H* $\approx_o$
((*?iP* · *F*) $=_\sigma$ (*?iP* · *H*)) ⊃ (*F* $=_{\sigma \Rightarrow o}$ *H*)
using *beq.beta*[*OF wB2 wH*] **by** (*simp add*: *opn-lc*[*OF lcF*]
opn-lc[*OF lcH*])
show *?thesis* **by** (*rule bprov.Beta*[*OF bq2 PiE-Forall*[*OF*
bprov.Beta[*OF bq1 PiE-Forall*[*OF 1 wF*]] *wH*]])
qed
have *3*: *?*Γ ⊢ *?PX* $\doteq_{(\sigma \Rightarrow o) \Rightarrow o}$ *?LG*
— by injectivity, the singleton predicate of *?a* is the Leibniz singleton of *?G*: pointwise by *NK*(*b*), closed by *NK*(Π*I*) and *NK*(*f*)
proof −
let *?body* = (*?PX* · *Bnd 0*) $\doteq_o$ (*?LG* · *Bnd 0*)
have *wbody*: *wff*$_{(\sigma \Rightarrow o) \Rightarrow o}$($\mathbf{\Lambda}_{\sigma \Rightarrow o}$ *?body*)
by (*rule wff-AbsI*) (*auto simp*: *opn-lc*[*OF wff-lc*[*OF wPX*]]
opn-lc[*OF wff-lc*[*OF wLG*]]
intro!: *wff-Eq wff-App wff-Fre*
wPX wLG wG)
have *A-beta*: *?PX* · *?hP* $\approx_o$ ((*?iP* · *?hP*) $\doteq_\sigma$ *?a*) — *NK*(β)
using *beq.beta*[*OF wPX wff-Par*] **by** (*simp add*: *opn-lc*[*OF lca*])
have *wA*: *wff*$_o$(*?PX* · *?hP*) **by** (*auto intro*!: *wff-App wPX wff-Par*)
have *wB*: *wff*$_o$(*?LG* · *?hP*)
by (*auto intro*!: *wff-App wLG wff-Par wG*)
have *wih*: *wff*$_\sigma$(*?iP* · *?hP*) **by** (*auto intro*!: *wff-App wff-Par*)
have *dir1*: *?*Γ ∪ {*?PX* · *?hP*} ⊢ *?LG* · *?hP*
— if *h* is in the singleton then *i* · *h* = *i* · *?G*, so *h* = *?G* by the injectivity instance (2) and *NK*(⊃*E*)
proof −
let *?*Δ = *?*Γ ∪ {*?PX* · *?hP*}
have *fp*Δ: *freep ?*Δ **by** (*intro freep-finite*) *simp*
have *d1*: *?*Δ ⊢ *?PX* · *?hP* **by** (*auto intro*: *bprov.Hyp*)
have *imp*: *?*Δ ⊢ ((*?iP* · *?hP*) $=_\sigma$ *?a*) ⊃ (*?hP* $=_{\sigma \Rightarrow o}$ *?G*)
by (*rule bprov-weaken*[*OF 2*[*OF wff-Par wG wff-lc*[*OF wff-Par*]
lcG] - *fp*Δ]) *auto*
have *?*Δ ⊢ *?hP* $=_{\sigma \Rightarrow o}$ *?G*

by (*rule bprov-ImpE*[*OF imp leib-to-peq*[*OF bprov.Beta*[*OF A-beta d1*] *fp*Δ *wih wa*] *fp*Δ])
(*auto intro*!: *wih wa wff-Par wG*)
hence *?*Δ ⊢ *?hP* $\doteq_\sigma \Rightarrow_o$ *?G* **by** (*rule bprov.EqL*)
hence *?*Δ ⊢ *?G* $\doteq_\sigma \Rightarrow_o$ *?hP*
by (*rule leib-sym*[*OF - fp*Δ *wff-Par wG*])
thus *?thesis* **by** *simp*
qed
have *dir2*: *?*Γ ∪ {*?LG* · *?hP*} ⊢ *?PX* · *?hP*
— conversely, Leibniz-equals of *?G* lie in the singleton — by congruence and *NK*(*β*)
proof −
let *?*Δ = *?*Γ ∪ {*?LG* · *?hP*}
have *fp*Δ: *freep ?*Δ **by** (*intro freep-finite*) *simp*
have *d1*: *?*Δ ⊢ *?G* $\doteq_\sigma \Rightarrow_o$ *?hP* **by** (*auto intro*: *bprov.Hyp*)
show *?thesis* **by** (*rule bprov.Beta*[*OF beq.sym*[*OF A-beta*]
leib-cong2[*OF leib-sym*[*OF d1 fp*Δ *wG wff-Par*] *fp*Δ *wff-Par*
wG wff-Par]])
qed
have *bqh*: (**Λ**$_\sigma \Rightarrow_o$ *?body*) · *?hP* $\approx_o$ (*?PX* · *?hP*) $\doteq_o$ (*?LG* · *?hP*)
using *beq.beta*[*OF wbody wff-Par*]
by (*simp add*: *opn-lc*[*OF wff-lc*[*OF wPX*]] *opn-lc*[*OF wff-lc*[*OF wLG*]])
have *allh*: *?*Γ ⊢ **Π**$_\sigma \Rightarrow_o$ *?body*
— *NK*(*b*), then *NK*(Π*I*) at the fresh *h*
by (*rule PiI-Forall*[*OF bprov.Beta*[*OF beq.sym*[*OF bqh*]
bprov.BoolE[*OF dir1 dir2 wA wB*]] *wbody*])
(*use hi* **in** ‹*auto simp*: *Leib-def*›)
show *?thesis* — *NK*(*f*): functional extensionality
by (*rule bprov.FuncE*[*OF allh wPX wLG*])
qed
have *4*: *?*Γ ⊢ ((*Iota* (*σ* ⇒ o)) · *?PX*) $\doteq_\sigma \Rightarrow_o$ *?G*
— *NK*(*ι*): description maps the Leibniz singleton — and by (3) the singleton predicate of *?a* — to *?G* itself
by (*rule leib-trans*[*OF leib-cong2*[*OF 3 fp*Γ *wPX wLG wff-Iota*]
bprov.Desc[*OF wG*] *fp*Γ])
(*auto intro*!: *wff-App wff-Iota wPX wLG wG*)
have *5*: *?G* · *?a* $\approx_o$ ¬ *?ψ* — *β*-reduce the diagonal at *?a*
using *beq.beta*[*OF wG wa*] **by** (*simp add*: *opn-lc*[*OF lca*])
have *6*: *?*Γ ⊢ ⊥
— apply (4) at *?a*, *β*-reduce by (5): *?ψ* $\doteq$ ¬ *?ψ*, contradictory — *NK*(Π*E*), *NK*(¬*E*), tertium non datur
by (*rule leib-neg-contra*[*OF bprov.Beta*[*OF beq.appR*[*OF 5*
wff-App[*OF wff-Leib wψ*]]
leib-cong1[*OF 4 fp*Γ *wIo wG wa*]] *fp*Γ *wψ*])
have *7*: {*?S*} ⊢ ∃$_{(\sigma \Rightarrow o) \Rightarrow \sigma}$ *?IBb*
— *NK*(*Hyp*), in locally-nameless normal form
by (*subst inj-norm*[*symmetric*]) (*auto intro*: *bprov.Hyp*)
have *8*: {*?S*} ⊢ ⊥ — *NK*(∃ *E*), discharging the witness *i*

proof (*rule ExE*[**where** $w = i$, *OF 7 - wIBb wff-FalseB*], *goal-cases*)
case *1* **show** *?case* **using** *6* **by** (*simp add*: *insert-commute*)
next case *2* **show** *?case*
by (*simp add*: *Leib-def ImpB-def Forall-def*)
next case *3* **show** *?case* **by** (*simp add*: *FalseB-def Forall-def*)
next case *4* **show** *?case* **by** *auto*
next case *5* **show** *?case* **by** (*intro freep-finite*) *simp*
qed
show *?thesis* — $NK(\neg I)$ discharges the assumed injection
by (*rule bprov.NegI*[*OF - wff-inj*]) (*use 8* **in** *simp*)
qed

8 Main results

The central results of this entry, restated in one place. Soundness and completeness of *NK* for the class $\mathcal{M}_{\beta fb}$ (BKK Theorem 7.3 and a strengthening of BKK Corollary 7.7): for well-formed (open) formulas over every signature with infinitely many parameters, derivability coincides with validity at *every* infinite value carrier — no cardinality link between the parameter type $'p$ and the carrier $'u$ is needed. Consistency holds for arbitrary signatures, by the concrete standard model — note the asymmetry: consistency needs *no* constraint on $'p$ at all, whereas completeness requires $'p$ infinite, since the rule $NK(\Pi I)$ consumes fresh eigen-parameters. Cantor's theorem, surjective and injective, is derived inside *NK* at every type.

theorem *NK-soundness*:
$\Phi \vdash C \Longrightarrow \Phi \models('u)\ C$
using *soundness-sat*.

theorem *NK-completeness*:
assumes $wff_{\mathrm{o}}(A::'p::infinite\ tm)$ **and** $\models('u::infinite)\ A$
shows $\vdash A$
using *completeness-at-any-signature*[*OF assms*].

theorem *sound-and-complete*:
assumes $wff_{\mathrm{o}}(A::'p::infinite\ tm)$
shows $\vdash A \longleftrightarrow \models('u::infinite)\ A$
using *assms completeness-at-any-signature soundness-valid* **by** *blast*

theorem *consistency*: $\neg \vdash \bot$
by (*rule nk-consistent*)

corollary *no-contradiction*: $\vdash A \Longrightarrow \neg \vdash \neg A$
by (*rule nk-not-both*)

theorem *cantor-surjective*: **fixes** $\sigma :: ty$ **shows**
$\vdash (\neg\ (\exists\, \mathcal{G}_\sigma \Rightarrow \sigma \Rightarrow \mathrm{o}.\ \mathbf{\Pi} \mathcal{F}_\sigma \Rightarrow \mathrm{o}.\ \exists\, \mathcal{X}_\sigma.$

$((\mathcal{G}^f\,\sigma \Rightarrow \sigma \Rightarrow \mathrm{o} \cdot \mathcal{X}^f\,\sigma) =_\sigma \Rightarrow \mathrm{o}\ \mathcal{F}^f\,\sigma \Rightarrow \mathrm{o})) :: {'p}{::}\textit{infinite tm})$
by (*rule nk-surjective-cantor*)

theorem *cantor-injective*: **fixes** $\sigma :: ty$ **shows**
$\vdash (\neg\ (\exists \mathcal{I}_{(\sigma \Rightarrow \mathrm{o}) \Rightarrow \sigma} \cdot\ \mathbf{\Pi}\mathcal{F}_\sigma \Rightarrow \mathrm{o} \cdot\ \mathbf{\Pi}\mathcal{H}_\sigma \Rightarrow \mathrm{o} \cdot$
$(((\mathcal{I}^f{}_{(\sigma \Rightarrow \mathrm{o}) \Rightarrow \sigma} \cdot \mathcal{F}^f\,\sigma \Rightarrow \mathrm{o}) =_\sigma (\mathcal{I}^f{}_{(\sigma \Rightarrow \mathrm{o}) \Rightarrow \sigma} \cdot \mathcal{H}^f\,\sigma \Rightarrow \mathrm{o}))$
$\supset (\mathcal{F}^f\,\sigma \Rightarrow \mathrm{o} =_\sigma \Rightarrow \mathrm{o}\ \mathcal{H}^f\,\sigma \Rightarrow \mathrm{o}))) :: {'p}{::}\textit{infinite tm})$
by (*rule nk-injective-cantor*)

References

[1] P. B. Andrews. General models, descriptions, and choice in type theory. *Journal of Symbolic Logic*, 37(2):385–394, 1972.

[2] P. B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*, volume 27 of *Applied Logic Series*. Kluwer Academic Publishers, 2nd edition, 2002.

[3] C. Benzmüller and P. Andrews. Church's type theory. In E. N. Zalta and U. Nodelman, editors, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, 2024. Summer 2024 Edition, https://plato.stanford.edu/archives/sum2024/entries/type-theory-church/.

[4] C. Benzmüller, C. E. Brown, and M. Kohlhase. Higher-order semantics and extensionality. *Journal of Symbolic Logic*, 69(4):1027–1088, 2004.

[5] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(2):56–68, 1940.

[6] J. Díaz. Metatheory of $\mathcal{Q}_0$. *Archive of Formal Proofs*, November 2023. https://isa-afp.org/entries/Q0_Metatheory.html, Formal proof development.

[7] A. H. From. An Isabelle/HOL framework for synthetic completeness proofs. In *Proceedings of the 14th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2025)*. ACM, 2025.

[8] J. Harrison. Towards self-verification of HOL Light. In *Automated Reasoning (IJCAR 2006)*, volume 4130 of *Lecture Notes in Computer Science*, pages 177–191. Springer, 2006.

[9] M. Koch and D. Kirst. Undecidability, incompleteness, and completeness of second-order logic in Coq. In *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2022)*, pages 274–290. ACM, 2022.