Isabelle's Metalogic: Formalization and Proof Checker

Tobias Nipkow and Simon RoSSkopf

September 9, 2026

Abstract

In this entry we formalize Isabelle's metalogic in Isabelle/HOL. Furthermore, we define a language of proof terms and an executable proof checker and prove its soundness wrt. the metalogic.

The formalization is intentionally kept close to the Isabelle implementation(for example using de Brujin indices) to enable easy integration of generated code with the Isabelle system without a complicated translation layer.

The formalization is described in our CADE 28 paper[2].

Contents

1 Core Inference system **2**

2 Preliminaries **8**

3 Terms **14**

4 Sorts **36**

5 Wellformed Signature and Theory **40**

6 More on Substitutions **43**

7 Names **50**

8 Beta Normalization **52**

9 Eta Normalization **58**

10 Logic **61**

11 Derived rules on equality and normalization **76**

12 Proof Terms and proof checker **86**

13 Executable Sorts **92**

14 Executable Instance Relations **97**

15 Executable Signature and Theory **105**

16 Code Generation **113**

1 Core Inference system

Contains just the stuff necessary for the definition of the Inference system

theory *Core*
  imports *Main*
begin

Basic types

type-synonym *name* = *String.literal*
type-synonym *indexname* = *name* × *int*

type-synonym *class* = *String.literal*

type-synonym *sort* = *class set*
abbreviation *full-sort* ≡ ({}::*sort*)

datatype *variable* = *Free name* | *Var indexname*

datatype *typ* =
  is-Ty: *Ty name typ list* |
  is-Tv: *Tv variable sort*

datatype *term* =
  is-Ct: *Ct name typ* |
  is-Fv: *Fv variable typ* |
  is-Bv: *Bv nat* |
  is-Abs: *Abs typ term* |
  is-App: *App term term* (**infixl** ‹$› *100*)

abbreviation *mk-fun-typ S T* ≡ *Ty STR ''fun''* [*S*,*T*]
notation *mk-fun-typ* (**infixr** ‹→› *100*)

Collect variables in a term

fun *fv* :: *term* ⇒ (*variable* × *typ*) *set* **where**
  fv (*Ct* - -) = {}
| *fv* (*Fv v T*) = {(*v*, *T*)}

| *fv* (*Bv* -) = {}
| *fv* (*Abs* - *body*) = *fv body*
| *fv* (*t* $ *u*) = *fv t* ∪ *fv u*
definition [*simp*]: *FV S* = (⋃ *s*∈*S* . *fv s*)

Typ/term instantiations

fun *tsubstT* :: *typ* ⇒ (*variable* ⇒ *sort* ⇒ *typ*) ⇒ *typ* **where**
  tsubstT (*Tv a s*) *ϱ* = *ϱ a s*
| *tsubstT* (*Ty* κ *σs*) *ϱ* = *Ty* κ (*map* (λ*σ*. *tsubstT σ ϱ*) *σs*)
definition *tinstT T1 T2* ≡ ∃ *ϱ*. *tsubstT T2 ϱ* = *T1*

fun *tsubst* :: *term* ⇒ (*variable* ⇒ *sort* ⇒ *typ*) ⇒ *term* **where**
  tsubst (*Ct s T*) *ϱ* = *Ct s* (*tsubstT T ϱ*)
| *tsubst* (*Fv v T*) *ϱ* = *Fv v* (*tsubstT T ϱ*)
| *tsubst* (*Bv i*) - = *Bv i*
| *tsubst* (*Abs T t*) *ϱ* = *Abs* (*tsubstT T ϱ*) (*tsubst t ϱ*)
| *tsubst* (*t* $ *u*) *ϱ* = *tsubst t ϱ* $ *tsubst u ϱ*

Typ of a term

inductive *has-typ1* :: *typ list* ⇒ *term* ⇒ *typ* ⇒ *bool* (‹- ⊢$_\tau$ - : -› [*51*, *51*, *51*] *51*)
where
  has-typ1 - (*Ct* - *T*) *T*
| *i* < *length Ts* ⟹ *has-typ1 Ts* (*Bv i*) (*nth Ts i*)
| *has-typ1* - (*Fv* - *T*) *T*
| *has-typ1* (*T*#*Ts*) *t T′* ⟹ *has-typ1 Ts* (*Abs T t*) (*T* → *T′*)
| *has-typ1 Ts u U* ⟹ *has-typ1 Ts t* (*U* → *T*) ⟹
    has-typ1 Ts (*t* $ *u*) *T*
definition *has-typ* :: *term* ⇒ *typ* ⇒ *bool* (‹⊢$_\tau$ - : -› [*51*, *51*] *51*) **where** *has-typ t T* = *has-typ1* [] *t T*

definition *typ-of t* = (*if* ∃ *T* . *has-typ t T then Some* (*THE T* . *has-typ t T*) *else None*)

More operations on terms

fun *lift* :: *term* ⇒ *nat* ⇒ *term* **where**
  lift (*Bv i*) *n* = (*if i* ≥ *n then Bv* (*i*+*1*) *else Bv i*)
| *lift* (*Abs T body*) *n* = *Abs T* (*lift body* (*n*+*1*))
| *lift* (*App f t*) *n* = *App* (*lift f n*) (*lift t n*)
| *lift u n* = *u*

fun *subst-bv2* :: *term* ⇒ *nat* ⇒ *term* ⇒ *term* **where**
  subst-bv2 (*Bv i*) *n u* = (*if i* < *n then Bv i*
    else if i = *n then u*
    else (*Bv* (*i* − *1*)))
| *subst-bv2* (*Abs T body*) *n u* = *Abs T* (*subst-bv2 body* (*n* + *1*) (*lift u 0*))
| *subst-bv2* (*f* $ *t*) *n u* = *subst-bv2 f n u* $ *subst-bv2 t n u*
| *subst-bv2 t* - - = *t*

definition *subst-bv u t* = *subst-bv2 t 0 u*

fun *bind-fv2* :: (*variable* × *typ*) ⇒ *nat* ⇒ *term* ⇒ *term* **where**
  bind-fv2 vT n (*Fv v T*) = (*if vT* = (*v*,*T*) *then Bv n else Fv v T*)
| *bind-fv2 vT n* (*Abs T t*) = *Abs T* (*bind-fv2 vT* (*n*+*1*) *t*)
| *bind-fv2 vT n* (*f* $ *u*) = *bind-fv2 vT n f* $ *bind-fv2 vT n u*
| *bind-fv2* - - *t* = *t*

definition *bind-fv vT t* = *bind-fv2 vT 0 t*

abbreviation *Abs-fv v T t* ≡ *Abs T* (*bind-fv* (*v*,*T*) *t*)

Some typ/term constants

abbreviation *itselfT ty* ≡ *Ty STR ''itself''* [*ty*]
abbreviation *constT name* ≡ *Ty name* []
abbreviation *propT* ≡ *constT STR ''prop''*

abbreviation *mk-eq t1 t2* ≡ *Ct STR ''Pure.eq''*
  (*the* (*typ-of t1*) → (*the* (*typ-of t2*) → *propT*)) $ *t1* $ *t2*

abbreviation *mk-eq' ty t1 t2* ≡ *Ct STR ''Pure.eq''*
  (*ty* → (*ty* → *propT*)) $ *t1* $ *t2*
abbreviation *mk-imp* :: *term* ⇒ *term* ⇒ *term* (**infixr** ‹⟼› *51*) **where**
  A ⟼ *B* ≡ *Ct STR ''Pure.imp''* (*propT* → (*propT* → *propT*)) $ *A* $ *B*
abbreviation *mk-all x ty t* ≡
  Ct STR ''Pure.all'' ((*ty* → *propT*) → *propT*) $ *Abs-fv x ty t*

Order sorted signature

type-synonym *osig* = (*class rel* × (*name* ⇀ (*class* ⇀ *sort list*)))

fun *subclass* :: *osig* ⇒ *class rel* **where** *subclass* (*cl*, -) = *cl*
fun *tcsigs* :: *osig* ⇒ (*name* ⇀ (*class* ⇀ *sort list*)) **where** *tcsigs* (-, *ars*) = *ars*

Relation in sorts

definition *class-leq sub c1 c2* = ((*c1*,*c2*) ∈ *sub*)
definition *class-les sub c1 c2* = (*class-leq sub c1 c2* ∧ ¬ *class-leq sub c2 c1*)
definition *sort-leq sub s1 s2* = (∀ c_2 ∈ *s2* . ∃ c_1 ∈ *s1*. *class-leq sub* c_1 c_2)

Is a class/sort defined

definition *class-ex rel c* = (*c* ∈ *Field rel*)
definition *sort-ex rel S* = (*S* ⊆ *Field rel*)

Normalizing sorts

definition *normalize-sort sub* (*S*::*sort*)
  = {*c* ∈ *S*. ¬ (∃ *c'* ∈ *S*. *class-les sub c' c*)}
abbreviation *normalized-sort sub S* ≡ *normalize-sort sub S* = *S*

definition *wf-sort sub S* = (*normalized-sort sub S* ∧ *sort-ex sub S*)

Wellformedness of osig

definition [*simp*]: *wf-subclass rel* =
(*rel* ⊆ *Field rel* × *Field rel* ∧ *trans rel* ∧ *antisym rel* ∧ *Refl rel*)

definition *complete-tcsigs sub tcs* ≡ (∀ *ars* ∈ *ran tcs* .
∀ (*c*$_1$, *c*$_2$) ∈ *sub* . *c*$_1$∈*dom ars* ⟶ *c*$_2$∈*dom ars*)

definition *coregular-tcsigs sub tcs* ≡ (∀ *ars* ∈ *ran tcs* .
∀ *c*$_1$ ∈ *dom ars*. ∀ *c*$_2$ ∈ *dom ars*.
(*class-leq sub c*$_1$ *c*$_2$ ⟶ *list-all2* (*sort-leq sub*) (*the* (*ars c*$_1$)) (*the* (*ars c*$_2$))))

definition *consistent-length-tcsigs tcs* ≡ (∀ *ars* ∈ *ran tcs* .
∀ *ss*$_1$ ∈ *ran ars*. ∀ *ss*$_2$ ∈ *ran ars*. *length ss*$_1$ = *length ss*$_2$)

definition *all-normalized-and-ex-tcsigs sub tcs* ≡
(∀ *ars* ∈ *ran tcs* . ∀ *ss* ∈ *ran ars* . ∀ *s* ∈ *set ss*. *wf-sort sub s*)

definition [*simp*]: *wf-tcsigs sub tcs* ⟷
coregular-tcsigs sub tcs
∧ *complete-tcsigs sub tcs*
∧ *consistent-length-tcsigs tcs*
∧ *all-normalized-and-ex-tcsigs sub tcs*

fun *wf-osig* **where** *wf-osig* (*sub*, *tcs*) ⟷ *wf-subclass sub* ∧ *wf-tcsigs sub tcs*

Embedding typs into terms/Encoding of type classes

definition *mk-type ty* = *Ct STR ''Pure.type''* (*Core.itselfT ty*)

abbreviation *mk-suffix* (*str*::*name*) *suff* ≡ *String.implode* (*String.explode str* @ *String.explode suff*)

abbreviation *classN* ≡ *STR ''-class''*
abbreviation *const-of-class name* ≡ *mk-suffix name classN*

definition *mk-of-class ty c* =
Ct (*const-of-class c*) (*Core.itselfT ty* → *propT*) $ *mk-type ty*

Checking if a typ belongs to a sort

inductive *has-sort* :: *osig* ⇒ *typ* ⇒ *sort* ⇒ *bool* **where**
has-sort-Tv[*intro*]: *sort-leq sub S S'* ⟹ *has-sort* (*sub*, *tcs*) (*Tv a S*) *S'*
| *has-sort-Ty*:
tcs κ = *Some dm* ⟹ ∀ *c* ∈ *S*. ∃ *Ss* . *dm c* = *Some Ss* ∧ *list-all2* (*has-sort* (*sub*, *tcs*)) *Ts Ss*
⟹ *has-sort* (*sub*, *tcs*) (*Ty κ Ts*) *S*

Signatures

type-synonym *signature* = (*name* ⇀ *typ*) × (*name* ⇀ *nat*) × *osig*

fun *const-type* :: *signature* ⇒ (*name* ⇀ *typ*) **where** *const-type* (*ctf*, -, -) = *ctf*
fun *type-arity* :: *signature* ⇒ (*name* ⇀ *nat*) **where** *type-arity* (-, *arf*, -) = *arf*

fun *osig* :: *signature* ⇒ *osig* **where** *osig* (-, -, *oss*) = *oss*

fun *is-std-sig* **where** *is-std-sig* (*ctf*, *arf*, -) ⟷
  arf STR ''fun'' = *Some 2* ∧ *arf STR ''prop''* = *Some 0*
 ∧ *arf STR ''itself''* = *Some 1*
 ∧ *ctf STR ''Pure.eq''*
  = *Some* ((*Tv* (*Var* (*STR '''a''*, *0*)) *full-sort*) → ((*Tv* (*Var* (*STR '''a''*, *0*)) *full-sort*) → *propT*))
 ∧ *ctf STR ''Pure.all''* = *Some* ((*Tv* (*Var* (*STR '''a''*, *0*)) *full-sort* → *propT*) → *propT*)
 ∧ *ctf STR ''Pure.imp''* = *Some* (*propT* → (*propT* → *propT*))
 ∧ *ctf STR ''Pure.type''* = *Some* (*itselfT* (*Tv* (*Var* (*STR '''a''*, *0*)) *full-sort*))

Wellformedness checks

definition [*simp*]: *class-ok-sig* Σ *c* ≡ *class-ex* (*subclass* (*osig* Σ)) *c*

inductive *wf-type* :: *signature* ⇒ *typ* ⇒ *bool* **where**
  typ-ok-Ty: *type-arity* Σ κ = *Some* (*length Ts*) ⟹ ∀ *T*∈*set Ts* . *wf-type* Σ *T*
   ⟹ *wf-type* Σ (*Ty* κ *Ts*)
| *typ-ok-Tv*[*intro*]: *wf-sort* (*subclass* (*osig* Σ)) *S* ⟹ *wf-type* Σ (*Tv a S*)

inductive *wf-term* :: *signature* ⇒ *term* ⇒ *bool* **where**
  wf-type Σ *T* ⟹ *wf-term* Σ (*Fv v T*)
| *wf-term* Σ (*Bv n*)
| *const-type* Σ *s* = *Some ty* ⟹ *wf-type* Σ *T* ⟹ *tinstT T ty* ⟹ *wf-term* Σ (*Ct s T*)
| *wf-term* Σ *t* ⟹ *wf-term* Σ *u* ⟹ *wf-term* Σ (*t* $ *u*)
| *wf-type* Σ *T* ⟹ *wf-term* Σ *t* ⟹ *wf-term* Σ (*Abs T t*)

definition *wt-term* Σ *t* ≡ *wf-term* Σ *t* ∧ (∃ *T*. *has-typ t T*)

fun *wf-sig* :: *signature* ⇒ *bool* **where**
  wf-sig (*ctf*, *arf*, *oss*) = (*wf-osig oss*
  ∧ *dom* (*tcsigs oss*) = *dom arf*
  ∧ (∀ *type* ∈ *dom* (*tcsigs oss*). (∀ *ars* ∈ *ran* (*the* (*tcsigs oss type*)) . *the* (*arf type*) = *length ars*))
  ∧ (∀ *ty* ∈ *Map.ran ctf* . *wf-type* (*ctf*, *arf*, *oss*) *ty*))

Theories

type-synonym *theory* = *signature* × *term set*

fun *sig* :: *theory* ⇒ *signature* **where** *sig* (Σ, -) = Σ
fun *axioms* :: *theory* ⇒ *term set* **where** *axioms* (-, *axs*) = *axs*

Equality axioms, stated directly

abbreviation *tvariable a* ≡ (*Tv* (*Var* (*a*, *0*)) *full-sort*)
abbreviation *variable x T* ≡ *Fv* (*Var* (*x*, *0*)) *T*

abbreviation $aT \equiv tvariable\ STR\ ''a''$
abbreviation $bT \equiv tvariable\ STR\ ''b''$
abbreviation $x \equiv variable\ STR\ ''x''\ aT$
abbreviation $y \equiv variable\ STR\ ''y''\ aT$
abbreviation $z \equiv variable\ STR\ ''z''\ aT$
abbreviation $f \equiv variable\ STR\ ''f''\ (aT \to bT)$
abbreviation $g \equiv variable\ STR\ ''g''\ (aT \to bT)$
abbreviation $P \equiv variable\ STR\ ''P''\ (aT \to propT)$
abbreviation $Q \equiv variable\ STR\ ''Q''\ (aT \to propT)$
abbreviation $A \equiv variable\ STR\ ''A''\ propT$
abbreviation $B \equiv variable\ STR\ ''B''\ propT$

definition *eq-reflexive-ax* $\equiv$ *mk-eq x x*
definition *eq-symmetric-ax* $\equiv$ *mk-eq x y* $\longmapsto$ *mk-eq y x*
definition *eq-transitive-ax* $\equiv$ *mk-eq x y* $\longmapsto$ *mk-eq y z* $\longmapsto$ *mk-eq x z*
definition *eq-intr-ax* $\equiv (A \longmapsto B) \longmapsto (B \longmapsto A) \longmapsto$ *mk-eq A B*
definition *eq-elim-ax* $\equiv$ *mk-eq A B* $\longmapsto A \longmapsto B$
definition *eq-combination-ax* $\equiv$ *mk-eq f g* $\longmapsto$ *mk-eq x y* $\longmapsto$ *mk-eq* $(f\ \$\ x)\ (g\ \$\ y)$
definition *eq-abstract-rule-ax* $\equiv$
$(Ct\ STR\ ''Pure.all''\ ((aT \to propT) \to propT)\ \$\ Abs\ aT\ (mk\text{-}eq'\ bT\ (f\ \$\ Bv\ 0)\ (g\ \$\ Bv\ 0)))$
$\longmapsto$ *mk-eq* $(Abs\ aT\ (f\ \$\ Bv\ 0))\ (Abs\ aT\ (g\ \$\ Bv\ 0))$

hide-const (**open**) *x y z f g P Q A B*

abbreviation *eq-axs* $\equiv \{$*eq-reflexive-ax*, *eq-symmetric-ax*, *eq-transitive-ax*, *eq-intr-ax*, *eq-elim-ax*,
eq-combination-ax, *eq-abstract-rule-ax*$\}$

Wellformedness of theories

fun *wf-theory* **where** *wf-theory* $(\Sigma,\ axs) \longleftrightarrow$
$(\forall\, p \in axs\ .\ wt\text{-}term\ \Sigma\ p \wedge has\text{-}typ\ p\ propT)$
$\wedge$ *is-std-sig* Σ
$\wedge$ *wf-sig* Σ
$\wedge$ *eq-axs* $\subseteq$ *axs*

Wellformedness of typ antiations

definition [*simp*]: *wf-inst* $\Theta\ \varrho \equiv$
$(\forall\, v\ S\ .\ \varrho\ v\ S \neq Tv\ v\ S \longrightarrow$
$(has\text{-}sort\ (osig\ (sig\ \Theta))\ (\varrho\ v\ S)\ S) \wedge wf\text{-}type\ (sig\ \Theta)\ (\varrho\ v\ S))$

Inference system

inductive *proves* :: *theory* $\Rightarrow$ *term set* $\Rightarrow$ *term* $\Rightarrow$ *bool* $(\langle(\text{-},\text{-})\vdash(\text{-})\rangle\ 50)$ **for** Θ **where**
axiom: *wf-theory* $\Theta \Longrightarrow A \in axioms\ \Theta \Longrightarrow wf\text{-}inst\ \Theta\ \varrho$
$\Longrightarrow \Theta, \Gamma \vdash tsubst\ A\ \varrho$
| *assume*: *wf-term* $(sig\ \Theta)\ A \Longrightarrow has\text{-}typ\ A\ propT \Longrightarrow A \in \Gamma \Longrightarrow \Theta,\Gamma \vdash A$
| *forall-intro*: *wf-theory* $\Theta \Longrightarrow \Theta, \Gamma \vdash B \Longrightarrow (x,\tau) \notin FV\ \Gamma \Longrightarrow wf\text{-}type\ (sig\ \Theta)\ \ \tau$

$\Longrightarrow \Theta, \Gamma \vdash mk\text{-}all\ x\ \tau\ B$
| *forall-elim*: $\Theta, \Gamma \vdash Ct\ STR\ ''Pure.all''\ ((\tau \to propT) \to propT)\ \$\ Abs\ \tau\ B$
$\Longrightarrow has\text{-}typ\ a\ \tau \Longrightarrow wf\text{-}term\ (sig\ \Theta)\ a$
$\Longrightarrow \Theta, \Gamma \vdash subst\text{-}bv\ a\ B$
| *implies-intro*: $wf\text{-}theory\ \Theta \Longrightarrow \Theta, \Gamma \vdash B \Longrightarrow wf\text{-}term\ (sig\ \Theta)\ A \Longrightarrow has\text{-}typ\ A\ propT$
$\Longrightarrow \Theta, \Gamma - \{A\} \vdash A \longmapsto B$
| *implies-elim*: $\Theta, \Gamma_1 \vdash A \longmapsto B \Longrightarrow \Theta, \Gamma_2 \vdash A \Longrightarrow \Theta, \Gamma_1 \cup \Gamma_2 \vdash B$
| *of-class*: $wf\text{-}theory\ \Theta$
$\Longrightarrow const\text{-}type\ (sig\ \Theta)\ (const\text{-}of\text{-}class\ c) = Some\ (Core.itselfT\ aT \to propT)$
$\Longrightarrow wf\text{-}type\ (sig\ \Theta)\ T$
$\Longrightarrow has\text{-}sort\ (osig\ (sig\ \Theta))\ T\ \{c\}$
$\Longrightarrow \Theta, \Gamma \vdash mk\text{-}of\text{-}class\ T\ c$

| β-*conversion*: $wf\text{-}theory\ \Theta \Longrightarrow wt\text{-}term\ (sig\ \Theta)\ (Abs\ T\ t) \Longrightarrow wf\text{-}term\ (sig\ \Theta)\ u \Longrightarrow has\text{-}typ\ u\ T$
$\Longrightarrow \Theta, \Gamma \vdash mk\text{-}eq\ (Abs\ T\ t\ \$\ u)\ (subst\text{-}bv\ u\ t)$
| *eta*: $wf\text{-}theory\ \Theta \Longrightarrow wf\text{-}term\ (sig\ \Theta)\ t \Longrightarrow has\text{-}typ\ t\ (\tau \to \tau') \Longrightarrow \Theta, \Gamma \vdash mk\text{-}eq\ (Abs\ \tau\ (t\ \$\ Bv\ 0))\ t$

Ensure no garbage in Θ,Γ

definition *proves'* :: *theory* ⇒ *term set* ⇒ *term* ⇒ *bool* (‹(-,-) ⊩ (-)› *51*) **where**
proves' Θ Γ *t* ≡ *wf-theory* Θ ∧ (∀ *h* ∈ Γ . *wf-term* (*sig* Θ) *h* ∧ *has-typ h propT*) ∧ Θ, Γ ⊢ *t*

hide-const (**open**) *aT bT*

end

2 Preliminaries

theory *Preliminaries*
imports *Complex-Main*
List−Index.List-Index
HOL−Library.AList
HOL−Library.Sublist
HOL−Eisbach.Eisbach
HOL−Library.Simps-Case-Conv
begin

Stuff about options

fun *the-default* :: *'a* ⇒ *'a option* ⇒ *'a* **where**
the-default a None = *a*
| *the-default* - (*Some b*) = *b*

abbreviation *Or* :: *'a option* ⇒ *'a option* ⇒ *'a option* (**infixl** ‹*OR*› *60*) **where**
e1 OR e2 ≡ *case e1 of None* ⇒ *e2* | *p* ⇒ *p*

lemma *Or-Some*: $(e1\ OR\ e2) = Some\ x \longleftrightarrow e1 = Some\ x \vee (e1 = None \wedge e2 = Some\ x)$
⟨*proof*⟩

lemma *Or-None*: $(e1\ OR\ e2) = None \longleftrightarrow e1 = None \wedge e2 = None$
⟨*proof*⟩

fun *lift2-option* :: $('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow 'a\ option \Rightarrow 'b\ option \Rightarrow 'c\ option$ **where**
lift2-option - None - = None |
lift2-option - - None = None |
lift2-option f (Some x) (Some y) = Some (f x y)

lemma *lift2-option-not-None*: $lift2\text{-}option\ f\ x\ y \neq None \longleftrightarrow (x \neq None \wedge y \neq None)$
⟨*proof*⟩
lemma *lift2-option-None*: $lift2\text{-}option\ f\ x\ y = None \longleftrightarrow (x = None \vee y = None)$
⟨*proof*⟩

Lookup functions for assoc lists

fun *find* :: $('a \Rightarrow 'b\ option) \Rightarrow 'a\ list \Rightarrow 'b\ option$ **where**
find f [] = None |
find f (x#xs) = f x OR find f xs

lemma *findD*:
$find\ f\ xs = Some\ p \Longrightarrow \exists x \in set\ xs.\ f\ x = Some\ p$
⟨*proof*⟩

lemma *find-None*:
$find\ f\ xs = None \longleftrightarrow (\forall x \in set\ xs.\ f\ x = None)$
⟨*proof*⟩

lemma *find-ListFind*: $find\ f\ l = Option.bind\ (List.find\ (\lambda x.\ case\ f\ x\ of\ None \Rightarrow False \mid - \Rightarrow True)\ l)\ f$
⟨*proof*⟩

lemma $List.find\ P\ l = Some\ p \Longrightarrow \exists p \in set\ l\ .\ P\ p$
⟨*proof*⟩

lemma *find-the-pair*:
assumes *distinct (map fst pairs)*
and $\bigwedge x\ y.\ x \in set\ (map\ fst\ pairs) \Longrightarrow y \in set\ (map\ fst\ pairs) \Longrightarrow P\ x \Longrightarrow P\ y \Longrightarrow x = y$
and $(x,y) \in set\ pairs$ **and** $P\ x$
shows $List.find\ (\lambda(x,\text{-})\ .\ P\ x)\ pairs = Some\ (x,y)$
⟨*proof*⟩

fun *remdups-on* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow 'a\ list$ **where**
remdups-on - [] = []
| *remdups-on cmp (x # xs) =*

(*if* $\exists x' \in$ *set xs . cmp x x' then remdups-on cmp xs else x # remdups-on cmp xs*)

fun *distinct-on* :: ('*a* ⇒ '*a* ⇒ *bool*) ⇒ '*a list* ⇒ *bool* **where**
distinct-on - [] ⟷ True
| *distinct-on cmp (x # xs) ⟷ ¬(*$\exists x' \in$ *set xs . cmp x x') ∧ distinct-on cmp xs*

lemma *remdups-on (=) xs = remdups xs*
⟨*proof*⟩

lemma *remdups-on-antimono*:
(⋀*x y . f x y ⟹ g x y*) ⟹ *set* (*remdups-on g xs*) ⊆ *set* (*remdups-on f xs*)
⟨*proof*⟩

lemma *remdups-on-subset-input*: *set* (*remdups-on f xs*) ⊆ *set xs*
⟨*proof*⟩

lemma *distinct-on-remdups-on*: *distinct-on f* (*remdups-on f xs*)
⟨*proof*⟩

lemma *distinct-on-no-compare*: (⋀*x y . f x y ⟹ f y x*)⟹
distinct-on f xs ⟹ x∈*set xs ⟹ y*∈*set xs ⟹ x*≠*y ⟹* ¬ *f x y*
⟨*proof*⟩

fun *lookup* :: ('*a* ⇒ *bool*) ⇒ ('*a* × '*b*) *list* ⇒ '*b option* **where**
lookup - [] = None
| *lookup f* ((*x*,*y*)#*xs*) = (*if f x then Some y else lookup f xs*)

lemma *lookup-present-eq-key*: *distinct* (*map fst al*) ⟹ (*k*, *v*) ∈ *set al* ⟷ *lookup* (λ*x*. *x*=*k*) *al* = *Some v*
⟨*proof*⟩

lemma *lookup-None-iff*: *lookup P xs = None* ⟷ ¬ (∃ *x*. *x* ∈ *set* (*map fst xs*) ∧ *P x*)
⟨*proof*⟩

lemma *find-Some*: *List.find P l = Some p ⟹ p*∈*set l* ∧ *P p*
⟨*proof*⟩

lemma *find-Some-imp-lookup-Some*:
List.find (λ(*k*,-). *P k*) *xs = Some* (*k*,*v*) ⟹ *lookup P xs = Some v*
⟨*proof*⟩

lemma *lookup-Some-imp-find-Some*:
lookup P xs = Some v ⟹ ∃ *x*. *List.find* (λ(*k*,-). *P k*) *xs = Some* (*x*,*v*)
⟨*proof*⟩

lemma *lookup-None-iff-find-None*: *lookup P xs = None* $\longleftrightarrow$ *List.find* (λ(*k*,-). *P k*) *xs = None*
⟨*proof*⟩

lemma *lookup-eq-order-irrelevant*:
assumes *distinct* (*map fst pairs*) **and** *distinct* (*map fst pairs'*) **and** *set pairs = set pairs'*
shows *lookup* ($\lambda x.\ x{=}k$) *pairs = lookup* ($\lambda x.\ x{=}k$) *pairs'*
⟨*proof*⟩

lemma *lookup-Some-append-back*:
lookup ($\lambda x.\ x{=}k$) *insts = Some v* $\Longrightarrow$ *lookup* ($\lambda x.\ x{=}k$) (*insts*@[(*k*,*v'*)]) *= Some v*
⟨*proof*⟩

lemma *lookup-eq-key-not-present*: *key* $\notin$ *set* (*map fst inst*) $\Longrightarrow$ *lookup* ($\lambda x.\ x = key$) *inst = None*
⟨*proof*⟩

lemma *lookup-in-empty*[*simp*]: *lookup f* [] *= None* ⟨*proof*⟩
lemma *lookup-in-single*[*simp*]: *lookup f* [(*k*, *v*)] = (*if f k then Some v else None*) ⟨*proof*⟩

lemma *lookup-present-eq-key'*: *lookup* ($\lambda x.\ x{=}k$) *al = Some v* $\Longrightarrow$ (*k*, *v*) $\in$ *set al*
⟨*proof*⟩

lemma *lookup-present-eq-key''*: *distinct* (*map fst al*) $\Longrightarrow$ *lookup* ($\lambda x.\ x{=}k$) *al = Some v* $\longleftrightarrow$ (*k*, *v*) $\in$ *set al*
⟨*proof*⟩

lemma *key-present-imp-eq-lookup-finds-value*: *k* $\in$ *fst* ‘ *set al* $\Longrightarrow$ $\exists$ *v* . *lookup* ($\lambda x.\ x{=}k$) *al = Some v*
⟨*proof*⟩

lemma *list-allI*: ($\bigwedge x.\ x{\in}set\ l \Longrightarrow P\ x$) $\Longrightarrow$ *list-all P l*
⟨*proof*⟩

lemma *map2-sym*: ($\bigwedge x\ y\ .\ f\ x\ y = f\ y\ x$) $\Longrightarrow$ *map2 f xs ys = map2 f ys xs*
⟨*proof*⟩

lemma *idem-map2*: **assumes** ($\bigwedge x.\ f\ x\ x = x$) **shows** *map2 f l l = l*
⟨*proof*⟩

lemma *rev-induct2*[*consumes 1*, *case-names Nil snoc*]:
assumes *length xs = length ys*
assumes *P* [] []
assumes ($\bigwedge x\ xs\ y\ ys.\ length\ xs = length\ ys \Longrightarrow P\ xs\ ys \Longrightarrow P\ (xs\ @\ [x])\ (ys\ @\ [y])$)
shows *P xs ys*

⟨*proof*⟩

lemma *alist-map-corr*: *distinct* (*map fst al*) ⟹ (*k*,*v*) ∈ *set al* ⟷ *map-of al k* = *Some v*
⟨*proof*⟩

lemma *distinct-fst-imp-distinct*: *distinct* (*map fst l*) ⟹ *distinct l*
⟨*proof*⟩

lemma *length-alist*:
assumes *distinct* (*map fst al*) **and** *distinct* (*map fst al'*) **and** *set al* = *set al'*
shows *length al* = *length al'*
⟨*proof*⟩

lemma *same-map-of-imp-same-length*:
distinct (*map fst ars1*) ⟹ *distinct* (*map fst ars2*) ⟹ *map-of ars1* = *map-of ars2*
⟹ *length ars1* = *length ars2*

⟨*proof*⟩

lemma *in-range-if-ex-key*: *v* ∈ *ran m* ⟷ (∃ *k*. *m k* = *Some v*)
⟨*proof*⟩

lemma *set-AList-delete-bound*: *set* (*AList.delete a l*) ⊆ *set l*
⟨*proof*⟩

lemma *list-all-clearjunk-cons*:
list-all P (*x*#(*AList.clearjunk l*)) ⟹ *list-all P* (*AList.clearjunk* (*x*#*l*))
⟨*proof*⟩

lemma *lookup-AList-delete*: *k'*≠*k* ⟹ *lookup* (λ*x*. *x* = *k*) *al* = *lookup* (λ*x*. *x* = *k*) (*AList.delete k' al*)
⟨*proof*⟩

lemma *lookup-AList-clearjunk*: *lookup* (λ*x*. *x* = *k*) *al* = *lookup* (λ*x*. *x* = *k*) (*AList.clearjunk al*)
⟨*proof*⟩

lemma *fold-Option-bind-eq-Some-start-not-None*:
fold (λ*new option* . *Option.bind option* (*f new*)) *list start* = *Some res*
⟹ *start* ≠ *None*
⟨*proof*⟩

lemma *fold-Option-bind-eq-Some-at-point-not-None*:
fold (λ*new option* . *Option.bind option* (*f new*)) (*l1*@*l2*) *start* = *Some res*
⟹ *fold* (λ*new option* . *Option.bind option* (*f new*)) (*l1*) *start* ≠ *None*
⟨*proof*⟩

lemma *fold-Option-bind-eq-Some-start-not-None'*:
fold (λ(*x*,*y*) *option* . *Option.bind option* (*f x y*)) *list start* = *Some res*
⟹ *start* ≠ *None*
⟨*proof*⟩

lemma *fold-Option-bind-eq-None-start-None*:
fold (λ(*x*,*y*) *option* . *Option.bind option* (*f x y*)) *list None* = *None*
⟨*proof*⟩

lemma *fold-Option-bind-at-some-point-None-eq-None*:
fold (λ(*x*,*y*) *option* . *Option.bind option* (*f x y*)) *l1 start* = *None* ⟹
fold (λ(*x*,*y*) *option* . *Option.bind option* (*f x y*)) (*l1*@*l2*) *start* = *None*
⟨*proof*⟩

lemma *fold-Option-bind-eq-Some-at-each-point-Some*:
fold (λ(*x*,*y*) *option* . *Option.bind option* (*f x y*)) (*l1*@*l2*) *start* = *Some res*
⟹ (∃ *point* . *fold* (λ(*x*,*y*) *option* . *Option.bind option* (*f x y*)) *l1 start* = *Some point*
∧ *fold* (λ(*x*,*y*) *option* . *Option.bind option* (*f x y*)) *l2* (*Some point*) = *Some res*)
⟨*proof*⟩

lemma *fold-Option-bind-eq-Some-at-each-point-Some'*:
assumes *fold* (λ(*x*,*y*) *option* . *Option.bind option* (*f x y*)) (*xs*@*ys*) *start* = *Some res*
obtains *point* **where**
fold (λ(*x*,*y*) *option* . *Option.bind option* (*f x y*)) *xs start* = *Some point* **and**
fold (λ(*x*,*y*) *option* . *Option.bind option* (*f x y*)) *ys* (*Some point*) = *Some res*
⟨*proof*⟩

corollary *fold-Option-bind-eq-Some-at-point-not-None'*:
fold (λ(*x*,*y*) *option* . *Option.bind option* (*f x y*)) (*l1*@*l2*) *start* = *Some res*
⟹ *fold* (λ(*x*,*y*) *option* . *Option.bind option* (*f x y*)) (*l1*) *start* ≠ *None*
⟨*proof*⟩

lemma *fold-matches-first-step-not-None*:
assumes
fold (λ(*T*, *U*) *subs* . *Option.bind subs* (*f T U*)) (*zip* (*x*#*xs*) (*y*#*ys*)) (*Some subs*) = *Some subs'*
obtains *point* **where**
f x y subs = *Some point*
fold (λ(*T*, *U*) *subs* . *Option.bind subs* (*f T U*)) (*zip* (*xs*) (*ys*)) (*Some point*) = *Some subs'*
⟨*proof*⟩

lemma *fold-matches-last-step-not-None*:
assumes

length xs = length ys
fold (λ(*T*, *U*) *subs* . *Option.bind subs* (*f T U*)) (*zip* (*xs*@[*x*]) (*ys*@[*y*])) (*Some subs*) = *Some subs′*
obtains *point* **where**
fold (λ(*T*, *U*) *subs* . *Option.bind subs* (*f T U*)) (*zip* (*xs*) (*ys*)) (*Some subs*) = *Some point*
f x y point = *Some subs′*
⟨*proof*⟩

end

3 Terms

Originally based on `~~/src/Pure/term.ML`. Diverged substantially, but some influences are still visible. Further influences from `~~/src/HOL/Proofs/Lambda/`.

theory *Term*
imports *Main Core Preliminaries*
begin

Collecting parts of typs/terms and more substitutions

fun *tvsT* :: *typ* ⇒ (*variable* × *sort*) *set* **where**
tvsT (*Tv v S*) = {(*v*,*S*)}
| *tvsT* (*Ty - Ts*) = ⋃(*set* (*map tvsT Ts*))

fun *tvs* :: *term* ⇒ (*variable* × *sort*) *set* **where**
tvs (*Ct - T*) = *tvsT T*
| *tvs* (*Fv - T*) = *tvsT T*
| *tvs* (*Bv* -) = {}
| *tvs* (*Abs T t*) = *tvsT T* ∪ *tvs t*
| *tvs* (*t* $ *u*) = *tvs t* ∪ *tvs u*

abbreviation *tvs-set S* ≡ ⋃*t*∈*S* . *tvs t*

lemma *tvsT-tsubstT*: *tvsT* (*tsubstT σ ϱ*) = ⋃ {*tvsT* (*ϱ a s*) | *a s*. (*a*, *s*) ∈ *tvsT σ*}
⟨*proof*⟩

lemma *tsubstT-cong*:
(∀(*v*,*S*) ∈ *tvsT σ*. *ϱ1 v* = *ϱ2 v*) ⟹ *tsubstT σ ϱ1* = *tsubstT σ ϱ2*
⟨*proof*⟩

lemma *tsubstT-ith*: *i* < *length Ts* ⟹ *map* (λ*T* . *tsubstT T ϱ*) *Ts* ! *i* = *tsubstT* (*Ts* ! *i*) *ϱ*
⟨*proof*⟩

lemma *tsubstT-fun-typ-dist*: *tsubstT* (*T* → *T1*) *ϱ* = *tsubstT T ϱ* → *tsubstT T1 ϱ*

⟨*proof*⟩

fun *subst* :: *term* ⇒ (*variable* ⇒ *typ* ⇒ *term*) ⇒ *term* **where**
  subst (*Ct s T*) *ϱ* = *Ct s T*
| *subst* (*Fv v T*) *ϱ* = *ϱ v T*
| *subst* (*Bv i*) - = *Bv i*
| *subst* (*Abs T t*) *ϱ* = *Abs T* (*subst t ϱ*)
| *subst* (*t* $ *u*) *ϱ* = *subst t ϱ* $ *subst u ϱ*

definition *tinst t1 t2* ≡ ∃ *ϱ*. *tsubst t2 ϱ* = *t1*
definition *inst t1 t2* ≡ ∃ *ϱ*. *subst t2 ϱ* = *t1*

fun *SortsT* :: *typ* ⇒ *sort set* **where**
  SortsT (*Tv* - *S*) = {*S*}
| *SortsT* (*Ty* - *Ts*) = (⋃ *T*∈*set Ts* . *SortsT T*)

fun *Sorts* :: *term* ⇒ *sort set* **where**
  Sorts (*Ct* - *T*) = *SortsT T*
| *Sorts* (*Fv* - *T*) = *SortsT T*
| *Sorts* (*Bv* -) = {}
| *Sorts* (*Abs T t*) = *SortsT T* ∪ *Sorts t*
| *Sorts* (*t* $ *u*) = *Sorts t* ∪ *Sorts u*

fun *Types* :: *term* ⇒ *typ set* **where**
  Types (*Ct* - *T*) = {*T*}
| *Types* (*Fv* - *T*) = {*T*}
| *Types* (*Bv* -) = {}
| *Types* (*Abs T t*) = *insert T* (*Types t*)
| *Types* (*t* $ *u*) = *Types t* ∪ *Types u*

abbreviation *tvs-Set S* ≡ ⋃ *s*∈*S* . *tvs s*
abbreviation *tvsT-Set S* ≡ ⋃ *s*∈*S* . *tvsT s*

lemma *finite-SortsT*[*simp*]: *finite* (*SortsT T*)
  ⟨*proof*⟩
lemma *finite-Sorts*[*simp*]: *finite* (*Sorts t*)
  ⟨*proof*⟩
lemma *finite-Types*[*simp*]: *finite* (*Types t*)
  ⟨*proof*⟩
lemma *finite-tvsT*[*simp*]: *finite* (*tvsT T*)
  ⟨*proof*⟩
lemma *no-tvsT-imp-tsubsT-unchanged*: *tvsT T* = {} ⟹ *tsubstT T ϱ* = *T*
  ⟨*proof*⟩
lemma *finite-fv*[*simp*]: *finite* (*fv t*)
  ⟨*proof*⟩
lemma *finite-tvs*[*simp*]: *finite* (*tvs t*)
  ⟨*proof*⟩

lemma *finite-FV*: *finite S* $\Longrightarrow$ *finite* (*FV S*)
⟨*proof*⟩
lemma *finite-tvs-Set*: *finite S* $\Longrightarrow$ *finite* (*tvs-Set S*)
⟨*proof*⟩
lemma *finite-tvsT-Set*: *finite S* $\Longrightarrow$ *finite* (*tvsT-Set S*)
⟨*proof*⟩

lemma *no-tvs-imp-tsubst-unchanged*: *tvs t* = {} $\Longrightarrow$ *tsubst t* ϱ = *t*
⟨*proof*⟩
lemma *no-fv-imp-subst-unchanged*: *fv t* = {} $\Longrightarrow$ *subst t* ϱ = *t*
⟨*proof*⟩

Functional(also executable) version of *has-typ*

fun *typ-of1* :: *typ list* ⇒ *term* ⇒ *typ option* **where**
typ-of1 - (*Ct* - *T*) = *Some T*
| *typ-of1 Ts* (*Bv i*) = (*if i* < *length Ts then Some* (*nth Ts i*) *else None*)
| *typ-of1* - (*Fv* - *T*) = *Some T*
| *typ-of1 Ts* (*Abs T body*) = *Option.bind* (*typ-of1* (*T*#*Ts*) *body*) (λ*x*. *Some* (*T* → *x*))
| *typ-of1 Ts* (*t* $ *u*) = *Option.bind* (*typ-of1 Ts u*) (λ*U*. *Option.bind* (*typ-of1 Ts t*) (λ*T*.
case T of
Ty fun [*T1*,*T2*] ⇒ *if fun* = *STR* ''*fun*'' *then*
if T1=*U then Some T2 else None*
else None
| - ⇒ *None*
))

For historic reasons a lot of proofs/definitions are still in terms of *typ-of1* instead of *has-typ1*

lemma *has-typ1-weaken-Ts*: *has-typ1 Ts t rT* $\Longrightarrow$ *has-typ1* (*Ts*@[*T*]) *t rT*
⟨*proof*⟩ **thm** *less-Suc-eq nth-butlast*

lemma *has-typ1-imp-typ-of1*: *has-typ1 Ts t ty* $\Longrightarrow$ *typ-of1 Ts t* = *Some ty*
⟨*proof*⟩

lemma *typ-of1-imp-has-typ1*: *typ-of1 Ts t* = *Some ty* $\Longrightarrow$ *has-typ1 Ts t ty*
⟨*proof*⟩

corollary *has-typ1-iff-typ-of1*[*iff*]: *has-typ1 Ts t ty* $\longleftrightarrow$ *typ-of1 Ts t* = *Some ty*
⟨*proof*⟩
corollary *has-typ-iff-typ-of*[*iff*]: *has-typ t ty* $\longleftrightarrow$ *typ-of t* = *Some ty*
⟨*proof*⟩

corollary *typ-of-imp-has-typ*: *typ-of t* = *Some ty* $\Longrightarrow$ *has-typ t ty*
⟨*proof*⟩

lemma *typ-of1-weaken-Ts*: *typ-of1 Ts t* = *Some ty* $\Longrightarrow$ *typ-of1* (*Ts*@[*T*]) *t* = *Some ty*

⟨*proof*⟩

lemma *typ-of1-weaken*:
assumes *typ-of1 Ts t* = *Some T*
shows *typ-of1* (*Ts*@*Ts'*) *t* = *Some T*
⟨*proof*⟩

lemma *has-typ1-tsubst*:
has-typ1 Ts t T ⟹ *has-typ1* (*map* (λ*T*. *tsubstT T* ϱ) *Ts*) (*tsubst t* ϱ) (*tsubstT T* ϱ)
⟨*proof*⟩

corollary *has-typ1-unique*:
assumes *has-typ1* τ*s t* τ*1* **and** *has-typ1* τ*s t* τ*2* **shows** τ*1* = τ*2*
⟨*proof*⟩

hide-fact *typ-of-def*

lemma *typ-of-def*: *typ-of t* ≡ *typ-of1* [] *t*
⟨*proof*⟩

Loose bound variables

fun *loose-bvar* :: *term* ⇒ *nat* ⇒ *bool* **where**
loose-bvar (*Bv i*) *k* ⟷ *i* ≥ *k*
| *loose-bvar* (*t* $ *u*) *k* ⟷ *loose-bvar t k* ∨ *loose-bvar u k*
| *loose-bvar* (*Abs* - *t*) *k* = *loose-bvar t* (*k*+*1*)
| *loose-bvar* - - = *False*

fun *loose-bvar1* :: *term* ⇒ *nat* ⇒ *bool* **where**
loose-bvar1 (*Bv i*) *k* ⟷ *i* = *k*
| *loose-bvar1* (*t* $ *u*) *k* ⟷ *loose-bvar1 t k* ∨ *loose-bvar1 u k*
| *loose-bvar1* (*Abs* - *t*) *k* = *loose-bvar1 t* (*k*+*1*)
| *loose-bvar1* - - = *False*

lemma *loose-bvar1-imp-loose-bvar*: *loose-bvar1 t n* ⟹ *loose-bvar t n*
⟨*proof*⟩
lemma *not-loose-bvar-imp-not-loose-bvar1*: ¬ *loose-bvar t n* ⟹ ¬ *loose-bvar1 t n*
⟨*proof*⟩

lemma *loose-bvar-iff-exist-loose-bvar1*: *loose-bvar t lev* ⟷ (∃ *lev'*≥*lev*. *loose-bvar1 t lev'*)
⟨*proof*⟩

definition *is-open t* ≡ *loose-bvar t 0*
abbreviation *is-closed t* ≡ ¬ *is-open t*
definition *is-dependent t* ≡ *loose-bvar1 t 0*

lemma *loose-bvar-Suc*: *loose-bvar t* (*Suc k*) ⟹ *loose-bvar t k*
⟨*proof*⟩
lemma *loose-bvar-leq*: *k*≥*p* ⟹ *loose-bvar t k* ⟹ *loose-bvar t p*
⟨*proof*⟩

lemma *has-typ1-imp-no-loose-bvar*: *has-typ1 Ts t ty* ⟹ ¬ *loose-bvar t* (*length Ts*)
⟨*proof*⟩

corollary *has-typ-imp-closed*: *has-typ t ty* ⟹ ¬ *is-open t*
⟨*proof*⟩

corollary *typ-of-imp-closed*: *typ-of t* = *Some ty* ⟹ ¬ *is-open t*
⟨*proof*⟩

Subterms

fun *exists-subterm* :: (*term* ⇒ *bool*) ⇒ *term* ⇒ *bool* **where**
exists-subterm P t ⟷ *P t* ∨ (*case t of*
(*t* $ *u*) ⇒ *exists-subterm P t* ∨ *exists-subterm P u*
| *Abs ty body* ⇒ *exists-subterm P body*
| - ⇒ *False*)

fun *exists-subterm′* :: (*term* ⇒ *bool*) ⇒ *term* ⇒ *bool* **where**
exists-subterm′ P (*t* $ *u*) ⟷ *P* (*t* $ *u*) ∨ *exists-subterm′ P t* ∨ *exists-subterm′ P u*
| *exists-subterm′ P* (*Abs ty body*) ⟷ *P* (*Abs ty body*) ∨ *exists-subterm′ P body*
| *exists-subterm′ P t* ⟷ *P t*

lemma *exists-subterm-iff-exists-subterm′*: *exists-subterm P t* ⟷ *exists-subterm′ P t*
⟨*proof*⟩
lemma *exists-subterm* (λ*t*. *t*=*Fv idx T*) *t* ⟷ (*idx*, *T*) ∈ *fv t*
⟨*proof*⟩

abbreviation *occs t u* ≡ *exists-subterm* (λ*s*. *t* = *s*) *u*

lemma *occs-Fv-eq-elem-fv*: *occs* (*Fv v S*) *t* ⟷ (*v*, *S*) ∈ *fv t*
⟨*proof*⟩

lemma *bind-fv2-unchanged*:
¬*loose-bvar tm lev* ⟹ *bind-fv2 v lev tm* = *tm* ⟹ *v* ∉ *fv tm*
⟨*proof*⟩
lemma *bind-fv2-unchanged′*:
¬*loose-bvar tm lev* ⟹ *bind-fv2 v lev tm* = *tm* ⟹ ¬ *occs* (*case-prod Fv v*) *tm*
⟨*proof*⟩

lemma *bind-fv2-changed*:
bind-fv2 v lev tm ≠ *tm* ⟹ *v* ∈ *fv tm*

⟨*proof*⟩
lemma *bind-fv2-changed'*:
bind-fv2 v lev tm ≠ *tm* ⟹ *occs* (*case-prod Fv v*) *tm*
⟨*proof*⟩

corollary *bind-fv-changed*: *bind-fv v tm* ≠ *tm* ⟹ *v* ∈ *fv tm*
⟨*proof*⟩
corollary *bind-fv-changed'*: *bind-fv v tm* ≠ *tm* ⟹ *occs* (*case-prod Fv v*) *tm*
⟨*proof*⟩

corollary *bind-fv-unchanged*: (*x*,τ) ∉ *fv t* ⟹ *bind-fv* (*x*,τ) *t* = *t*
⟨*proof*⟩

inductive-cases *has-typ1-app-elim*: *has-typ1 Ts* (*t* $ *u*) *R*
lemma *has-typ1-arg-typ*: *has-typ1 Ts* (*t* $ *u*) *R* ⟹ *has-typ1 Ts u U* ⟹ *has-typ1 Ts t* (*U* → *R*)
⟨*proof*⟩

lemma *has-typ1-fun-typ*: *has-typ1 Ts* (*t* $ *u*) *R* ⟹ *has-typ1 Ts t* (*U* → *R*) ⟹ *has-typ1 Ts u U*
⟨*proof*⟩

lemma *typ-of1-arg-typ*:
typ-of1 Ts (*t* $ *u*) = *Some R* ⟹ *typ-of1 Ts u* = *Some U* ⟹ *typ-of1 Ts t* = *Some* (*U* → *R*)
⟨*proof*⟩

corollary *typ-of-arg*: *typ-of* (*t*$*u*) = *Some R* ⟹ *typ-of u* = *Some T* ⟹ *typ-of t* = *Some* (*T* → *R*)
⟨*proof*⟩

lemma *typ-of1-fun-typ*:
typ-of1 Ts (*t* $ *u*) = *Some R* ⟹ *typ-of1 Ts t* = *Some* (*U* → *R*) ⟹ *typ-of1 Ts u* = *Some U*
⟨*proof*⟩

corollary *typ-of-fun*: *typ-of* (*t*$*u*) = *Some R* ⟹ *typ-of t* = *Some* (*U* → *R*) ⟹ *typ-of u* = *Some U*
⟨*proof*⟩

lemma *typ-of-eta-expand*: *typ-of f* = *Some* (τ → τ′) ⟹ *typ-of* (*Abs* τ (*f* $ *Bv 0*)) = *Some* (τ → τ′)
⟨*proof*⟩

lemma *bind-fv2-preserves-type*:
assumes *typ-of1 Ts t* = *Some ty*
shows *typ-of1* (*Ts*@[*T*]) (*bind-fv2* (*v*, *T*) (*length Ts*) *t*) = *Some ty*
⟨*proof*⟩

lemma *typ-of-Abs-bind-fv*:
assumes *typ-of A = Some ty*
shows *typ-of (Abs bT (bind-fv (v, bT) A)) = Some (bT → ty)*
⟨*proof*⟩

corollary *typ-of-Abs-fv*:
assumes *typ-of A = Some ty*
shows *typ-of (Abs-fv v bT A) = Some (bT → ty)*
⟨*proof*⟩

lemma *typ-of-mk-all*:
assumes *typ-of A = Some propT*
shows *typ-of (mk-all x ty A) = Some propT*
⟨*proof*⟩

fun *incr-bv* :: *nat* ⇒ *nat* ⇒ *term* ⇒ *term* **where**
incr-bv inc n (Bv i) = (if i ≥ n then Bv (i+inc) else Bv i)
| *incr-bv inc n (Abs T body) = Abs T (incr-bv inc (n+1) body)*
| *incr-bv inc n (App f t) = App (incr-bv inc n f) (incr-bv inc n t)*
| *incr-bv - - u = u*

lemma *lift-def*: *lift t n = incr-bv 1 n t*
⟨*proof*⟩

declare *lift.simps*[*simp del*]
declare *lift-def*[*simp*]

definition *incr-boundvars inc t = incr-bv inc 0 t*

fun *decr* :: *nat* ⇒ *term* ⇒ *term* **where**
decr lev (Bv i) = (if i ≥ lev then Bv (i − 1) else Bv i)
| *decr lev (Abs T t) = Abs T (decr (lev + 1) t)*
| *decr lev (t $ u) = (decr lev t $ decr lev u)*
| *decr - t = t*

lemma *incr-bv-0*[*simp*]: *incr-bv 0 lev t = t*
⟨*proof*⟩

lemma *loose-bvar-incr-bvar*: *loose-bvar t lev ⟷ loose-bvar (incr-bv inc lev t) (lev+inc)*
⟨*proof*⟩

lemma *no-loose-bvar-no-incr*[*simp*]: ¬ *loose-bvar t lev* ⟹ *incr-bv inc lev t = t*
⟨*proof*⟩

lemma *is-close-no-incr-boundvars*[*simp*]: *is-closed t* ⟹ *incr-boundvars inc t = t*
⟨*proof*⟩

lemma *fv-incr-bv* [*simp*]: *fv* (*incr-bv inc lev t*) = *fv t*
⟨*proof*⟩
lemma *fv-incr-boundvars* [*simp*]: *fv* (*incr-boundvars inc t*) = *fv t*
⟨*proof*⟩

lemma *loose-bvar-decr*: ¬ *loose-bvar t k* ⟹ ¬ *loose-bvar* (*decr k t*) *k*
⟨*proof*⟩
lemma *loose-bvar-decr-unchanged*[*simp*]: ¬ *loose-bvar t k* ⟹ *decr k t* = *t*
⟨*proof*⟩
lemma *is-closed-decr-unchanged*[*simp*]: *is-closed t* ⟹ *decr 0 t* = *t*
⟨*proof*⟩

fun *subst-bv1* :: *term* ⇒ *nat* ⇒ *term* ⇒ *term* **where**
subst-bv1 (*Bv i*) *lev u* = (*if i* < *lev then Bv i*
else if i = *lev then* (*incr-boundvars lev u*)
else (*Bv* (*i* − *1*)))
| *subst-bv1* (*Abs T body*) *lev u* = *Abs T* (*subst-bv1 body* (*lev* + *1*) *u*)
| *subst-bv1* (*f* $ *t*) *lev u* = *subst-bv1 f lev u* $ *subst-bv1 t lev u*
| *subst-bv1 t - - = t*

lemma *incr-bv-combine*: *incr-bv m k* (*incr-bv n k s*) = *incr-bv* (*m*+*n*) *k s*
⟨*proof*⟩

lemma *substn-subst-n* : *subst-bv1 t n s* = *subst-bv2 t n* (*incr-bv n 0 s*)
⟨*proof*⟩

theorem *substn-subst-0*: *subst-bv1 t 0 s* = *subst-bv2 t 0 s*
⟨*proof*⟩

corollary *substn-subst-0′*: *subst-bv s t* = *subst-bv2 t 0 s*
⟨*proof*⟩

lemma *subst-bv2-eq* [*simp*]: *subst-bv2* (*Bv k*) *k u* = *u*
⟨*proof*⟩

lemma *subst-bv2-gt* [*simp*]: *i* < *j* ⟹ *subst-bv2* (*Bv j*) *i u* = *Bv* (*j* − *1*)
⟨*proof*⟩

lemma *subst-bv2-subst-lt* [*simp*]: *j* < *i* ⟹ *subst-bv2* (*Bv j*) *i u* = *Bv j*
⟨*proof*⟩

lemma *lift-lift*:
i < *k* + *1* ⟹ *lift* (*lift t i*) (*Suc k*) = *lift* (*lift t k*) *i*
⟨*proof*⟩

lemma *lift-subst* [*simp*]:
j < *i* + *1* ⟹ *lift* (*subst-bv2 t j s*) *i* = *subst-bv2* (*lift t* (*i* + *1*)) *j* (*lift s i*)
⟨*proof*⟩

lemma *lift-subst-bv2-subst-lt*:
$i < j + 1 \Longrightarrow lift\ (subst\text{-}bv2\ t\ j\ s)\ i = subst\text{-}bv2\ (lift\ t\ i)\ (j + 1)\ (lift\ s\ i)$
⟨*proof*⟩

lemma *subst-bv2-lift* [*simp*]:
$subst\text{-}bv2\ (lift\ t\ k)\ k\ s = t$
⟨*proof*⟩

lemma *subst-bv2-subst-bv2*:
$i < j + 1 \Longrightarrow subst\text{-}bv2\ (subst\text{-}bv2\ t\ (Suc\ j)\ (lift\ v\ i))\ i\ (subst\text{-}bv2\ u\ j\ v)$
$= subst\text{-}bv2\ (subst\text{-}bv2\ t\ i\ u)\ j\ v$
⟨*proof*⟩

hide-fact (**open**) *subst-bv-def*
lemma *subst-bv-def*: $subst\text{-}bv\ u\ t \equiv subst\text{-}bv1\ t\ 0\ u$
⟨*proof*⟩

fun *subst-bvs1* :: $term \Rightarrow nat \Rightarrow term\ list \Rightarrow term$ **where**
$subst\text{-}bvs1\ (Bv\ n)\ lev\ args = (if\ n < lev$
then $Bv\ n$
else if $n - lev < length\ args$
then $incr\text{-}boundvars\ lev\ (nth\ args\ (n-lev))$
else $Bv\ (n - length\ args))$
| $subst\text{-}bvs1\ (Abs\ T\ body)\ lev\ args = Abs\ T\ (subst\text{-}bvs1\ body\ (lev+1)\ args)$
| $subst\text{-}bvs1\ (f\ \$\ t)\ lev\ args = subst\text{-}bvs1\ f\ lev\ args\ \$\ subst\text{-}bvs1\ t\ lev\ args$
| $subst\text{-}bvs1\ t\ \text{-}\ \text{-} = t$

definition $subst\text{-}bvs\ args\ t \equiv subst\text{-}bvs1\ t\ 0\ args$

lemma *subst-bvs-App*[*simp*]: $subst\text{-}bvs\ args\ (s\$t) = subst\text{-}bvs\ args\ s\ \$\ subst\text{-}bvs\ args\ t$
⟨*proof*⟩

lemma *subst-bv1-special-case-subst-bvs1*: $subst\text{-}bvs1\ t\ lev\ [x] = subst\text{-}bv1\ t\ lev\ x$
⟨*proof*⟩

lemma *no-loose-bvar-imp-no-subst-bv1*: $\neg loose\text{-}bvar\ t\ lev \Longrightarrow subst\text{-}bv1\ t\ lev\ u = t$
⟨*proof*⟩
lemma *no-loose-bvar-imp-no-subst-bvs1*: $\neg loose\text{-}bvar\ t\ lev \Longrightarrow subst\text{-}bvs1\ t\ lev\ us = t$
⟨*proof*⟩

lemma *subst-bvs1-step*:
assumes $\neg\ loose\text{-}bvar\ t\ lev$
shows $subst\text{-}bvs1\ t\ lev\ (args@[u]) = subst\text{-}bv1\ (subst\text{-}bvs1\ t\ lev\ args)\ lev\ u$

⟨*proof*⟩

corollary *closed-subst-bv-no-change*: *is-closed t* ⟹ *subst-bv u t* = *t*
⟨*proof*⟩

lemma *is-variable-imp-incr-bv-unchanged*: *incr-bv inc lev* (*Fv v T*) = (*Fv v T*)
⟨*proof*⟩
lemma *is-variable-imp-incr-boundvars-unchganged*: *incr-boundvars inc* (*Fv v T*) = (*Fv v T*)
⟨*proof*⟩

lemma *loose-bvar-subst-bv1*:
¬ *loose-bvar* (*subst-bv1 t lev u*) *lev* ⟹ ¬ *loose-bvar t* (*Suc lev*)
⟨*proof*⟩
lemma *is-closed-subst-bv*: *is-closed* (*subst-bv u t*) ⟹ ¬ *loose-bvar t 1*
⟨*proof*⟩

lemma *subst-bv1-bind-fv2*:
assumes ¬ *loose-bvar t lev*
shows *subst-bv1* (*bind-fv2* (*v*, *T*) *lev t*) *lev* (*Fv v T*) = *t*
⟨*proof*⟩

corollary *subst-bv-bind-fv*:
assumes *is-closed t*
shows *subst-bv* (*Fv v T*) (*bind-fv* (*v*, *T*) *t*) = *t*
⟨*proof*⟩

fun *betapply* :: *term* ⇒ *term* ⇒ *term* (**infixl** ‹•› *52*) **where**
betapply (*Abs - t*) *u* = *subst-bv u t*
| *betapply t u* = *t* $ *u*

lemma *betapply-Abs-fv*:
assumes *is-closed t*
shows *betapply* (*Abs-fv v T t*) (*Fv v T*) = *t*
⟨*proof*⟩

lemma *typ-of1-imp-no-loose-bvar*: *typ-of1 Ts t* = *Some ty* ⟹ ¬ *loose-bvar t* (*length Ts*)
⟨*proof*⟩

lemma *typ-of1-subst-bv*:
assumes *typ-of1* (*Ts*@[*uty*]) *f* = *Some fty*
and *typ-of u* = *Some uty*
shows *typ-of1 Ts* (*subst-bv1 f* (*length Ts*) *u*) = *Some fty*
⟨*proof*⟩

lemma *typ-of1-split-App*:
typ-of1 Ts (*t* $ *u*) = *Some ty* ⟹ (∃ *uty* . *typ-of1 Ts t* = *Some* (*uty* → *ty*) ∧ *typ-of1 Ts u* = *Some uty*)

⟨*proof*⟩

corollary *typ-of1-split-App-obtains*:
assumes *typ-of1 Ts* (*t* $ *u*) = *Some ty*
obtains *uty* **where** *typ-of1 Ts t* = *Some* (*uty* → *ty*) *typ-of1 Ts u* = *Some uty*
⟨*proof*⟩

lemma *typ-of1-incr-bv*:
assumes *typ-of1 Ts t* = *Some ty*
and *lev* ≤ *length Ts*
shows *typ-of1* (*take lev Ts* @ *Ts'* @ *drop lev Ts*) (*incr-bv* (*length Ts'*) *lev t*) = *Some ty*
⟨*proof*⟩

corollary *typ-of1-incr-bv-lev0*:
assumes *typ-of1 Ts t* = *Some ty*
shows *typ-of1* (*Ts'* @ *Ts*) (*incr-bv* (*length Ts'*) *0 t*) = *Some ty*
⟨*proof*⟩

lemma *typ-of1-subst-bv-gen*:
assumes *typ-of1* (*Ts'*@[*uty*]@*Ts*) *t* = *Some tty* **and** *typ-of1 Ts u* = *Some uty*
shows *typ-of1* (*Ts'* @ *Ts*) (*subst-bv1 t* (*length Ts'*) *u*) = *Some tty*
⟨*proof*⟩

lemma *typ-of1-subst-bv-gen-depre*:
assumes *typ-of1* (*Ts'*@*Ts*) *f* = *Some* (*fty*)
and *typ-of1* (*Ts*) *u* = *Some uty*
and *last Ts'* = *uty* **and** *Ts'* ≠ []
shows *typ-of1* (*butlast Ts'* @ *Ts*) (*subst-bv1 f* (*length Ts'*−*1*) *u*) = *Some fty*
⟨*proof*⟩

corollary *typ-of1-subst-bv-gen'*:
assumes *typ-of1* (*uty*#*Ts*) *t* = *Some tty*
and *typ-of1 Ts u* = *Some uty*
shows *typ-of1 Ts* (*subst-bv1 t 0 u*) = *Some tty*
⟨*proof*⟩

lemma *typ-of-betapply*:
assumes *typ-of1 Ts* (*Abs uty t*) = *Some* (*uty* → *tty*)
assumes *typ-of1 Ts u* = *Some uty*
shows *typ-of1 Ts* ((*Abs uty t*) · *u*) = *Some tty*
⟨*proof*⟩

lemma *no-Bv-Type-param-irrelevant-typ-of*:
¬*exists-subterm* (λ*x* . *case x of Bv* - ⇒ *True* | - ⇒ *False*) *t*
⟹ *typ-of1 Ts t* = *typ-of1 Ts' t*
⟨*proof*⟩

lemma *typ-of1-drop-extra-bounds*:
  $\neg$*loose-bvar t* (*length Ts*)
  $\Longrightarrow$ *typ-of1* (*Ts*@*rest*) *t* = *typ-of1 Ts t*
  ⟨*proof*⟩

lemma *typ-of-betaply*:
  assumes *typ-of t* = *Some* (*uty* → *tty*) *typ-of u* = *Some uty*
  shows *typ-of* (*t* · *u*) = *Some tty*
⟨*proof*⟩

fun *beta-reducible* :: *term* ⇒ *bool* **where**
  beta-reducible (*App* (*Abs* - -) -) = *True*
| *beta-reducible* (*Abs* - *t*) = *beta-reducible t*
| *beta-reducible* (*App t u*) = (*beta-reducible t* ∨ *beta-reducible u*)
| *beta-reducible* - = *False*

fun *eta-reducible* :: *term* ⇒ *bool* **where**
  eta-reducible (*Abs* - (*t* $ *Bv 0*)) = ($\neg$ *is-dependent t* ∨ *eta-reducible t*)
| *eta-reducible* (*Abs* - *t*) = *eta-reducible t*
| *eta-reducible* (*App t u*) = (*eta-reducible t* ∨ *eta-reducible u*)
| *eta-reducible* - = *False*

lemma $\neg$ *loose-bvar t lev* $\Longrightarrow$ *decr lev t* = *t*
  ⟨*proof*⟩

lemma *decr-incr-bv1*: *decr lev* (*incr-bv 1 lev t*) = *t*
  ⟨*proof*⟩

fun *depth* :: *term* ⇒ *nat* **where**
  depth (*Abs* - *t*) = *depth t* + *1*
| *depth* (*t* $ *u*) = *max* (*depth t*) (*depth u*) *+1*
| *depth t* = *0*

lemma *depth-decr*: *depth* (*decr lev t*) = *depth t*
  ⟨*proof*⟩

lemma *loose-bvar1-decr*: *lev* > *0* $\Longrightarrow$ $\neg$ *loose-bvar1 t* (*Suc lev*) $\Longrightarrow$ $\neg$ *loose-bvar1* (*decr lev t*) *lev*
  ⟨*proof*⟩

lemma *loose-bvar1-decr'*:
  $\neg$ *loose-bvar1 t* (*Suc lev*) $\Longrightarrow$ $\neg$ *loose-bvar1 t lev* $\Longrightarrow$ $\neg$ *loose-bvar1* (*decr lev t*) *lev*
  ⟨*proof*⟩

lemma *eta-reducible-Abs1*: $\neg$ *eta-reducible* (*Abs T* (*t* $ *Bv 0*)) $\Longrightarrow$ $\neg$ *eta-reducible t* ⟨*proof*⟩

lemma *eta-reducible-Abs2*:
  assumes ¬ (∃*f*. *t*=*f* $ *Bv 0*) ¬ *eta-reducible* (*Abs T t*)
  shows ¬ *eta-reducible t*
⟨*proof*⟩

lemma *eta-reducible-Abs*: ¬ *eta-reducible* (*Abs T t*) ⟹ ¬ *eta-reducible t*
  ⟨*proof*⟩

lemma *loose-bvar1-decr''*: *loose-bvar1 t lev* ⟹ *lev* < *lev'*⟹ *loose-bvar1* (*decr lev' t*) *lev*
  ⟨*proof*⟩
lemma *loose-bvar1-decr'''*: *loose-bvar1 t* (*Suc lev*) ⟹ *lev'* ≤ *lev* ⟹ *loose-bvar1* (*decr lev' t*) *lev*
  ⟨*proof*⟩

lemma *loose-bvar1-decr''''*: ¬ *loose-bvar1 t lev'* ⟹ *lev'* ≤ *lev* ⟹ ¬ *loose-bvar1 t* (*Suc lev*)
  ⟹ ¬ *loose-bvar1* (*decr lev' t*) *lev*
  ⟨*proof*⟩

lemma *not-eta-reducible-decr*:
  ¬ *eta-reducible t* ⟹ ¬ *loose-bvar1 t lev* ⟹ ¬ *eta-reducible* (*decr lev t*)
⟨*proof*⟩

function (*sequential*, *domintros*) *eta-norm* :: *term* ⇒ *term* **where**
  eta-norm (*Abs T t*) = (*case eta-norm t of*
    f $ *Bv 0* ⇒ (*if is-dependent f then Abs T* (*f* $ *Bv 0*) *else decr 0* (*eta-norm f*))
  | *body* ⇒ *Abs T body*)
| *eta-norm* (*t* $ *u*) = *eta-norm t* $ *eta-norm u*
| *eta-norm t* = *t*
  ⟨*proof*⟩

lemma *eta-norm-reduces-depth*: *eta-norm-dom t* ⟹ *depth* (*eta-norm t*) <= *depth t*
  ⟨*proof*⟩

termination *eta-norm*
⟨*proof*⟩

lemma *loose-bvar1-eta-norm*: *loose-bvar1 t lev* ⟹ *loose-bvar1* (*eta-norm t*) *lev*
  ⟨*proof*⟩

lemma *loose-bvar1-eta-norm'*: ¬ *loose-bvar1 t lev* ⟹ ¬ *loose-bvar1* (*eta-norm t*) *lev*
⟨*proof*⟩

lemma *not-eta-reducible-eta-norm*: ¬ *eta-reducible* (*eta-norm t*)
⟨*proof*⟩

lemma *not-eta-reducible-imp-eta-norm-no-change*: $\neg$ *eta-reducible t* $\Longrightarrow$ *eta-norm t* = *t*
⟨*proof*⟩

lemma *eta-norm-collapse*: *eta-norm* (*eta-norm t*) = *eta-norm t*
⟨*proof*⟩

lemma *typ-of1-decr*: *typ-of1* (*Ts*@[*T*]@*Ts'*) *t* = *Some ty* $\Longrightarrow$ $\neg$ *loose-bvar1 t* (*length Ts*)
$\Longrightarrow$ *typ-of1* (*Ts*@*Ts'*) (*decr* (*length Ts*) *t*) = *Some ty*
⟨*proof*⟩

lemma *typ-of1-decr-gen*: *typ-of1* (*Ts*@[*T*]@*Ts'*) *t* = *tyo* $\Longrightarrow$ $\neg$ *loose-bvar1 t* (*length Ts*)
$\Longrightarrow$ *typ-of1* (*Ts*@*Ts'*) (*decr* (*length Ts*) *t*) = *tyo*
⟨*proof*⟩

lemma *typ-of1-decr-gen'*: *typ-of1* (*Ts*@*Ts'*) (*decr* (*length Ts*) *t*) = *tyo*$\Longrightarrow$ $\neg$ *loose-bvar1 t* (*length Ts*)
$\Longrightarrow$ *typ-of1* (*Ts*@[*T*]@*Ts'*) *t* = *tyo*
⟨*proof*⟩

lemma *typ-of1-eta-norm*: *typ-of1 Ts t* = *Some ty* $\Longrightarrow$ *typ-of1 Ts* (*eta-norm t*) = *Some ty*
⟨*proof*⟩

corollary *typ-of-eta-norm*: *typ-of t* = *Some ty* $\Longrightarrow$ *typ-of* (*eta-norm t*) = *Some ty*
⟨*proof*⟩

lemma *typ-of-Abs-body-typ*: *typ-of1 Ts* (*Abs T t*) = *Some ty* $\Longrightarrow$ $\exists$ *rty*. *ty* = (*T* $\rightarrow$ *rty*)
⟨*proof*⟩
lemma *typ-of-Abs-body-typ'*: *typ-of1 Ts* (*Abs T t*) = *Some ty*
$\Longrightarrow$ $\exists$ *rty*. *ty* = (*T* $\rightarrow$ *rty*) $\wedge$ *typ-of1* (*T*#*Ts*) *t* = *Some rty*
⟨*proof*⟩

lemma *typ-of-beta-redex-arg*: *typ-of* (*Abs T s* $ *t*) $\neq$ *None* $\Longrightarrow$ *typ-of t* = *Some T*
⟨*proof*⟩

lemma [*partial-function-mono*]: *option.mono-body*
(λ*beta-norm*. *map-option* (*Abs T*) (*beta-norm t*))
⟨*proof*⟩
lemma [*partial-function-mono*]: *option.mono-body*
(λ*beta-norm*.
case beta-norm x of None ⇒ *None*
| *Some* (*Ct list typ*) ⇒
map-option (($) (*Ct list typ*)) (*beta-norm u*)

```
    | Some (Fv p typ) ⇒
        map-option ((⟨$⟩) (Fv p typ)) (beta-norm u)
    | Some (Bv n) ⇒
        map-option ((⟨$⟩) (Bv n)) (beta-norm u)
    | Some (Abs T body) ⇒
        beta-norm (subst-bv u body)
    | Some (term1 $ term2) ⇒
        map-option ((⟨$⟩) (term1 $ term2)) (beta-norm u))
⟨proof⟩


partial-function (option) beta-norm :: term ⇒ term option where
  beta-norm t = (case t of
    (Abs T body) ⇒ map-option (Abs T) (beta-norm body)
  | (Abs T body $ u) ⇒ beta-norm (subst-bv u body)
  | (f $ u) ⇒ (case beta-norm f of
      Some (Abs T body) ⇒ beta-norm (subst-bv u body)
    | Some f' ⇒ map-option (App f') (beta-norm u)
    | None ⇒ None)
  | t ⇒ Some t)

simps-of-case beta-norm-simps [simp, code]: beta-norm.simps

lemma not-beta-reducible-imp-beta-norm-unchanged: ¬ beta-reducible t ⟹ beta-norm
t = Some t
⟨proof⟩

lemma not-beta-reducible-decr: ¬ beta-reducible t ⟹ ¬ beta-reducible (decr n t)
  ⟨proof⟩

lemma ¬ beta-reducible t ⟹ eta-norm t = t' ⟹ ¬ beta-reducible t'
⟨proof⟩

fun is-variable :: term ⇒ bool where
  is-variable (Fv - -) = True
| is-variable - = False

lemma fv-occs: (x,τ) ∈ fv t ⟹ occs (Fv x τ) t
  ⟨proof⟩

lemma fv-iff-occs: (x,τ) ∈ fv t ⟷ occs (Fv x τ) t
  ⟨proof⟩


fun strip-abs :: term ⇒ typ list * term where
  strip-abs (Abs T t) = (let (a', t') = strip-abs t in (T # a', t'))
| strip-abs t = ([], t)
```

fun *strip-abs-body* :: *term* ⇒ *term* **where**
  strip-abs-body (*Abs* - *t*) = *strip-abs-body t*
| *strip-abs-body u* = *u*

fun *strip-abs-vars* :: *term* ⇒ *typ list* **where**
  strip-abs-vars (*Abs T t*) = *T* # *strip-abs-vars t*
| *strip-abs-vars u* = []

fun *strip-qnt-body* :: *name* ⇒ *term* ⇒ *term* **where**
  strip-qnt-body qnt ((*Ct c ty*) $ (*Abs* - *t*)) =
    (*if c*=*qnt then strip-qnt-body qnt t else* (*Ct c ty*))
| *strip-qnt-body* - *t* = *t*

fun *strip-qnt-vars* :: *name* ⇒ *term* ⇒ *typ list* **where**
  strip-qnt-vars qnt (*Ct c* - $ *Abs T t*)= (*if c*=*qnt then T* # *strip-qnt-vars qnt t else* [])
| *strip-qnt-vars qnt t* = []

definition *list-comb* :: *term* ∗ *term list* ⇒ *term* **where** *list-comb* = *case-prod* (*foldl* ($))

definition *list-comb′* :: *term* ⇒ *term list* ⇒ *term* **where** *list-comb′* = *foldl* ($)

lemma *list-comb* (*h*,*t*) = *list-comb′ h t* ⟨*proof*⟩

fun *strip-comb-imp* **where**
  strip-comb-imp (*f*$*t*, *ts*) = *strip-comb-imp* (*f*, *t* # *ts*)
| *strip-comb-imp x* = *x*

definition *strip-comb* :: *term* ⇒ *term* ∗ *term list* **where**
  strip-comb u = *strip-comb-imp* (*u*,[])

fun *head-of* :: *term* ⇒ *term* **where**
  head-of (*f*$*t*) = *head-of f*
| *head-of u* = *u*

lemma *fst-strip-comb-imp-eq-head-of*: *fst* (*strip-comb-imp* (*t*,*ts*)) = *head-of t*
  ⟨*proof*⟩
corollary *fst* (*strip-comb t*) = *head-of t*
  ⟨*proof*⟩

fun *is-app* :: *term* ⇒ *bool* **where**
  is-app (- $ -) = *True*
| *is-app* - = *False*

lemma *not-is-app-imp-strip-com-imp-unchanged*: ¬ *is-app t* ⟹ *strip-comb-imp* (*t*,*ts*) = (*t*,*ts*)
  ⟨*proof*⟩
corollary *not-is-app-imp-strip-com-unchanged*: ¬ *is-app t* ⟹ *strip-comb t* = (*t*,[])

  ⟨*proof*⟩

lemma *list-comb-fuse*: *list-comb* (*list-comb* (*t*,*ts*), *ss*) = *list-comb* (*t*,*ts*@*ss*)
  ⟨*proof*⟩

fun *add-size-term* :: *term* ⇒ *int* ⇒ *int* **where**
  add-size-term (*t* $ *u*) *n* = *add-size-term t* (*add-size-term u n*)
| *add-size-term* (*Abs* - *t*) *n* = *add-size-term t* (*n* + *1*)
| *add-size-term* - *n* = *n* + *1*

definition *size-of-term t* = *add-size-term t 0*

fun *add-size-type* :: *typ* ⇒ *int* ⇒ *int* **where**
  add-size-type (*Ty* - *tys*) *n* = *fold add-size-type tys* (*n* + *1*)
| *add-size-type* - *n* = *n* + *1*

definition *size-of-type ty* = *add-size-type ty 0*

fun *map-types* :: (*typ* ⇒ *typ*) ⇒ *term* ⇒ *term* **where**
  map-types f (*Ct a T*) = *Ct a* (*f T*)
| *map-types f* (*Fv v T*) = *Fv v* (*f T*)
| *map-types f* (*Bv i*) = *Bv i*
| *map-types f* (*Abs T t*) = *Abs* (*f T*) (*map-types f t*)
| *map-types f* (*t* $ *u*) = *map-types f t* $ *map-types f u*

fun *map-atyps* :: (*typ* ⇒ *typ*) ⇒ *typ* ⇒ *typ* **where**
  map-atyps f (*Ty a Ts*) = *Ty a* (*map* (*map-atyps f*) *Ts*)
| *map-atyps f T* = *f T*

lemma *map-atyps id ty* = *ty*
  ⟨*proof*⟩

fun *map-aterms* :: (*term* ⇒ *term*) ⇒ *term* ⇒ *term* **where**
  map-aterms f (*t* $ *u*) = *map-aterms f t* $ *map-aterms f u*
| *map-aterms f* (*Abs T t*) = *Abs T* (*map-aterms f t*)
| *map-aterms f t* = *f t*

lemma *map-aterms id t* = *t*

⟨*proof*⟩

definition *map-type-tvar f* = *map-atyps* (λ*x* . *case x of Tv iname s* ⇒ *f iname s* | *T* ⇒ *T*)

lemma *map-types-id*[*simp*]: *map-types id t* = *t*
⟨*proof*⟩
lemma *map-types-id′*[*simp*]: *map-types* (λ*a* . *a*) *t* = *t*
⟨*proof*⟩

fun *fold-atyps* :: (*typ* ⇒ *′a* ⇒ *′a*) ⇒ *typ* ⇒ *′a* ⇒ *′a* **where**
fold-atyps f (*Ty - Ts*) *s* = *fold* (*fold-atyps f*) *Ts s*
| *fold-atyps f T s* = *f T s*

definition *fold-atyps-sorts f* =
fold-atyps (λ*x* . *case x of Tv vn S* ⇒ *f* (*Tv vn S*) *S*)

fun *fold-aterms* :: (*term* ⇒ *′a* ⇒ *′a*) ⇒ *term* ⇒ *′a* ⇒ *′a* **where**
fold-aterms f (*t* $ *u*) *s* = *fold-aterms f u* (*fold-aterms f t s*)
| *fold-aterms f* (*Abs - t*) *s* = *fold-aterms f t s*
| *fold-aterms f a s* = *f a s*

fun *fold-term-types* :: (*term* ⇒ *typ* ⇒ *′a* ⇒ *′a*) ⇒ *term* ⇒ *′a* ⇒ *′a* **where**
fold-term-types f (*Ct n T*) *s* = *f* (*Ct n T*) *T s*
| *fold-term-types f* (*Fv idn T*) *s* = *f* (*Fv idn T*) *T s*
| *fold-term-types f* (*Bv -*) *s* = *s*
| *fold-term-types f* (*Abs T b*) *s* = *fold-term-types f b* (*f* (*Abs T b*) *T s*)
| *fold-term-types f* (*t* $ *u*) *s* = *fold-term-types f u* (*fold-term-types f t s*)

definition *fold-types f* = *fold-term-types* (λ*x* . *f*)

fun *replace-types* :: *term* ⇒ *typ list* ⇒ *term* × *typ list* **where**
replace-types (*Ct c -*) (*T* # *Ts*) = (*Ct c T*, *Ts*)
| *replace-types* (*Fv xi -*) (*T* # *Ts*) = (*Fv xi T*, *Ts*)
| *replace-types* (*Bv i*) *Ts* = (*Bv i*, *Ts*)
| *replace-types* (*Abs - b*) (*T* # *Ts*) =
(*let* (*b′*, *Ts′*) = *replace-types b Ts*
in (*Abs T b′*, *Ts′*))
| *replace-types* (*t* $ *u*) *Ts* =
(*let*
(*t′*, *Ts′*) = *replace-types t Ts in*
(*let* (*u′*, *Ts′′*) = *replace-types u Ts*
in (*t′* $ *u′*, *Ts′′*)))

definition *add-tvar-namesT′* = *fold-atyps* (λ*x l* . *case x of Tv xi - => List.insert xi l* | *- => l*)

definition *add-tvar-names′* = *fold-types add-tvar-namesT′*
definition *add-tvarsT′* = *fold-atyps* (λ*x l* . *case x of Tv idn s* => *List.insert* (*idn*,*s*) *l* | - => *l*)
definition *add-tvars′* = *fold-types add-tvarsT′*
definition *add-vars′* = *fold-aterms* (λ*x l* . *case x of Fv idn s* => *List.insert* (*idn*,*s*) *l* | - => *l*)
definition *add-var-names′* = *fold-aterms* (λ*x l* . *case x of Fv xi* - => *List.insert xi l* | - => *l*)

definition *add-const-names′* = *fold-aterms* (λ*x l* . *case x of Ct c* - => *List.insert c l* | - => *l*)
definition *add-consts′* = *fold-aterms* (λ*x l* . *case x of Ct n s* => *List.insert* (*n*,*s*) *l* | - => *l*)

definition *add-tvar-namesT* = *fold-atyps* (λ*x* . *case x of Tv xi* - => *insert xi* | - => *id*)
definition *add-tvar-names* = *fold-types add-tvar-namesT*
definition *add-tvarsT* = *fold-atyps* (λ*x* . *case x of Tv idn s* => *insert* (*idn*,*s*) | - => *id*)
definition *add-tvars* = *fold-types add-tvarsT*
definition *add-var-names* = *fold-aterms* (λ*x* . *case x of Fv xi* - => *insert xi* | - => *id*)
definition *add-vars* = *fold-aterms* (λ*x* . *case x of Fv idn s* => *insert* (*idn*,*s*) | - => *id*)

definition *add-const-names* = *fold-aterms* (λ*x* . *case x of Ct c* - => *insert c* | - => *id*)
definition *add-consts* = *fold-aterms* (λ*x* . *case x of Ct n s* => *insert* (*n*,*s*) | - => *id*)

lemma *add-tvarsT′-tvsT-pre*[*simp*]: *set* (*add-tvarsT′ T acc*) = *set acc* ∪ *tvsT T*
⟨*proof*⟩

lemma *add-tvars′-tvs-pre*[*simp*]: *set* (*add-tvars′ t acc*) = *set acc* ∪ *tvs t*
⟨*proof*⟩

lemma *add-tvarsT T acc* = *acc* ∪ *tvsT T*
⟨*proof*⟩

lemma *add-vars′-fv-pre*: *set* (*add-vars′ t acc*) = *set acc* ∪ *fv t*
⟨*proof*⟩
corollary *add-vars′-fv*: *set* (*add-vars′ t* []) = *fv t*
⟨*proof*⟩

fun *strip-all-body* :: *term* ⇒ *term* **where**
strip-all-body (*Ct all S* $ *Abs T t*) = (*if all*= *STR* ''*Pure.all*'' ∧ *S*=(*T*→*propT*)→*propT*
then strip-all-body t else (*Ct all S* $ *Abs T t*))
| *strip-all-body t* = *t*

fun *strip-all-vars* :: *term* ⇒ *typ list* **where**
strip-all-vars (*Ct all S* $ *Abs T t*) = (*if all*= *STR* ''*Pure.all*'' ∧ *S*=(*T*→*propT*)→*propT*

then T # *strip-all-vars t else* [])
| *strip-all-vars t* = []

fun *strip-all-single-body* :: *term* ⇒ *term* **where**
strip-all-single-body (*Ct all S* $ *Abs T t*) = (*if all*= *STR* ''*Pure.all*'' ∧ *S*=(*T*→*propT*)→*propT*

then t else (*Ct all S* $ *Abs T t*))
| *strip-all-single-body t* = *t*

fun *strip-all-single-var* :: *term* ⇒ *typ option* **where**
strip-all-single-var (*Ct all S* $ *Abs T t*) = (*if all*= *STR* ''*Pure.all*'' ∧ *S*=(*T*→*propT*)→*propT*
then Some T else None)
| *strip-all-single-var t* = *None*

fun *strip-all-multiple-body* :: *nat* ⇒ *term* ⇒ *term* **where**
strip-all-multiple-body 0 t = *t*
| *strip-all-multiple-body* (*Suc n*) (*Ct all S* $ *Abs T t*) = (*if all*= *STR* ''*Pure.all*'' ∧
S=(*T*→*propT*)→*propT*
then strip-all-multiple-body n t else (*Ct all S* $ *Abs T t*))
| *strip-all-multiple-body - t* = *t*

fun *strip-all-multiple-vars* :: *nat* ⇒ *term* ⇒ *typ list* **where**
strip-all-multiple-vars 0 - = []
| *strip-all-multiple-vars* (*Suc n*) (*Ct all S* $ *Abs T t*) = (*if all*= *STR* ''*Pure.all*'' ∧
S=(*T*→*propT*)→*propT*
then T # *strip-all-multiple-vars n t else* [])
| *strip-all-multiple-vars - t* = []

lemma *strip-all-vars-strip-all-multiple-vars*:
n≥*length* (*strip-all-vars t*) ⟹ *strip-all-multiple-vars n t* = *strip-all-vars t*
⟨*proof*⟩
lemma *n*≥*length* (*strip-all-vars t*) ⟹ *strip-all-multiple-body n t* = *strip-all-body*
t
⟨*proof*⟩

lemma *length-strip-all-multiple-vars*: *length* (*strip-all-multiple-vars n t*) ≤ *n*
⟨*proof*⟩

lemma *prefix-strip-all-multiple-vars*: *prefix* (*strip-all-multiple-vars n t*) (*strip-all-vars*

t)
⟨*proof*⟩

definition *mk-all-list l t* = *fold* (λ(*n*,*T*) *acc* . *mk-all n T acc*) *l t*

lemma *mk-all-list-empty*[*simp*]: *mk-all-list* [] *t* = *t* ⟨*proof*⟩

fun *is-all* :: *term* ⇒ *bool* **where**
is-all (*Ct all S* $ *Abs T t*) = (*all*= *STR* ''*Pure.all*'' ∧ *S*=(*T*→*propT*)→*propT*)
| *is-all* - = *False*

lemma *strip-all-single-var-is-all*: *strip-all-single-var t* ≠ *None* ⟷ *is-all t*
⟨*proof*⟩

lemma *is-all t* ⟹ *hd* (*strip-all-vars t*) = *the* (*strip-all-single-var t*)
⟨*proof*⟩

lemma *strip-all-body-single-simp*[*simp*]: *strip-all-body* (*strip-all-single-body t*) = *strip-all-body t*
⟨*proof*⟩
lemma *strip-all-body-single-simp'*[*simp*]: *strip-all-single-body* (*strip-all-body t*) = *strip-all-body t*
⟨*proof*⟩

lemma *strip-all-vars-step*:
strip-all-single-var t = *Some T* ⟹ *T* # *strip-all-vars* (*strip-all-single-body t*) = *strip-all-vars t*
⟨*proof*⟩

lemma *is-all-iff-strip-all-vars-not-empty*: *is-all t* ⟷ *strip-all-vars t* ≠ []
⟨*proof*⟩

lemma *strip-all-vars-bind-fv*:
strip-all-vars (*bind-fv2 v lev t*) = (*strip-all-vars t*)
⟨*proof*⟩

lemma *strip-all-vars-mk-all*[*simp*]: *strip-all-vars* (*mk-all s ty t*) = *ty* # *strip-all-vars t*
⟨*proof*⟩

lemma *strip-all-vars-mk-all-list*:
¬*is-all t* ⟹ *strip-all-vars* (*mk-all-list l t*) = *rev* (*map snd l*)
⟨*proof*⟩

lemma *subst-bv-no-loose-unchanged*:
assumes ⋀*x* . *x* ≥ *lev* ⟹ ¬ *loose-bvar1 t x*
assumes *is-variable v*

shows (*subst-bv1 t lev v*) = *t*
⟨*proof*⟩

lemma *bind-fv2-no-occs-unchanged*:
assumes ¬ *occs* (*case-prod Fv v*) *t*
shows (*bind-fv2 v lev t*) = *t*
⟨*proof*⟩

lemma *bind-fv2-subst-bv1-cancel*:
assumes ⋀*x* . *x* > *lev* ⟹ ¬ *loose-bvar1 t x*
assumes ¬ *occs* (*case-prod Fv v*) *t*
shows *bind-fv2 v lev* (*subst-bv1 t lev* (*case-prod Fv v*)) = *t*
⟨*proof*⟩

lemma *bind-fv-subst-bv-cancel*:
assumes ⋀*x* . *x* > *0* ⟹ ¬ *loose-bvar1 t x*
assumes ¬ *occs* (*case-prod Fv v*) *t*
shows *bind-fv v* (*subst-bv* (*case-prod Fv v*) *t*) = *t*
⟨*proof*⟩

lemma *not-loose-bvar-imp-not-loose-bvar1-all-greater*: ¬ *loose-bvar t lev* ⟹ *x*>*lev*
⟹ ¬ *loose-bvar1 t x*
⟨*proof*⟩

lemma *mk-all′-subst-bv-strip-all-single-body-cancel*:
assumes *strip-all-single-var t* = *Some T*
assumes *is-closed t*
assumes (*name*, *T*) ∉ *fv t*
shows *mk-all name T* (*subst-bv* (*Fv name T*) (*strip-all-single-body t*)) = *t*
⟨*proof*⟩

lemma *not-is-all-imp-strip-all-body-unchanged*: ¬ *is-all t* ⟹ *strip-all-body t* = *t*
⟨*proof*⟩

lemma *no-loose-bvar-imp-no-subst-bvs*: *is-closed t* ⟹ *subst-bvs* [] *t* = *t*
⟨*proof*⟩

lemma *is-closed* (*Abs T t*) ⟹ ¬ *loose-bvar t 1* ⟨*proof*⟩

lemma *bind-fv2-Fv-fv*[*simp*]: *fv* (*bind-fv2* (*x*, *τ*) *lev t*) = *fv t* − {(*x*,*τ*)}
⟨*proof*⟩

corollary *mk-all-fv-unchanged*: *fv* (*mk-all x τ B*) = *fv B* − {(*x*,*τ*)}
⟨*proof*⟩

lemma *mk-all-list-fv-unchanged*: *fv* (*mk-all-list l B*) = *fv B* − *set l*
⟨*proof*⟩

abbreviation *forall-intro-vars t Hs* ≡ *mk-all-list*
(*minus-list-set* (*add-vars′ t* []) (*fold* (*add-vars′*) *Hs* [])) *t*

end

4 Sorts

theory *Sorts*
imports *Term*
begin

definition [*simp*]: *empty-osig* = ({}, *Map.empty*)

definition *sort-les cs s1 s2* = (*sort-leq cs s1 s2* ∧ ¬ *sort-leq cs s2 s1*)
definition *sort-eqv cs s1 s2* = (*sort-leq cs s1 s2* ∧ *sort-leq cs s2 s1*)

lemmas *class-defs* = *class-leq-def class-les-def class-ex-def*
lemmas *sort-defs* = *sort-leq-def sort-les-def sort-eqv-def sort-ex-def*

lemma *sort-ex-class-ex*: *sort-ex cs S* ≡ ∀ *c* ∈ *S*. *class-ex cs c*
⟨*proof*⟩

locale *wf-subclass-loc* =
fixes *cs* :: *class rel*
assumes *wf*[*simp*]: *wf-subclass cs*
begin

lemma *class-les-irrefl*: ¬ *class-les cs c c*
⟨*proof*⟩
lemma *class-les-trans*: *class-les cs x y* ⟹ *class-les cs y z* ⟹ *class-les cs x z*
⟨*proof*⟩

lemma *class-leq-refl*[*iff*]: *class-ex cs c* ⟹ *class-leq cs c c*
⟨*proof*⟩
lemma *class-leq-trans*: *class-leq cs x y* ⟹ *class-leq cs y z* ⟹ *class-leq cs x z*
⟨*proof*⟩
lemma *class-leq-antisym*: *class-leq cs c1 c2* ⟹ *class-leq cs c2 c1* ⟹ *c1*=*c2*
⟨*proof*⟩

lemma *sort-leq-refl*[*iff*]: *sort-ex cs s* ⟹ *sort-leq cs s s*
⟨*proof*⟩
lemma *sort-leq-trans*: *sort-leq cs x y* ⟹ *sort-leq cs y z* ⟹ *sort-leq cs x z*
⟨*proof*⟩
lemma *sort-leq-ex*: *sort-leq cs s1 s2* ⟹ *sort-ex cs s2*
⟨*proof*⟩

lemma *sort-leq-minimize*:
sort-leq cs s1 s2 $\Longrightarrow \exists s1'.\ (\forall c1 \in s1'.\ \exists c2 \in s2.\ \textit{class-leq cs c1 c2}) \wedge \textit{sort-leq cs s1' s2}$
⟨*proof*⟩

lemma *sort-ex cs s2* $\Longrightarrow$ *s1* $\subseteq$ *s2* $\Longrightarrow$ *sort-ex cs s1*
⟨*proof*⟩

lemma *superset-imp-sort-leq*: *sort-ex cs s2* $\Longrightarrow$ *s1* $\supseteq$ *s2* $\Longrightarrow$ *sort-leq cs s1 s2*
⟨*proof*⟩
lemma *full-sort-top*: *sort-ex cs s* $\Longrightarrow$ *sort-leq cs s full-sort*
⟨*proof*⟩

lemma *sort-les-trans*: *sort-les cs x y* $\Longrightarrow$ *sort-les cs y z* $\Longrightarrow$ *sort-les cs x z*
⟨*proof*⟩

lemma *sort-eqvI*: *sort-leq cs s1 s2* $\Longrightarrow$ *sort-leq cs s2 s1* $\Longrightarrow$ *sort-eqv cs s1 s2*
⟨*proof*⟩
lemma *sort-eqv-refl*: *sort-ex cs s* $\Longrightarrow$ *sort-eqv cs s s*
⟨*proof*⟩
lemma *sort-eqv-trans*: *sort-eqv cs x y* $\Longrightarrow$ *sort-eqv cs y z* $\Longrightarrow$ *sort-eqv cs x z*
⟨*proof*⟩
lemma *sort-eqv-sym*: *sort-eqv cs x y* $\Longrightarrow$ *sort-eqv cs y x*
⟨*proof*⟩

lemma *normalize-sort-empty*[*simp*]: *normalize-sort cs full-sort* = *full-sort*
⟨*proof*⟩
lemma *normalize-sort-normalize-sort*[*simp*]:
normalize-sort cs (*normalize-sort cs s*) = *normalize-sort cs s*
⟨*proof*⟩

lemma *sort-ex-norm-sort*: *sort-ex cs s* $\Longrightarrow$ *sort-ex cs* (*normalize-sort cs s*)
⟨*proof*⟩

lemma *normalized-sort-subset*: *normalize-sort cs s* $\subseteq$ *s*
⟨*proof*⟩

lemma *normalize-sort-removed-elem-irrelevant'*:
assumes *sort-ex cs* (*insert c s*)
assumes *c* $\notin$ (*normalize-sort cs* (*insert c s*))
shows *normalize-sort cs* (*insert c s*) = *normalize-sort cs s*
⟨*proof*⟩

corollary *normalize-sort-removed-elem-irrelevant*:
assumes *sort-ex cs* (*insert c s*)
assumes *c* $\notin$ (*normalize-sort cs* (*insert c s*))

shows *normalize-sort cs* (*insert c s*) = *normalize-sort cs s*
⟨*proof*⟩

lemma *normalize-sort-nempt-is-nempty*:
assumes *finite*: *finite s*
assumes *nempty*: $s \neq$ *full-sort*
assumes *sort-ex cs s*
shows *normalize-sort cs s* $\neq$ *full-sort*
⟨*proof*⟩

lemma *choose-smaller-in-sort*:
assumes *elem*: $c \in s$ **and** *nelem*: $c \notin$ (*normalize-sort cs s*) **and** *sort-ex cs s*
obtains c' **where** $c' \in s$ **and** *class-les cs c' c*
⟨*proof*⟩

lemma *normalize-ex-bound'*:
assumes *finite*: *finite s* **and** *elem*: $c \in s$ **and** *nelem*: $c \notin$ (*normalize-sort cs s*)
and *sort-ex cs s*
shows $\exists\, c' \in$ (*normalize-sort cs s*) . *class-les cs c' c*
⟨*proof*⟩

corollary *normalize-ex-bound*:
assumes *finite*: *finite s* **and** *elem*: $c \in s$ **and** *nelem*: $c \notin$ (*normalize-sort cs s*)
and *sort-ex cs s*
obtains c' **where** $c' \in$ (*normalize-sort cs s*) **and** *class-les cs c' c*
⟨*proof*⟩

lemma *sort-ex cs s* $\Longrightarrow$ *sort-leq cs s* (*normalize-sort cs s*)
⟨*proof*⟩
lemma *sort-eqv-normalize-sort*:
assumes *finite s*
assumes *sort-ex cs s*
shows *sort-eqv cs s* (*normalize-sort cs s*)
⟨*proof*⟩

lemma *normalize-sort-eq-imp-sort-eqv*: *sort-ex cs s1* $\Longrightarrow$ *sort-ex cs s2* $\Longrightarrow$ *finite s1* $\Longrightarrow$ *finite s2*
$\Longrightarrow$ *normalize-sort cs s1* = *normalize-sort cs s2*
$\Longrightarrow$ *sort-eqv cs s1 s2*
⟨*proof*⟩

lemma *class-leq cs c1 c2* $\longleftrightarrow$ *class-les cs c1 c2* $\vee$ (*c1*=*c2* $\wedge$ *class-ex cs c1*)
⟨*proof*⟩

lemma *sort-eqv-imp-normalize-sort-eq*:
assumes *sort-ex cs s1 sort-ex cs s2 sort-eqv cs s1 s2*
shows *normalize-sort cs s1* = *normalize-sort cs s2*
⟨*proof*⟩

corollary *sort-eqv-iff-normalize-sort-eq*:
assumes *finite s1 finite s2*
assumes *sort-ex cs s1 sort-ex cs s2*
shows *sort-eqv cs s1 s2* $\longleftrightarrow$ *normalize-sort cs s1* = *normalize-sort cs s2*
⟨*proof*⟩

end

lemma *tcsigs-sorts-defined*: *wf-osig oss* $\Longrightarrow$
($\forall$ *ars* $\in$ *ran* (*tcsigs oss*) . $\forall$ *ss* $\in$ *ran ars* . $\forall$ *s* $\in$ *set ss*. *sort-ex* (*subclass oss*) *s*)
⟨*proof*⟩

lemma *osig-subclass-loc*: *wf-osig oss* $\Longrightarrow$ *wf-subclass-loc* (*subclass oss*)
⟨*proof*⟩

lemma *wf-osig-imp-wf-subclass-loc*: *wf-osig oss* $\Longrightarrow$ *wf-subclass-loc* (*subclass oss*)
⟨*proof*⟩

lemma *has-sort-Tv-imp-sort-leq*: *has-sort oss* (*Tv idn S*) *S'* $\Longrightarrow$ *sort-leq* (*subclass oss*) *S S'*
⟨*proof*⟩

end

Constants for encoding class/sort constraints in term language

theory *SortConstants*
imports *Sorts*
begin

fun *dest-type* :: *term* $\Rightarrow$ *typ option* **where**
dest-type (*Ct nc* (*Ty nt* [*ty*])) =
(*if nc* = *STR* ''*Pure.type*'' $\wedge$ *nt* = *STR* ''*Pure.type*'' *then Some ty else None*)
| *dest-type t* = *None*

definition *type-map f t* = *map-option* (λ*ty*. *mk-type* (*f ty*)) (*dest-type t*)

consts *unsuffix* :: *name* $\Rightarrow$ *name* $\Rightarrow$ *name option*

abbreviation *class-of-const c* $\equiv$ (*unsuffix classN c*)

fun *dest-of-class* :: *term* $\Rightarrow$ (*typ* $*$ *class*) *option* **where**
dest-of-class (*Ct c-class* - $ *ty*) = *lift2-option Pair* (*dest-type ty*) (*class-of-const c-class*)
| *dest-of-class* - = *None*

definition *mk-of-sort ty S == map* (λ*c* . *mk-of-class ty c*) *S*

end

5 Wellformed Signature and Theory

theory *Theory*
imports *Term Sorts SortConstants*
begin

fun *typ-ok-sig* :: *signature* ⇒ *typ* ⇒ *bool* **where**
typ-ok-sig Σ (*Ty c Ts*) = (*case type-arity* Σ *c of*
None ⇒ *False*
| *Some ar* ⇒ *length Ts* = *ar* ∧ *list-all* (*typ-ok-sig* Σ) *Ts*)
| *typ-ok-sig* Σ (*Tv - S*) = *wf-sort* (*subclass* (*osig* Σ)) *S*

lemma *typ-ok-sig-imp-wf-type*: *typ-ok-sig* Σ *T* ⟹ *wf-type* Σ *T*
⟨*proof*⟩
lemma *wf-type-imp-typ-ok-sig*: *wf-type* Σ *T* ⟹ *typ-ok-sig* Σ *T*
⟨*proof*⟩

corollary *wf-type-iff-typ-ok-sig*[*iff*]: *wf-type* Σ *T* = *typ-ok-sig* Σ *T*
⟨*proof*⟩

fun *term-ok'* :: *signature* ⇒ *term* ⇒ *bool* **where**
term-ok' Σ (*Fv - T*) = *typ-ok-sig* Σ *T*
| *term-ok'* Σ (*Bv -*) = *True*
| *term-ok'* Σ (*Ct s T*) = (*case const-type* Σ *s of*
None ⇒ *False*
| *Some ty* ⇒ *typ-ok-sig* Σ *T* ∧ *tinstT T ty*)
| *term-ok'* Σ (*t* $ *u*) ⟷ *term-ok'* Σ *t* ∧ *term-ok'* Σ *u*
| *term-ok'* Σ (*Abs T t*) ⟷ *typ-ok-sig* Σ *T* ∧ *term-ok'* Σ *t*

lemma *term-ok'-imp-wf-term*: *term-ok'* Σ *t* ⟹ *wf-term* Σ *t*
⟨*proof*⟩
lemma *wf-term-imp-term-ok'*: *wf-term* Σ *t* ⟹ *term-ok'* Σ *t*
⟨*proof*⟩
corollary *wf-term-iff-term-ok'*[*iff*]: *wf-term* Σ *t* = *term-ok'* Σ *t*
⟨*proof*⟩

lemma *acyclic-empty*[*simp*]: *acyclic* {} ⟨*proof*⟩

lemma *wf-sig* (*Map.empty*, *Map.empty*, *empty-osig*)
⟨*proof*⟩
lemma
term-ok-imp-typ-ok-pre:

is-std-sig $\Sigma \Longrightarrow$ *wf-term* Σ *t* $\Longrightarrow$ *list-all* (*typ-ok-sig* Σ) *Ts*
$\Longrightarrow$ *typ-of1 Ts t* = *Some ty* $\Longrightarrow$ *typ-ok-sig* Σ *ty*
⟨*proof*⟩

lemma *theory-full-exhaust*: ($\bigwedge$*cto tao sorts axioms*.
Θ = ((*cto*, *tao*, *sorts*), *axioms*) $\Longrightarrow$ *P*)
$\Longrightarrow$ *P*
⟨*proof*⟩

definition [*simp*]: *typ-ok* Θ *T* ≡ *wf-type* (*sig* Θ) *T*
definition [*simp*]: *term-ok* Θ *t* ≡ *wt-term* (*sig* Θ) *t*

corollary *typ-of-subst-bv-no-change*: *typ-of t* ≠ *None* $\Longrightarrow$ *subst-bv u t* = *t*
⟨*proof*⟩
corollary *term-ok-subst-bv-no-change*: *term-ok* Θ *t* $\Longrightarrow$ *subst-bv u t* = *t*
⟨*proof*⟩

lemmas *eq-axs-def* = *eq-reflexive-ax-def eq-symmetric-ax-def eq-transitive-ax-def*
eq-intr-ax-def
eq-elim-ax-def eq-combination-ax-def eq-abstract-rule-ax-def

bundle *eq-axs-simp*
begin
declare *eq-axs-def*[*simp*]
declare *mk-all-list-def*[*simp*] *add-vars'-def*[*simp*] *bind-eq-Some-conv*[*simp*] *bind-fv-def*[*simp*]
end

lemma *typ-of-eq-ax*: *typ-of* (*eq-reflexive-ax*) = *Some propT*
typ-of (*eq-symmetric-ax*) = *Some propT*
typ-of (*eq-transitive-ax*) = *Some propT*
typ-of (*eq-intr-ax*) = *Some propT*
typ-of (*eq-elim-ax*) = *Some propT*
typ-of (*eq-combination-ax*) = *Some propT*
typ-of (*eq-abstract-rule-ax*) = *Some propT*
⟨*proof*⟩

lemma *term-ok-eq-ax*:
assumes *is-std-sig* (*sig* Θ)
shows *term-ok* Θ (*eq-reflexive-ax*)
term-ok Θ (*eq-symmetric-ax*)
term-ok Θ (*eq-transitive-ax*)
term-ok Θ (*eq-intr-ax*)
term-ok Θ (*eq-elim-ax*)
term-ok Θ (*eq-combination-ax*)
term-ok Θ (*eq-abstract-rule-ax*)
⟨*proof*⟩

lemma *wf-theory-imp-is-std-sig*: *wf-theory* $\Theta \Longrightarrow$ *is-std-sig* (*sig* Θ)

⟨*proof*⟩
lemma *wf-theory-imp-wf-sig*: *wf-theory* Θ ⟹ *wf-sig* (*sig* Θ)
⟨*proof*⟩

lemma
term-ok-imp-typ-ok:
wf-theory thy ⟹ *term-ok thy t* ⟹ *typ-of t* = *Some ty* ⟹ *typ-ok thy ty*
⟨*proof*⟩

lemma *axioms-terms-ok*: *wf-theory thy* ⟹ *A*∈*axioms thy* ⟹ *term-ok thy A*
⟨*proof*⟩

lemma *axioms-typ-of-propT*: *wf-theory thy* ⟹ *A*∈*axioms thy* ⟹ *typ-of A* = *Some propT*
⟨*proof*⟩

lemma *propT-ok*[*simp*]: *wf-theory* Θ ⟹ *typ-ok* Θ *propT*
⟨*proof*⟩

lemma *term-ok-mk-eqD*: *term-ok* Θ (*mk-eq s t*) ⟹ *term-ok* Θ *s* ∧ *term-ok* Θ *t*
⟨*proof*⟩
lemma *term-ok-app-eqD*: *term-ok* Θ (*s* $ *t*) ⟹ *term-ok* Θ *s* ∧ *term-ok* Θ *t*
⟨*proof*⟩

lemma *wf-type-Type-imp-mgd*:
wf-sig Σ ⟹ *wf-type* Σ (*Ty n Ts*) ⟹ *tcsigs* (*osig* Σ) *n* ≠ *None*
⟨*proof*⟩

lemma *term-ok-eta-expand*:
assumes *wf-theory* Θ *term-ok* Θ *f typ-of f* = *Some* (τ → τ′) *typ-ok* Θ τ
shows *term-ok* Θ (*Abs* τ (*f* $ *Bv 0*))
⟨*proof*⟩

lemma *term-ok′-incr-bv*: *term-ok′* Σ *t* ⟹ *term-ok′* Σ (*incr-bv inc lev t*)
⟨*proof*⟩

lemma *term-ok′-subst-bv2*: *term-ok′* Σ *s* ⟹ *term-ok′* Σ *u* ⟹ *term-ok′* Σ (*subst-bv2 s lev u*)
⟨*proof*⟩

lemma *term-ok′-subst-bv*: *term-ok′* Σ (*Abs T t*) ⟹ *term-ok′* Σ (*subst-bv* (*Fv x T*) *t*)
⟨*proof*⟩
lemma *term-ok-subst-bv*: *term-ok* Θ (*Abs T t*) ⟹ *term-ok* Θ (*subst-bv* (*Fv x T*) *t*)
⟨*proof*⟩

lemma *term-ok-subst-bv2-0*: *term-ok* Θ (*Abs T t*) ⟹ *term-ok* Θ (*subst-bv2 t 0* (*Fv x T*))

⟨*proof*⟩

lemma *has-sort-empty*[*simp*]:
assumes *wf-sig* Σ *wf-type* Σ *T*
shows *has-sort* (*osig* Σ) *T full-sort*
⟨*proof*⟩

lemma *typ-Fv-of-full-sort*[*simp*]:
wf-theory Θ ⟹ *term-ok* Θ (*Fv v T*) ⟹ *has-sort* (*osig* (*sig* Θ)) *T full-sort*
⟨*proof*⟩

end

6 More on Substitutions

theory *Term-Subst*
imports *Term*
begin

fun *subst-typ* :: ((*variable* × *sort*) × *typ*) *list* ⇒ *typ* ⇒ *typ* **where**
subst-typ insts (*Ty a Ts*) =
Ty a (*map* (*subst-typ insts*) *Ts*)
| *subst-typ insts* (*Tv idn S*) = *the-default* (*Tv idn S*)
(*lookup* (λ*x* . *x* = (*idn*, *S*)) *insts*)

lemma *subst-typ-nil*[*simp*]: *subst-typ* [] *T* = *T*
⟨*proof*⟩

lemma *subst-typ-irrelevant-order*:
assumes *distinct* (*map fst pairs*) **and** *distinct* (*map fst pairs'*) **and** *set pairs* = *set pairs'*
shows *subst-typ pairs T* = *subst-typ pairs' T*
⟨*proof*⟩

lemma *subst-typ-simulates-tsubstT-gen'*: *distinct l* ⟹ *tvsT T* ⊆ *set l*
⟹ *tsubstT T* ϱ = *subst-typ* (*map* (λ(*x*,*y*).((*x*,*y*), ϱ *x y*)) *l*) *T*
⟨*proof*⟩

lemma *subst-typ-simulates-tsubstT-gen*: *tsubstT T* ϱ
= *subst-typ* (*map* (λ(*x*,*y*).((*x*,*y*), ϱ *x y*)) (*SOME l* . *distinct l* ∧ *tvsT T* ⊆ *set l*))
T
⟨*proof*⟩

corollary *subst-typ-simulates-tsubstT*: *tsubstT T* ϱ
= *subst-typ* (*map* (λ(*x*,*y*).((*x*,*y*), ϱ *x y*)) (*SOME l* . *distinct l* ∧ *set l* = *tvsT T*))
T
⟨*proof*⟩

lemma *tsubstT-simulates-subst-typ*: *subst-typ insts T*
= tsubstT T (*λidn S . the-default* (*Tv idn S*) (*lookup* (*λx. x=*(*idn*, *S*)) *insts*))
⟨*proof*⟩

lemma *subst-typ-comp*:
subst-typ inst1 (*subst-typ inst2 T*) = *subst-typ* (*map* (*apsnd* (*subst-typ inst1*)) *inst2* @ *inst1*) *T*
⟨*proof*⟩

lemma *subst-typ-AList-clearjunk*: *subst-typ insts T* = *subst-typ* (*AList.clearjunk insts*) *T*
⟨*proof*⟩

fun *subst-type-term* :: ((*variable* × *sort*) × *typ*) *list* ⇒
((*variable* × *typ*) × *term*) *list* ⇒ *term* ⇒ *term* **where**
subst-type-term instT insts (*Ct c T*) = *Ct c* (*subst-typ instT T*)
| *subst-type-term instT insts* (*Fv idn T*) = (*let T′* = *subst-typ instT T in*
the-default (*Fv idn T′*) (*lookup* (*λx. x* = (*idn*, *T′*)) *insts*))
| *subst-type-term - -* (*Bv n*) = *Bv n*
| *subst-type-term instT insts* (*Abs T t*) = *Abs* (*subst-typ instT T*) (*subst-type-term instT insts t*)
| *subst-type-term instT insts* (*t* $ *u*) = *subst-type-term instT insts t* $ *subst-type-term instT insts u*

lemma *subst-type-term-empty-no-change*[*simp*]: *subst-type-term* [] [] *t* = *t*
⟨*proof*⟩

lemma *subst-type-term-irrelevant-order*:
assumes *instT-assms*: *distinct* (*map fst instT*) *distinct* (*map fst instT′*) *set instT* = *set instT′*
assumes *insts-assms*: *distinct* (*map fst insts*) *distinct* (*map fst insts′*) *set insts* = *set insts′*
shows *subst-type-term instT insts t* = *subst-type-term instT′ insts′ t*
⟨*proof*⟩

lemma *subst-type-term-simulates-subst-tsubst-gen′*:
assumes *lty-assms*: *distinct lty tvs t* ⊆ *set lty*
assumes *lt-assms*: *distinct lt fv* (*tsubst t ϱty*) ⊆ *set lt*
shows *subst* (*tsubst t ϱty*) *ϱt*
= *subst-type-term* (*map* (*λ*(*x*,*y*).((*x*,*y*), *ϱty x y*)) *lty*) (*map* (*λ*(*x*,*y*).((*x*,*y*), *ϱt x y*)) *lt*) *t*
⟨*proof*⟩

corollary *subst-type-term-simulates-subst-tsubst*: *subst* (*tsubst t ϱty*) *ϱt*
= *subst-type-term* (*map* (*λ*(*x*,*y*).((*x*,*y*), *ϱty x y*)) (*SOME lty . distinct lty* ∧ *tvs t* = *set lty*))

(*map* (λ(*x*,*y*).((*x*,*y*), ϱ*t* *x* *y*)) (*SOME* *lt* . *distinct* *lt* ∧ *fv* (*tsubst* *t* ϱ*ty*) = *set* *lt*)) *t*
⟨*proof*⟩

abbreviation *subst-typ′* *pairs* *t* ≡ *map-types* (*subst-typ* *pairs*) *t*

lemma *subst-typ′-nil*[*simp*]: *subst-typ′* [] *A* = *A*
⟨*proof*⟩

lemma *subst-typ′-simulates-tsubst-gen′*: *distinct* *pairs* ⟹ *tvs* *t* ⊆ *set* *pairs*
⟹ *tsubst* *t* ϱ = *subst-typ′* (*map* (λ(*x*,*y*).((*x*,*y*), ϱ *x* *y*)) *pairs*) *t*
⟨*proof*⟩

lemma *subst-typ′-simulates-tsubst-gen*: *tsubst* *t* ϱ
= *subst-typ′* (*map* (λ(*x*,*y*).((*x*,*y*), ϱ *x* *y*)) (*SOME* *l* . *distinct* *l* ∧ *tvs* *t* ⊆ *set* *l*)) *t*
⟨*proof*⟩

lemma *tsubst-simulates-subst-typ′*: *subst-typ′* *insts* *T*
= *tsubst* *T* (λ*idn* *S* . *the-default* (*Tv* *idn* *S*) (*lookup* (λ*x*. *x*=(*idn*, *S*)) *insts*))
⟨*proof*⟩

lemma *subst-type-add-degenerate-instance*:
(*idx*,*s*) ∉ *set* (*map* *fst* *insts*) ⟹ *subst-typ* *insts* *T* = *subst-typ* (((*idx*,*s*), *Tv* *idx* *s*)#*insts*) *T*
⟨*proof*⟩

lemma *subst-typ′-add-degenerate-instance*:
(*idx*,*s*) ∉ *set* (*map* *fst* *insts*) ⟹ *subst-typ′* *insts* *t* = *subst-typ′* (((*idx*,*s*), *Tv* *idx* *s*)#*insts*) *t*
⟨*proof*⟩

lemma *subst-typ′-comp*:
subst-typ′ *inst1* (*subst-typ′* *inst2* *t*) = *subst-typ′* (*map* (*apsnd* (*subst-typ* *inst1*)) *inst2* @ *inst1*) *t*
⟨*proof*⟩

lemma *subst-typ′-AList-clearjunk*: *subst-typ′* *insts* *t* = *subst-typ′* (*AList.clearjunk* *insts*) *t*
⟨*proof*⟩

fun *subst-term* :: ((*variable* ∗ *typ*) ∗ *term*) *list* ⇒ *term* ⇒ *term* **where**
subst-term *insts* (*Ct* *c* *T*) = *Ct* *c* *T*
| *subst-term* *insts* (*Fv* *idn* *T*) = *the-default* (*Fv* *idn* *T*) (*lookup* (λ*x*. *x*=(*idn*, *T*)) *insts*)
| *subst-term* - (*Bv* *n*) = *Bv* *n*
| *subst-term* *insts* (*Abs* *T* *t*) = *Abs* *T* (*subst-term* *insts* *t*)

| *subst-term insts* (*t* $ *u*) = *subst-term insts t* $ *subst-term insts u*

lemma *subst-term-empty-no-change*[*simp*]: *subst-term* [] *t* = *t*
⟨*proof*⟩

lemma *subst-type-term-without-type-insts-eq-subst-term*[*simp*]:
subst-type-term [] *insts t* = *subst-term insts t*
⟨*proof*⟩

lemma *subst-type-term-split-levels*:
subst-type-term instT insts t = *subst-term insts* (*subst-typ' instT t*)
⟨*proof*⟩

lemma *subst-typ-stepwise*:
assumes *distinct* (*map fst instT*)
assumes ⋀*x* . *x* ∈ (⋃ *t* ∈ *snd* ‘ *set instT* . *tvsT t*) ⟹ *x* ∉ *fst* ‘ *set instT*
shows *subst-typ instT T* = *fold* (λ*single acc* . *subst-typ* [*single*] *acc*) *instT T*
⟨*proof*⟩

corollary *subst-typ-split-first*:
assumes *distinct* (*map fst* (*x*#*xs*))
assumes ⋀*y* . *y* ∈ (⋃ *t* ∈ *snd* ‘ *set* (*x*#*xs*) . *tvsT t*) ⟹ *y* ∉ *fst* ‘ (*set* (*x*#*xs*))
shows *subst-typ* (*x*#*xs*) *T* = *subst-typ xs* (*subst-typ* [*x*] *T*)
⟨*proof*⟩

corollary *subst-typ-split-last*:
assumes *distinct* (*map fst* (*xs* @ [*x*]))
assumes ⋀*y* . *y* ∈ (⋃ *t* ∈ *snd* ‘ (*set* (*xs* @ [*x*])) . *tvsT t*) ⟹ *y* ∉ *fst* ‘ (*set* (*xs* @ [*x*]))
shows *subst-typ* (*xs* @ [*x*]) *T* = *subst-typ* [*x*] (*subst-typ xs T*)
⟨*proof*⟩

lemma *subst-typ'-stepwise*:
assumes *distinct* (*map fst instT*)
assumes ⋀*x* . *x* ∈ (⋃ *t* ∈ *snd* ‘ (*set instT*) . *tvsT t*) ⟹ *x* ∉ *fst* ‘ (*set instT*)
shows *subst-typ' instT t* = *fold* (λ*single acc* . *subst-typ'* [*single*] *acc*) *instT t*

⟨*proof*⟩

lemma *subst-term-stepwise*:
assumes *distinct* (*map fst insts*)
assumes ⋀*x* . *x* ∈ (⋃ *t* ∈ *snd* ‘ (*set insts*) . *fv t*) ⟹ *x* ∉ *fst* ‘ (*set insts*)
shows *subst-term insts t* = *fold* (λ*single acc* . *subst-term* [*single*] *acc*) *insts t*
⟨*proof*⟩

corollary *subst-term-split-last*:
assumes *distinct* (*map fst* (*xs* @ [*x*]))
assumes $\bigwedge y$. $y \in (\bigcup t \in snd$ ‘ (*set* (*xs* @ [*x*])) . *fv t*) $\Longrightarrow y \notin$ *fst* ‘ (*set* (*xs* @ [*x*]))
shows *subst-term* (*xs* @ [*x*]) *t* = *subst-term* [*x*] (*subst-term xs t*)
⟨*proof*⟩

corollary *subst-type-term-stepwise*:
assumes *distinct* (*map fst instT*)
assumes $\bigwedge x$. $x \in (\bigcup T \in snd$ ‘ (*set instT*) . *tvsT T*) $\Longrightarrow x \notin$ *fst* ‘ (*set instT*)
assumes *distinct* (*map fst insts*)
assumes $\bigwedge x$. $x \in (\bigcup t \in snd$ ‘ (*set insts*) . *fv t*) $\Longrightarrow x \notin$ *fst* ‘ (*set insts*)
shows *subst-type-term instT insts t*
= *fold* (λ*single* . *subst-term* [*single*]) *insts* (*fold* (λ*single* . *subst-typ′* [*single*]) *instT t*)
⟨*proof*⟩

lemma *distinct-fst-imp-distinct*: *distinct* (*map fst l*) $\Longrightarrow$ *distinct l* ⟨*proof*⟩
lemma *distinct-kv-list*: *distinct l* $\Longrightarrow$ *distinct* (*map* (λ*x*. (*x*, *f x*)) *l*) ⟨*proof*⟩

lemma *subst-subst-term*:
assumes *distinct l* **and** *fv t* ⊆ *set l*
shows *subst t* ϱ = *subst-term* (*map* (λ*x*.(*x*, *case-prod* ϱ *x*)) *l*) *t*
⟨*proof*⟩

lemma *subst-term-subst*:
assumes *distinct* (*map fst l*)
shows *subst-term l t* = *subst t* (*fold* (λ((*idn*, *T*), *t*) *f x y*. *if x*=*idn* ∧*y*=*T then t else f x y*) *l Fv*)
⟨*proof*⟩

lemma *subst-typ-combine-single*:
assumes *fresh-idn* ∉ *fst* ‘ *tvsT* τ
shows *subst-typ* [((*fresh-idn*, *S*), τ*2*)] (*subst-typ* [((*idn*, *S*), *Tv fresh-idn S*)] τ)
= *subst-typ* [((*idn*, *S*), τ*2*)] τ
⟨*proof*⟩

lemma *subst-typ-combine*:
assumes *length fresh-idns* = *length insts*
assumes *distinct fresh-idns*
assumes *distinct* (*map fst insts*)
assumes ∀ *idn* ∈ *set fresh-idns* . *idn* ∉ *fst* ‘ (*tvsT* τ ∪ (⋃ *ty*∈*snd* ‘ *set insts* . (*tvsT ty*)) ∪ (*fst* ‘ *set insts*))
shows *subst-typ insts* τ
= *subst-typ* (*zip* (*zip fresh-idns* (*map snd* (*map fst insts*))) (*map snd insts*))
(*subst-typ* (*zip* (*map fst insts*) (*map2 Tv fresh-idns* (*map snd* (*map fst insts*))))

τ)
⟨*proof*⟩

lemma *subst-typ-combine′*:
assumes *length fresh-idns* = *length insts*
assumes *distinct fresh-idns*
assumes *distinct* (*map fst insts*)
assumes ∀ *idn* ∈ *set fresh-idns* . *idn* ∉ *fst* ‘ (*tvsT* τ ∪ (⋃ *ty*∈*snd* ‘ *set insts* . (*tvsT ty*))
∪ (*fst* ‘ *set insts*))
shows *subst-typ insts* τ
= *fold* (λ*single acc* . *subst-typ* [*single*] *acc*) (*zip* (*zip fresh-idns* (*map snd* (*map fst insts*))) (*map snd insts*))
(*fold* (λ*single acc* . *subst-typ* [*single*] *acc*) (*zip* (*map fst insts*) (*map2 Tv fresh-idns* (*map snd* (*map fst insts*)))) τ)
⟨*proof*⟩

lemma *subst-typ′-combine*:
assumes *length fresh-idns* = *length insts*
assumes *distinct fresh-idns*
assumes *distinct* (*map fst insts*)
assumes ∀ *idn* ∈ *set fresh-idns* . *idn* ∉ *fst* ‘ (*tvs t* ∪ (⋃ *ty*∈*snd* ‘ *set insts* . (*tvsT ty*))
∪ (*fst* ‘ *set insts*))
shows *subst-typ′ insts t*
= *subst-typ′* (*zip* (*zip fresh-idns* (*map snd* (*map fst insts*))) (*map snd insts*))
(*subst-typ′* (*zip* (*map fst insts*) (*map2 Tv fresh-idns* (*map snd* (*map fst insts*)))) *t*)
⟨*proof*⟩

lemma *subst-term-combine*:
assumes *length fresh-idns* = *length insts*
assumes *distinct fresh-idns*
assumes *distinct* (*map fst insts*)
assumes ∀ *idn* ∈ *set fresh-idns* . *idn* ∉ *fst* ‘ (*fv t* ∪ (⋃ *t*∈*snd* ‘ *set insts* . (*fv t*))
∪ (*fst* ‘ *set insts*))
shows *subst-term insts t*
= *subst-term* (*zip* (*zip fresh-idns* (*map snd* (*map fst insts*))) (*map snd insts*))
(*subst-term* (*zip* (*map fst insts*) (*map2 Fv fresh-idns* (*map snd* (*map fst insts*)))) *t*)
⟨*proof*⟩

corollary *subst-term-combine′*:
assumes *length fresh-idns* = *length insts*
assumes *distinct fresh-idns*
assumes *distinct* (*map fst insts*)
assumes ∀ *idn* ∈ *set fresh-idns* . *idn* ∉ *fst* ‘ (*fv t* ∪ (⋃ *t*∈*snd* ‘ *set insts* . (*fv t*))
∪ (*fst* ‘ *set insts*))

shows *subst-term insts t*
= fold (λ*single acc . subst-term* [*single*] *acc*) (*zip* (*zip fresh-idns* (*map snd* (*map fst insts*))) (*map snd insts*))
(*fold* (λ*single acc . subst-term* [*single*] *acc*) (*zip* (*map fst insts*) (*map2 Fv fresh-idns* (*map snd* (*map fst insts*)))) *t*)
⟨*proof*⟩

lemma *subst-term-not-loose-bvar*:
assumes ¬ *loose-bvar t n is-closed b*
shows ¬ *loose-bvar* (*subst-term* [((*idn*,*T*),*b*)] *t*) *n*
⟨*proof*⟩

lemma *bind-fv2-subst-bv1-eq-subst-term*:
assumes ¬*loose-bvar t n is-closed b*
shows *subst-term* [((*idn*,*T*),*b*)] *t* = *subst-bv1* (*bind-fv2* (*idn*, *T*) *n t*) *n b*
⟨*proof*⟩

corollary
assumes *is-closed t is-closed b*
shows *subst-bv b* (*bind-fv* (*idn*, *T*) *t*) = (*subst-term* [((*idn*, *T*),*b*)] *t*)
⟨*proof*⟩

corollary *instantiate-var-same-typ*:
assumes *typ-a*: *typ-of a* = *Some* τ
assumes *closed-B*: ¬ *loose-bvar B lev*
shows *subst-bv1* (*bind-fv2* (*x*, τ) *lev B*) *lev a* = *subst-term* [((*x*, τ), *a*)] *B*
⟨*proof*⟩

corollary *instantiate-var-same-typ′*:
assumes *typ-a*: *typ-of a* = *Some* τ
assumes *closed-B*: *is-closed B*
shows *subst-bv a* (*bind-fv* (*x*, τ) *B*) = *subst-term* [((*x*, τ), *a*)] *B*
⟨*proof*⟩

corollary *instantiate-var-same-type′′*:
assumes *typ-a*: *typ-of a* = *Some* τ
assumes *closed-B*: *is-closed B*
shows *Abs* τ (*bind-fv* (*x*, τ) *B*) · *a* = *subst-term* [((*x*, τ), *a*)] *B*
⟨*proof*⟩

lemma *instantiate-vars-same-typ*:
assumes *typs*: *list-all* (λ((*idx*, *ty*), *t*) . *typ-of t* = *Some ty*) *insts*
assumes *closed-B*: ¬ *loose-bvar B lev*
shows *fold* (λ((*idx*, *ty*), *t*) *B* . *subst-bv1* (*bind-fv2* (*idx*, *ty*) *lev B*) *lev t*) *insts B*
= fold (λ*single* . *subst-term* [*single*]) *insts B*
⟨*proof*⟩

corollary *instantiate-vars-same-typ′*:
assumes *typs*: *list-all* (λ((*idx*, *ty*), *t*) . *typ-of t* = *Some ty*) *insts*
assumes *closed-B*: ¬ *loose-bvar B lev*
assumes *distinct*: *distinct* (*map fst insts*)
assumes *no-overlap*: ⋀*x* . *x* ∈ (⋃ *t* ∈ *snd* ‘ (*set insts*) . *fv t*) ⟹ *x* ∉ *fst* ‘ (*set insts*)
shows *fold* (λ((*idx*, *ty*), *t*) *B* . *subst-bv1* (*bind-fv2* (*idx*, *ty*) *lev B*) *lev t*) *insts B*
= *subst-term insts B*
⟨*proof*⟩

end

7 Names

theory *Name*
imports *Preliminaries Term*
HOL−Library.Char-ord
begin

fun *fresh-name* :: *string set* ⇒ *string* **where**
fresh-name S = (*if S*=*empty then ″a″ else replicate* (*Max* (*length* ‘ *S*) + *1*) (*CHR* ″*a*″))

lemma *fresh-name-fresh*:
assumes *finite S*
shows *fresh-name S* ∉ *S*
⟨*proof*⟩

context
includes *String.literal.lifting*
begin
lift-definition *fresh-name′* :: *String.literal set* ⇒ *String.literal* **is** *fresh-name*
⟨*proof*⟩

lemma [*code*]: *fresh-name′ S* = *String.implode* (*fresh-name* (*String.explode* ‘ *S*))
⟨*proof*⟩

lemma *fresh-name′-fresh*:
assumes *finite S*
shows *fresh-name′ S* ∉ *S*
⟨*proof*⟩
end

fun *variant-name* :: *name* ⇒ *name set* ⇒ (*name* × *name set*) **where**
variant-name s S = (*let s′* = (*fresh-name′ S*) *in* (*s′*, *insert s′ S*))

lemma *variant-name-fresh*:
assumes *finite S*
shows *fst* (*variant-name s S*) ∉ *S*
⟨*proof*⟩

lemma *variant-name-adds*:
shows *snd* (*variant-name s S*) = *insert* (*fst* (*variant-name s S*)) *S*
⟨*proof*⟩

fun *name* :: *variable* ⇒ *name* **where**
name (*variable.Free n*) = *n*
| *name* (*Var* (*n*,-)) = *n*

fun *variant-variable* :: *variable* ⇒ *variable set* ⇒ (*variable* × *variable set*) **where**
variant-variable (*variable.Free n*) *S* = (*let s′* = *fresh-name′* (*name* ‘ *S*) *in*
(*Free s′*, *insert* (*variable.Free s′*) *S*))
| *variant-variable* (*Var* (*n*,-)) *S* = (*let s′* = *fresh-name′* (*name* ‘ *S*) *in*
(*Var* (*s′*,*0*), *insert* (*Var* (*s′*,*0*)) *S*))

lemma *variant-variable-fresh*:
assumes *finite S*
shows *fst* (*variant-variable s S*) ∉ *S*
⟨*proof*⟩

lemma *variant-variable-adds*:
shows *snd* (*variant-variable s S*) = *insert* (*fst* (*variant-variable s S*)) *S*
⟨*proof*⟩

fun *variant-variables* :: *nat* ⇒ *variable* ⇒ *variable set* ⇒ (*variable list* × *variable set*) **where**
variant-variables 0 - S = ([], *S*)
| *variant-variables* (*Suc n*) *s S* =
(*let* (*s′*, *S′*) = *variant-variable s S in*
(*let* (*ss*, *S′′*) = *variant-variables n s′ S′ in*
(*s′*#*ss*, *S′′*)))

lemma *variant-names-fresh*:
assumes *finite S*
shows ∀ *s* ∈ *set* (*fst* (*variant-variables n s S*)) . *s* ∉ *S*
⟨*proof*⟩

lemma *variant-names-distinct*:
assumes *finite S*
shows *distinct* (*fst* (*variant-variables n s S*))

⟨*proof*⟩

corollary *variant-names-amount*:
assumes *finite S*
shows *length* (*fst* (*variant-variables n s S*)) = *n*
⟨*proof*⟩

abbreviation *fresh-rename-ns n B insts G* ≡ *fst* (*variant-variables n* (*Free STR* ''*lol*'')
(*fst* ‘ (*fv B* ∪ (⋃ *t*∈*snd* ‘ *set insts* . *fv t*) ∪ (*fst* ‘ *set insts*)) ∪ *G*))
abbreviation *fresh-rename-idns n B insts* ≡ *fresh-rename-ns n B insts*

lemma *map-Pair-zip-replicate-conv*: *map* (λ*x*. *Pair x c*) *l* = *zip l* (*replicate* (*length l*) *c*)
⟨*proof*⟩

lemma *distinct-fresh-rename-ns*: *finite G* ⟹ *distinct* (*fresh-rename-ns n B insts G*)
⟨*proof*⟩

lemma *fresh-fresh-rename-ns*: *finite G* ⟹ ∀ *nm* ∈ *set* (*fresh-rename-ns n B insts G*) .
nm ∉ (*fst* ‘ (*fv B* ∪ (⋃ *t* ∈ *snd* ‘ *set insts* . (*fv t*)) ∪ (*fst* ‘ *set insts*)) ∪ *G*)
⟨*proof*⟩

lemma *length-fresh-rename-ns*: *finite G* ⟹ *length* (*fresh-rename-ns n B insts G*) = *n*
⟨*proof*⟩

lemma *distinct-fresh-rename-idns*: *finite G* ⟹ *distinct* (*fresh-rename-idns n B insts G*)
⟨*proof*⟩

lemma *fresh-fresh-rename-idns*: *finite G* ⟹ ∀ *nm* ∈ *set* (*fresh-rename-idns n B insts G*) .
nm ∉ (*fst* ‘ (*fv B* ∪ (⋃ *t* ∈ *snd* ‘ *set insts* . (*fv t*)) ∪ (*fst* ‘ *set insts*)) ∪ *G*)
⟨*proof*⟩

lemma *length-fresh-rename-idns*: *finite G* ⟹ *length* (*fresh-rename-idns n B insts G*) = *n*
⟨*proof*⟩

end

8 Beta Normalization

theory *BetaNorm*
imports *Term*

begin

inductive *beta* :: *term* ⇒ *term* ⇒ *bool* (**infixl** ‹→$_\beta$› *50*)
where
beta [*simp*, *intro*!]: *Abs T s* $ *t* →$_\beta$ *subst-bv2 s 0 t*
| *appL* [*simp*, *intro*!]: *s* →$_\beta$ *t* ⟹ *s* $ *u* →$_\beta$ *t* $ *u*
| *appR* [*simp*, *intro*!]: *s* →$_\beta$ *t* ⟹ *u* $ *s* →$_\beta$ *u* $ *t*
| *abs* [*simp*, *intro*!]: *s* →$_\beta$ *t* ⟹ *Abs T s* →$_\beta$ *Abs T t*

abbreviation
beta-reds :: *term* ⇒ *term* ⇒ *bool* (**infixl** ‹→$_\beta^*$› *50*) **where**
s →$_\beta^*$ *t* == *beta*** *s t*

inductive-cases *beta-cases* [*elim*!]:
Bv i →$_\beta$ *t*
Fv idn S →$_\beta$ *t*
Abs T r →$_\beta$ *s*
s $ *t* →$_\beta$ *u*

declare *if-not-P* [*simp*] *not-less-eq* [*simp*]

lemma *rtrancl-beta-Abs* [*intro*!]:
s →$_\beta^*$ *s'* ⟹ *Abs T s* →$_\beta^*$ *Abs T s'*
⟨*proof*⟩

lemma *rtrancl-beta-AppL*:
s →$_\beta^*$ *s'* ⟹ *s* $ *t* →$_\beta^*$ *s'* $ *t*
⟨*proof*⟩

lemma *rtrancl-beta-AppR*:
t →$_\beta^*$ *t'* ⟹ *s* $ *t* →$_\beta^*$ *s* $ *t'*
⟨*proof*⟩

lemma *rtrancl-beta-App* [*intro*]:
s →$_\beta^*$ *s'* ⟹ *t* →$_\beta^*$ *t'* ⟹ *s* $ *t* →$_\beta^*$ *s'* $ *t'*
⟨*proof*⟩

theorem *subst-bv2-preserves-beta* [*simp*]:
r →$_\beta$ *s* ⟹ *subst-bv2 r k u* →$_\beta$ *subst-bv2 s k u*
⟨*proof*⟩

theorem *subst-bv2-preserves-beta'*: *r* →$_\beta^*$ *s* ⟹ *subst-bv2 r i t* →$_\beta^*$ *subst-bv2 s i t*
⟨*proof*⟩

theorem *lift-preserves-beta* [*simp*]:
r →$_\beta$ *s* ⟹ *lift r i* →$_\beta$ *lift s i*
⟨*proof*⟩
theorem *lift-preserves-beta'*: *r* →$_\beta^*$ *s* ⟹ *lift r i* →$_\beta^*$ *lift s i*

⟨*proof*⟩

theorem *subst-bv2-preserves-beta2* [*simp*]: $r \to_\beta s \Longrightarrow$ *subst-bv2 t i r* $\to_\beta{}^*$ *subst-bv2 t i s*
⟨*proof*⟩

theorem *subst-bv2-preserves-beta2'*: $r \to_\beta{}^* s \Longrightarrow$ *subst-bv2 t i r* $\to_\beta{}^*$ *subst-bv2 t i s*
⟨*proof*⟩

lemma *beta-preserves-typ-of1*: *typ-of1 Ts r* = *Some T* $\Longrightarrow$ $r \to_\beta s \Longrightarrow$ *typ-of1 Ts s* = *Some T*
⟨*proof*⟩

lemma *beta-preserves-typ-of*: *typ-of r* = *Some T* $\Longrightarrow$ $r \to_\beta s \Longrightarrow$ *typ-of s* = *Some T*
⟨*proof*⟩

lemma *beta-star-preserves-typ-of1*: $r \to_\beta{}^* s \Longrightarrow$ *typ-of1 Ts r* = *Some T* $\Longrightarrow$ *typ-of1 Ts s* = *Some T*
⟨*proof*⟩

lemma *beta-reducible-imp-beta-step*: *beta-reducible t* $\Longrightarrow \exists t'.\ t \to_\beta t'$
⟨*proof*⟩

lemma *beta-step-imp-beta-reducible*: $t \to_\beta t' \Longrightarrow$ *beta-reducible t*
⟨*proof*⟩

lemma *beta-norm-imp-beta-reds*: **assumes** *beta-norm t* = *Some t'* **shows** $t \to_\beta{}^* t'$
⟨*proof*⟩

corollary *beta-norm t* = *Some t'* $\Longrightarrow$ *typ-of1 Ts t* = *Some T* $\Longrightarrow$ *typ-of1 Ts t'* = *Some T*
⟨*proof*⟩

lemma *beta-imp-beta-norm*: **assumes** $t \to_\beta t' \neg$ *beta-reducible t'* **shows** *beta-norm t* = *Some t'*
⟨*proof*⟩

lemma *beta-subst-bv1*: $s \to_\beta t \Longrightarrow$ *subst-bv1 s lev x* $\to_\beta$ *subst-bv1 t lev x*
⟨*proof*⟩

lemma *beta-subst-bv*: $s \to_\beta t \Longrightarrow$ *subst-bv x s* $\to_\beta$ *subst-bv x t*
⟨*proof*⟩

lemma *subst-bv1-beta*:

subst-bv1 s (length (T#Ts)) x $\rightarrow_\beta$ *subst-bv1 t (length (T#Ts)) x*
$\Longrightarrow$ *typ-of1 Ts s = Some ty*
$\Longrightarrow$ *typ-of1 Ts t = Some ty*
$\Longrightarrow$ *s* $\rightarrow_\beta$ *t*
⟨*proof*⟩

fun *subst-bvs1'* :: *term* ⇒ *nat* ⇒ *term list* ⇒ *term* **where**
subst-bvs1' (Bv i) lev args = (if i < lev then Bv i
else if i − lev < length args then (nth args (i−lev))
else Bv (i − length args))
| *subst-bvs1' (Abs T body) lev args = Abs T (subst-bvs1' body (lev + 1) (map (λt. lift t 0) args))*
| *subst-bvs1' (f $ t) lev u = subst-bvs1' f lev u $ subst-bvs1' t lev u*
| *subst-bvs1' t - - = t*

lemma *subst-bvs1'-empty* [*simp*]: *subst-bvs1' t lev* [] = *t*
⟨*proof*⟩

lemma *subst-bvs1'-eq* [*simp*]: *args* ≠ [] $\Longrightarrow$ *subst-bvs1' (Bv k) k args = args ! 0*
⟨*proof*⟩
lemma *subst-bvs1'-eq'* [*simp*]: *i < length args* $\Longrightarrow$ *subst-bvs1' (Bv (k+i)) k args = args ! i*
⟨*proof*⟩

lemma *subst-bvs1'-gt* [*simp*]:
i + length args < j $\Longrightarrow$ *subst-bvs1' (Bv j) i args = Bv (j − length args)*
⟨*proof*⟩

lemma *subst-bv2-lt* [*simp*]: *j < i* $\Longrightarrow$ *subst-bvs1' (Bv j) i u = Bv j*
⟨*proof*⟩

lemma *subst-bvs1'-App*[*simp*]: *subst-bvs1' (s$t) k args*
= subst-bvs1' s k args $ subst-bvs1' t k args
⟨*proof*⟩

lemma *incr-bv-incr-bv*:
i < k + 1 $\Longrightarrow$ *incr-bv inc2 (k+inc1) (incr-bv inc1 i t) = incr-bv inc1 i (incr-bv inc2 k t)*
⟨*proof*⟩

lemma *subst-bvs1-subst-bvs1'*: *subst-bvs1 t n s = subst-bvs1' t n (map (incr-bv n 0) s)*
⟨*proof*⟩

theorem *subst-bvs1-subst-bvs1'-0*: *subst-bvs1 t 0 s = subst-bvs1' t 0 s*
⟨*proof*⟩

corollary *subst-bvs-subst-bvs1'*: *subst-bvs s t = subst-bvs1' t 0 s*

⟨*proof*⟩

lemma *no-loose-bvar-subst-bvs1'-unchanged*: ¬ *loose-bvar t lev* ⟹ *subst-bvs1' t lev args* = *t*
⟨*proof*⟩

lemma *subst-bvs1'-step*: ∀ *x* ∈ *set* (*a*#*args*) . *is-closed x* ⟹
subst-bvs1' t lev (*a*#*args*) = *subst-bvs1'* (*subst-bv2 t lev a*) *lev args*
⟨*proof*⟩

lemma *not-loose-bvar-incr-bv*: ¬ *loose-bvar a lev* ⟹ ¬ *loose-bvar* (*incr-bv inc lev a*) (*lev*+*inc*)
⟨*proof*⟩

lemma *not-loose-bvar-incr-bv-less*:
i < *j* ⟹ ¬ *loose-bvar* (*incr-bv inc i a*) (*lev*+*inc*) ⟹ ¬ *loose-bvar* (*incr-bv inc j a*) (*lev*+*inc*)
⟨*proof*⟩

lemma *subst-bvs1'-step-work*: ∀ *x* ∈ *set args* . *is-closed x* ⟹ ¬ *loose-bvar* (*subst-bv2 t lev a*) *lev* ⟹
subst-bvs1' t lev (*a*#*args*) = *subst-bvs1'* (*subst-bv2 t lev a*) *lev args*
⟨*proof*⟩

lemma *is-closed-subst-bv2-unchanged*: *is-closed t* ⟹ *subst-bv2 t n u* = *t*
⟨*proof*⟩

lemma *subst-bvs1'-step-extend-lower-level*: ∀ *x* ∈ *set* (*a*#*args*) . *is-closed x* ⟹
subst-bv2 (*subst-bvs1' t* (*Suc lev*) *args*) *lev a*
= *subst-bvs1' t lev* (*a*#*args*)
⟨*proof*⟩

corollary *subst-bvs-extend-lower-level*:
∀ *x* ∈ *set* (*a*#*args*) . *is-closed x* ⟹
subst-bv a (*subst-bvs1' t 1 args*) = *subst-bvs* (*a*#*args*) *t*
⟨*proof*⟩

lemma *subst-bvs1'-preserves-beta*:
∀ *x* ∈ *set u* . *is-closed x* ⟹ *r* →$_\beta$ *s* ⟹ *subst-bvs1' r k u* →$_\beta$ *subst-bvs1' s k u*
⟨*proof*⟩

lemma *subst-bvs1'-fold*: ∀ *x* ∈ *set args* . *is-closed x* ⟹
subst-bvs1' t lev args = *fold* (λ*arg t* . *subst-bv2 t lev arg*) *args t*
⟨*proof*⟩

lemma *subst-bvs1'-Abs*[*simp*]: ∀ *x* ∈ *set args* . *is-closed x* ⟹
subst-bvs1' (*Abs T t*) *lev args* = *Abs T* (*subst-bvs1' t* (*Suc lev*) *args*)

⟨*proof*⟩

lemma *subst-bvs-Abs*[*simp*]: ∀ *x* ∈ *set args* . *is-closed x* ⟹
subst-bvs args (*Abs T t*) = *Abs T* (*subst-bvs1'* *t* 1 *args*)
⟨*proof*⟩

lemma *subst-bvs1'-incr-bv* [*simp*]:
subst-bvs1' (*incr-bv* (*length ss*) *k t*) *k ss* = *t*
⟨*proof*⟩

lemma *lift-subst-bvs1'* [*simp*]:
j < *i* + 1 ⟹ *lift* (*subst-bvs1'* *t j ss*) *i*
= *subst-bvs1'* (*lift t* (*i* + *length ss*)) *j* (*map* (λ*s* . *lift s i*) *ss*)
⟨*proof*⟩

lemma *lift-subst-bvs1'-lt*:
i < *j* + 1 ⟹ *lift* (*subst-bvs1'* *t j ss*) *i*
= *subst-bvs1'* (*lift t i*) (*j* + 1) (*map* (λ*s* . *lift s i*) *ss*)
⟨*proof*⟩

lemma *subst-bvs1'-subst-bv2*:
i < *j* + 1 ⟹
subst-bv2(*subst-bvs1'* *t* (*Suc j*) (*map* (λ*v*. *lift v i*) *vs*)) *i* (*subst-bvs1'* *u j vs*)
= *subst-bvs1'* (*subst-bv2 t i u*) *j vs*
⟨*proof*⟩

lemma *fv-subst-bv2-upper-bound*: *fv* (*subst-bv2 t lev u*) ⊆ *fv t* ∪ *fv u*
⟨*proof*⟩
lemma *beta-fv*: *s* →$_\beta$ *t* ⟹ *fv t* ⊆ *fv s*
⟨*proof*⟩

lemma *loose-bvar1-subst-bvs1'-closeds*: ¬ *loose-bvar1 t lev* ⟹ *lev* < *k* ⟹ ∀ *x*∈*set us* . *is-closed x*
⟹ ¬ *loose-bvar1* (*subst-bvs1'* *t k us*) *lev*
⟨*proof*⟩

lemma *is-closed-subst-bvs1'-closeds*: ¬ *is-dependent t* ⟹ ∀ *x*∈*set us* . *is-closed x*
⟹ ¬ *is-dependent* (*subst-bvs1'* *t* (*Suc k*) *us*)
⟨*proof*⟩

end

Facts about beta normalization involving theories

theory *BetaNormProof*
imports *BetaNorm Theory*
begin

lemma *beta-preserves-term-ok'*: *term-ok'* Σ *r* ⟹ *r* →$_\beta$ *s* ⟹ *term-ok'* Σ *s*
⟨*proof*⟩

lemma *beta-preserves-term-ok*: *term-ok* Θ *r* $\Longrightarrow$ *r* $\rightarrow_\beta$ *s* $\Longrightarrow$ *term-ok* Θ *s*
⟨*proof*⟩

lemma *beta-star-preserves-term-ok'*: *r* $\rightarrow_\beta{}^*$ *s* $\Longrightarrow$ *term-ok'* Σ *r* $\Longrightarrow$ *term-ok'* Σ *s*
⟨*proof*⟩

corollary *beta-star-preserves-term-ok*: *r* $\rightarrow_\beta{}^*$ *s* $\Longrightarrow$ *term-ok thy r* $\Longrightarrow$ *term-ok thy s*
⟨*proof*⟩

corollary *term-ok-beta-norm*: *term-ok thy t* $\Longrightarrow$ *beta-norm t* = *Some t'* $\Longrightarrow$ *term-ok thy t'*
⟨*proof*⟩

end

9 Eta Normalization

theory *EtaNorm*
imports *Term BetaNorm*
begin

inductive
eta :: *term* ⇒ *term* ⇒ *bool* (**infixl** ‹$\rightarrow_\eta$› *50*)
where
eta [*simp*, *intro*]: ¬ *is-dependent s* $\Longrightarrow$ *Abs T* (*s* $ *Bv 0*) $\rightarrow_\eta$ *decr 0 s*
| *appL* [*simp*, *intro*]: *s* $\rightarrow_\eta$ *t* $\Longrightarrow$ *s* $ *u* $\rightarrow_\eta$ *t* $ *u*
| *appR* [*simp*, *intro*]: *s* $\rightarrow_\eta$ *t* $\Longrightarrow$ *u* $ *s* $\rightarrow_\eta$ *u* $ *t*
| *abs* [*simp*, *intro*]: *s* $\rightarrow_\eta$ *t* $\Longrightarrow$ *Abs T s* $\rightarrow_\eta$ *Abs T t*

abbreviation
eta-reds :: *term* ⇒ *term* ⇒ *bool* (**infixl** ‹$\rightarrow_\eta{}^*$› *50*) **where**
s $\rightarrow_\eta{}^*$ *t* ≡ *eta*** *s t*

abbreviation
eta-red0 :: *term* ⇒ *term* ⇒ *bool* (**infixl** ‹$\rightarrow_\eta{}^=$› *50*) **where**
s $\rightarrow_\eta{}^=$ *t* ≡ *eta*$^{==}$ *s t*

inductive-cases *eta-cases* [*elim*!]:
Abs T s $\rightarrow_\eta$ *z*
s $ *t* $\rightarrow_\eta$ *u*
Bv i $\rightarrow_\eta$ *t*

lemma *subst-bv2-not-free* [*simp*]: ¬ *loose-bvar1 s i* $\Longrightarrow$ *subst-bv2 s i t* = *subst-bv2 s i u*
⟨*proof*⟩

lemma *free-lift* [*simp*]:

loose-bvar1 (*lift t k*) *i* = (*i* < *k* ∧ *loose-bvar1 t i* ∨ *k* < *i* ∧ *loose-bvar1 t* (*i* − *1*))
⟨*proof*⟩

lemma *free-subst-bv2* [*simp*]:
loose-bvar1 (*subst-bv2 s k t*) *i* =
(*loose-bvar1 s k* ∧ *loose-bvar1 t i* ∨ *loose-bvar1 s* (*if i* < *k then i else i* + *1*))
⟨*proof*⟩

lemma *free-eta*: *s* →$_\eta$ *t* ⟹ *loose-bvar1 t i* = *loose-bvar1 s i*
⟨*proof*⟩

lemma *not-free-eta*:
s →$_\eta$ *t* ⟹ ¬ *loose-bvar1 s i* ⟹ ¬ *loose-bvar1 t i*
⟨*proof*⟩

lemma *no-loose-bvar1-subst-bv2-decr*: ¬ *loose-bvar1 t i* ⟹ *subst-bv2 t i x* = *decr i t*
⟨*proof*⟩

lemma *eta-subst-bv2* [*simp*]:
s →$_\eta$ *t* ⟹ *subst-bv2 s i u* →$_\eta$ *subst-bv2 t i u*
⟨*proof*⟩

theorem *lift-subst-bv2-dummy*: ¬ *loose-bvar s i* ⟹ *lift* (*decr i s*) *i* = *s*
⟨*proof*⟩

lemma *decr-is-closed*[*simp*]: *is-closed t* ⟹ *decr lev t* = *t*
⟨*proof*⟩

lemma *eta-reducible-imp-eta-step*: *eta-reducible t* ⟹ ∃ *t'*. *t* →$_\eta$ *t'*
⟨*proof*⟩

lemma *eta-step-imp-eta-reducible*: *t* →$_\eta$ *t'* ⟹ *eta-reducible t*
⟨*proof*⟩

lemma *eta-reds-appR*: *s* →$_\eta$* *t* ⟹ *u* $ *s* →$_\eta$* *u* $ *t*
⟨*proof*⟩
lemma *eta-reds-appL*: *s* →$_\eta$* *t* ⟹ *s* $ *u* →$_\eta$* *t* $ *u*
⟨*proof*⟩
lemma *eta-reds-abs*: *s* →$_\eta$* *t* ⟹ *Abs T s* →$_\eta$* *Abs T t*
⟨*proof*⟩

lemma *eta-norm-imp-eta-reds*: **assumes** *eta-norm t* = *t'* **shows** *t* →$_\eta$* *t'*
⟨*proof*⟩

lemma *rtrancl-eta-App*:
s →$_\eta$* *s'* ⟹ *t* →$_\eta$* *t'* ⟹ *s* $ *t* →$_\eta$* *s'* $ *t'*
⟨*proof*⟩

lemma *eta-preserves-typ-of1*: $t \rightarrow_\eta t' \Longrightarrow$ *typ-of1 Ts t* = *Some* $\tau \Longrightarrow$ *typ-of1 Ts t'* = *Some* τ
⟨*proof*⟩

lemma *eta-preserves-typ-of*: $t \rightarrow_\eta t' \Longrightarrow$ *typ-of t* = *Some* $\tau \Longrightarrow$ *typ-of t'* = *Some* τ
⟨*proof*⟩

lemma *eta-star-preserves-typ-of1*: $r \rightarrow_\eta^* s \Longrightarrow$ *typ-of1 Ts r* = *Some T* $\Longrightarrow$ *typ-of1 Ts s* = *Some T*
⟨*proof*⟩

lemma *eta-star-preserves-typ-of*: $r \rightarrow_\eta^* s \Longrightarrow$ *typ-of r* = *Some T* $\Longrightarrow$ *typ-of s* = *Some T*
⟨*proof*⟩

lemma *subst-bvs1'-decr*: $\forall x \in$ *set us. is-closed x* $\Longrightarrow \neg$ *loose-bvar1 t k* $\Longrightarrow$ *subst-bvs1' (decr k t) k us* = *decr k* (*subst-bvs1' t* (*Suc k*) *us*)
⟨*proof*⟩

lemma *subst-bvs-decr*: $\forall x \in$ *set us. is-closed x* $\Longrightarrow \neg$ *is-dependent t* $\Longrightarrow$ *subst-bvs us* (*decr 0 t*) = *decr 0* (*subst-bvs1' t 1 us*)
⟨*proof*⟩

end

Facts about eta normalization involving theories

theory *EtaNormProof*
imports *EtaNorm Theory*

BetaNormProof
begin

lemma *term-ok'-decr*: *term-ok'* Σ *t* $\Longrightarrow$ *term-ok'* Σ (*decr i t*)
⟨*proof*⟩

lemma *eta-preserves-term-ok'*: *term-ok'* Σ *r* $\Longrightarrow r \rightarrow_\eta s \Longrightarrow$ *term-ok'* Σ *s*
⟨*proof*⟩

lemma *eta-preserves-term-ok*: *term-ok* Θ *r* $\Longrightarrow r \rightarrow_\eta s \Longrightarrow$ *term-ok* Θ *s*
⟨*proof*⟩

lemma *eta-star-preserves-term-ok'*: $r \rightarrow_\eta^* s \Longrightarrow$ *term-ok'* Σ *r* $\Longrightarrow$ *term-ok'* Σ *s*
⟨*proof*⟩

corollary *eta-star-preserves-term-ok*: $r \rightarrow_\eta^* s \Longrightarrow$ *term-ok thy r* $\Longrightarrow$ *term-ok thy s*
⟨*proof*⟩

corollary *term-ok-eta-norm*: *term-ok thy t* $\Longrightarrow$ *eta-norm t* = *t'*$\Longrightarrow$ *term-ok thy t'*
⟨*proof*⟩

end

10 Logic

theory *Logic*
imports *Theory Term-Subst SortConstants Name BetaNormProof EtaNormProof*
begin

term *proves*

abbreviation *inst-ok* Θ *insts* ≡
distinct (*map fst insts*) — No duplicates, makes stuff easier
∧ *list-all* (*typ-ok* Θ) (*map snd insts*) — Stuff I substitute in is well typed
∧ *list-all* (λ((*idn*, *S*), *T*) . *has-sort* (*osig* (*sig* Θ)) *T S*) *insts* — Types "fit" in the Fviables

lemma *inst-ok-imp-wf-inst*:
inst-ok Θ *insts* $\Longrightarrow$ *wf-inst* Θ (λ*idn S* .*the-default* (*Tv idn S*) (*lookup* (λ*x*. *x*=(*idn*, *S*)) *insts*))
⟨*proof*⟩

lemma *term-ok'-eta-norm*: *term-ok'* Σ *t* $\Longrightarrow$ *term-ok'* Σ (*eta-norm t*)
⟨*proof*⟩
corollary *term-ok-eta-norm*: *term-ok thy t* $\Longrightarrow$ *term-ok thy* (*eta-norm t*)
⟨*proof*⟩

abbreviation *beta-eta-norm t* ≡ *map-option eta-norm* (*beta-norm t*)

lemma *beta-eta-norm t* = *Some t'* $\Longrightarrow$ ¬ *eta-reducible t'*
⟨*proof*⟩
lemma *term-ok-beta-eta-norm*: *term-ok thy t* $\Longrightarrow$ *beta-eta-norm t* = *Some t'* $\Longrightarrow$ *term-ok thy t'*
⟨*proof*⟩
lemma *typ-of-beta-eta-norm*:
typ-of t = *Some T* $\Longrightarrow$ *beta-eta-norm t* = *Some t'* $\Longrightarrow$ *typ-of t'* = *Some T*
⟨*proof*⟩

lemma *inst-ok-nil*[*simp*]: *inst-ok* Θ [] ⟨*proof*⟩

lemma *axiom-subst-typ'*:
assumes *wf-theory* Θ *A*∈*axioms* Θ *inst-ok* Θ *insts*
shows Θ, Γ ⊢ *subst-typ' insts A*
⟨*proof*⟩

corollary *axiom'*: *wf-theory* Θ $\Longrightarrow$ *A* ∈ *axioms* Θ $\Longrightarrow$ Θ, Γ ⊢ *A*

⟨*proof*⟩

lemma *has-sort-Tv-refl*: *wf-osig oss* ⟹ *sort-ex* (*subclass oss*) *S* ⟹ *has-sort oss* (*Tv v S*) *S*
⟨*proof*⟩

lemma *has-sort-Tv-refl'*:
wf-theory Θ ⟹ *typ-ok* Θ (*Tv v S*) ⟹ *has-sort* (*osig* (*sig* Θ)) (*Tv v S*) *S*
⟨*proof*⟩

lemma *wf-inst-imp-inst-ok*:
wf-theory Θ ⟹ *distinct l* ⟹ ∀(*v*, *S*) ∈ *set l* . *typ-ok* Θ (*Tv v S*) ⟹ *wf-inst* Θ ϱ
⟹ *inst-ok* Θ (*map* (λ(*v*, *S*) . ((*v*, *S*), ϱ *v S*)) *l*)
⟨*proof*⟩

lemma *typs-of-fv-subset-Types*: *snd* ‘ *fv t* ⊆ *Types t*
⟨*proof*⟩
lemma *osig-tvsT-subset-SortsT*: *snd* ‘ *tvsT T* ⊆ *SortsT T*
⟨*proof*⟩
lemma *osig-tvs-subset-Sorts*: *snd* ‘ *tvs t* ⊆ *Sorts t*
⟨*proof*⟩

lemma *term-ok-Types-imp-typ-ok-pre*:
is-std-sig Σ ⟹ *term-ok'* Σ *t* ⟹ τ ∈ *Types t* ⟹ *typ-ok-sig* Σ τ
⟨*proof*⟩

lemma *term-ok-Types-typ-ok*: *wf-theory* Θ ⟹ *term-ok* Θ *t* ⟹ τ ∈ *Types t* ⟹ *typ-ok* Θ τ
⟨*proof*⟩

lemma *term-ok-fv-imp-typ-ok-pre*:
is-std-sig Σ ⟹ *term-ok'* Σ *t* ⟹ (*x*,τ) ∈ *fv t* ⟹ *typ-ok-sig* Σ τ
⟨*proof*⟩

lemma *term-ok-vars-typ-ok*: *wf-theory* Θ ⟹ *term-ok* Θ *t* ⟹ (*x*, τ) ∈ *fv t* ⟹ *typ-ok* Θ τ
⟨*proof*⟩

lemma *typ-ok-TFreesT-imp-sort-ok-pre*:
is-std-sig Σ ⟹ *typ-ok-sig* Σ *T* ⟹ (*x*, *S*) ∈ *tvsT T* ⟹ *wf-sort* (*subclass* (*osig* Σ)) *S*
⟨*proof*⟩

lemma *term-ok-TFrees-imp-sort-ok-pre*:
is-std-sig Σ ⟹ *term-ok'* Σ *t* ⟹ (*x*, *S*) ∈ *tvs t* ⟹ *wf-sort* (*subclass* (*osig* Σ)) *S*
⟨*proof*⟩

lemma *typ-ok-tvsT-imp-sort-ok-pre*:
is-std-sig $\Sigma \Longrightarrow$ *typ-ok-sig* Σ $T \Longrightarrow (x,S) \in tvsT\ T \Longrightarrow$ *wf-sort* (*subclass* (*osig* Σ)) S
⟨*proof*⟩

lemma *term-ok-tvars-sort-ok*:
assumes *wf-theory* Θ *term-ok* Θ t $(x, S) \in tvs\ t$
shows *wf-sort* (*subclass* (*osig* (*sig* Θ))) S
⟨*proof*⟩

lemma *term-ok'-bind-fv2*:
assumes *term-ok'* Σ t
shows *term-ok'* Σ (*bind-fv2* (v,T) *lev* t)
⟨*proof*⟩

lemma *term-ok'-bind-fv*:
assumes *term-ok'* Σ t
shows *term-ok'* Σ (*bind-fv* (v,τ) t)
⟨*proof*⟩

lemma *term-ok'-Abs-fv*:
assumes *term-ok'* Σ t *typ-ok-sig* Σ τ
shows *term-ok'* Σ (*Abs* τ (*bind-fv* (v,τ) t))
⟨*proof*⟩

lemma *term-ok'-mk-all*:
assumes *wf-theory* Θ **and** *term-ok'* (*sig* Θ) B **and** *typ-of* B = *Some propT*
and *typ-ok* Θ τ
shows *term-ok'* (*sig* Θ) (*mk-all* x τ B)
⟨*proof*⟩

lemma *term-ok-mk-all*:
assumes *wf-theory* Θ **and** *term-ok'* (*sig* Θ) B **and** *typ-of* B = *Some propT* **and** *typ-ok* Θ τ
shows *term-ok* Θ (*mk-all* x τ B)
⟨*proof*⟩

lemma *term-ok'-incr-boundvars*:
term-ok' (*sig* Θ) $t \Longrightarrow$ *term-ok'* (*sig* Θ) (*incr-boundvars lev* t)
⟨*proof*⟩

lemma *term-ok'-subst-bv1*:
assumes *term-ok'* (*sig* Θ) f **and** *term-ok'* (*sig* Θ) u
shows *term-ok'* (*sig* Θ) (*subst-bv1* f *lev* u)
⟨*proof*⟩

lemma *term-ok'-subst-bv*:
assumes *term-ok'* (*sig* Θ) f **and** *term-ok'* (*sig* Θ) u

shows *term-ok′* (*sig* Θ) (*subst-bv f u*)
⟨*proof*⟩

lemma *term-ok′-betapply*:
assumes *term-ok′* (*sig* Θ) *f term-ok′* (*sig* Θ) *u*
shows *term-ok′* (*sig* Θ) (*f* ⋅ *u*)
⟨*proof*⟩

lemma *term-ok-betapply*:
assumes *term-ok* Θ *f term-ok* Θ *u*
assumes *typ-of f* = *Some* (*uty* → *tty*) *typ-of u* = *Some uty*
shows *term-ok* Θ (*f* ⋅ *u*)
⟨*proof*⟩

lemma *typ-ok-sig-subst-typ*:
assumes *is-std-sig* Σ **and** *typ-ok-sig* Σ *ty* **and** *distinct* (*map fst insts*)
and *list-all* (*typ-ok-sig* Σ) (*map snd insts*)
shows *typ-ok-sig* Σ (*subst-typ insts ty*)
⟨*proof*⟩

corollary *subst-typ-tinstT*: *tinstT* (*subst-typ insts ty*) *ty*
⟨*proof*⟩

lemma *tsubstT-trans*: *tsubstT ty ϱ1* = *ty1* ⟹ *tsubstT ty1 ϱ2* = *ty2*
⟹ *tsubstT ty* (λ*idx s* . *case ϱ1 idx s of Tv idx′ s′* ⇒ *ϱ2 idx′ s′*
| *Ty s Ts* ⇒ *Ty s* (*map* (λ*T*. *tsubstT T ϱ2*) *Ts*)) = *ty2*
⟨*proof*⟩

corollary *tinstT-trans*: *tinstT ty1 ty* ⟹ *tinstT ty2 ty1* ⟹ *tinstT ty2 ty*
⟨*proof*⟩

lemma *term-ok′-subst-typ′*:
assumes *is-std-sig* Σ **and** *term-ok′* Σ *t* **and** *distinct* (*map fst insts*)
and *list-all* (*typ-ok-sig* Σ) (*map snd insts*)
shows *term-ok′* Σ (*subst-typ′ insts t*)
⟨*proof*⟩

lemma
term-ok′-occs:
is-std-sig Σ ⟹ *term-ok′* Σ *t* ⟹ *occs u t* ⟹ *term-ok′* Σ *u*
⟨*proof*⟩

lemma *typ-of1-tsubst*:
typ-of1 Ts t = *Some ty* ⟹ *typ-of1* (*map* (λ*T* . *tsubstT T ϱ*) *Ts*) (*tsubst t ϱ*) =
Some (*tsubstT ty ϱ*)
⟨*proof*⟩

corollary *typ-of1-tsubst-weak*:
assumes *typ-of1 Ts t = Some ty*
assumes *typ-of1 (map (λT . tsubstT T ϱ) Ts) (tsubst t ϱ) = Some ty′*
shows *tsubstT ty ϱ = ty′*
⟨*proof*⟩

lemma *tsubstT-no-change*[*simp*]: *tsubstT T Tv = T*
⟨*proof*⟩

lemma *term-ok-mk-eq-same-typ*:
assumes *wf-theory Θ*
assumes *term-ok Θ (mk-eq s t)*
shows *typ-of s = typ-of t*
⟨*proof*⟩

lemma *typ-of-eta-expand*: *typ-of f = Some (τ → τ′) ⟹ typ-of (Abs τ (f $ Bv 0)) = Some (τ → τ′)*
⟨*proof*⟩

lemma *term-okI*: *term-ok′ (sig Θ) t ⟹ typ-of t ≠ None ⟹ term-ok Θ t*
⟨*proof*⟩
lemma *term-okD1*: *term-ok Θ t ⟹ term-ok′ (sig Θ) t*
⟨*proof*⟩
lemma *term-okD2*: *term-ok Θ t ⟹ typ-of t ≠ None*
⟨*proof*⟩

lemma *term-ok-imp-typ-ok′*: **assumes** *wf-theory Θ term-ok Θ t* **shows** *typ-ok Θ (the (typ-of t))*
⟨*proof*⟩

lemma *term-ok-mk-eqI*:
assumes *wf-theory Θ term-ok Θ s term-ok Θ t typ-of s = typ-of t*
shows*term-ok Θ (mk-eq s t)*
⟨*proof*⟩

lemma *typ-of1-decr′*: *¬ loose-bvar1 t 0 ⟹ typ-of1 (T#Ts) t = Some τ ⟹ typ-of1 Ts (decr 0 t) = Some τ*
⟨*proof*⟩

lemma *typ-of1-eta-red-step-pre*: *¬ loose-bvar1 t 0 ⟹*
typ-of1 Ts (Abs τ (t $ Bv 0)) = Some (τ → τ′) ⟹ typ-of1 Ts (decr 0 t) = Some (τ → τ′)
⟨*proof*⟩

lemma *typ-of1-eta-red-step*: *¬ is-dependent t ⟹*
typ-of (Abs τ (t $ Bv 0)) = Some (τ → τ′) ⟹ typ-of (decr 0 t) = Some (τ → τ′)
⟨*proof*⟩

lemma *distinct-add-vars'*: *distinct acc* $\Longrightarrow$ *distinct* (*add-vars'* *t acc*)
⟨*proof*⟩

lemma *distinct-add-tvarsT'*: *distinct acc* $\Longrightarrow$ *distinct* (*add-tvarsT'* *T acc*)
⟨*proof*⟩

lemma *distinct-add-tvars'*: *distinct acc* $\Longrightarrow$ *distinct* (*add-tvars'* *t acc*)
⟨*proof*⟩

lemma *proved-terms-well-formed-pre*: Θ, $\Gamma \vdash p \Longrightarrow$ *typ-of p* = *Some propT* $\wedge$ *term-ok* Θ *p*
⟨*proof*⟩

corollary *proved-terms-well-formed*:
assumes Θ, $\Gamma \vdash p$
shows *typ-of p* = *Some propT term-ok* Θ *p*
⟨*proof*⟩

lemma *forall-intros*:
wf-theory $\Theta \Longrightarrow \Theta$,$\Gamma \vdash B \Longrightarrow \forall$ (*x*, τ)$\in$*set frees* . (*x*,τ) $\notin$ *FV* Γ $\wedge$ *typ-ok* Θ τ
$\Longrightarrow \Theta$,$\Gamma \vdash$ *mk-all-list frees B*
⟨*proof*⟩

lemma *term-ok-var*[*simp*]: *term-ok* Θ (*Fv idn* τ) = *typ-ok* Θ τ
⟨*proof*⟩
lemma *typ-of-var*[*simp*]: *typ-of* (*Fv idn* τ) = *Some* τ
⟨*proof*⟩

lemma *is-closed-Fv*[*simp*]: *is-closed* (*Fv idn* τ) ⟨*proof*⟩

corollary *proved-terms-closed*: Θ, $\Gamma \vdash B \Longrightarrow$ *is-closed B*
⟨*proof*⟩

lemma *not-loose-bvar-bind-fv2*:
$\neg$ *loose-bvar t lev* $\Longrightarrow \neg$ *loose-bvar* (*bind-fv2 v lev t*) (*Suc lev*)
⟨*proof*⟩
lemma *not-loose-bvar-bind-fv2-*:
$\neg$ *loose-bvar* (*bind-fv2 v lev t*) *lev* $\Longrightarrow \neg$ *loose-bvar t lev*
⟨*proof*⟩

lemma *fold-add-vars'-FV-pre*: *set* (*fold add-vars' Hs acc*) = *set acc* $\cup$ *FV* (*set Hs*)
⟨*proof*⟩
corollary *fold-add-vars'-FV*[*simp*]: *set* (*fold* (*add-vars'*) *Hs* []) = *FV* (*set Hs*)
⟨*proof*⟩

lemma *forall-intro-vars*:
assumes *wf-theory* Θ Θ, *set Hs* $\vdash$ *B*
shows Θ, *set Hs* $\vdash$ *forall-intro-vars B Hs*
⟨*proof*⟩

lemma *mk-all-list'-preserves-term-ok-typ-of*:
assumes *wf-theory* Θ *term-ok* Θ *B typ-of B* = *Some propT* $\forall$ (*idn*,*ty*)$\in$*set vs* . *typ-ok* Θ *ty*
shows *term-ok* Θ (*mk-all-list vs B*) $\wedge$ *typ-of* (*mk-all-list vs B*) = *Some propT*
⟨*proof*⟩

corollary *forall-intro-vars-preserves-term-ok-typ-of*:
assumes *wf-theory* Θ *term-ok* Θ *B typ-of B* = *Some propT*
shows *term-ok* Θ (*forall-intro-vars B Hs*) $\wedge$ *typ-of* (*forall-intro-vars B Hs*) = *Some propT*
⟨*proof*⟩

lemma *bind-fv-remove-var-from-fv*: *fv* (*bind-fv* (*idn*, τ) *t*) = *fv t* − {(*idn*, τ)}
⟨*proof*⟩

lemma *forall-intro-vars-remove-fv*[*simp*]: *fv* (*forall-intro-vars t* []) = {}
⟨*proof*⟩

lemma *term-ok-mk-all-list*:
assumes *wf-theory* Θ
assumes *term-ok* Θ *B*
assumes *typ-of B* = *Some propT*
assumes $\forall$ (*idn*, τ) $\in$ *set l* . *typ-ok* Θ τ
shows *term-ok* Θ (*mk-all-list l B*) $\wedge$ *typ-of* (*mk-all-list l B*) = *Some propT*
⟨*proof*⟩

lemma *tvs-bind-fv2*: *tvs* (*bind-fv2* (*v*, *T*) *lev t*) $\cup$ *tvsT T* = *tvs t* $\cup$ *tvsT T*
⟨*proof*⟩
lemma *tvs-bind-fv*: *tvs* (*bind-fv* (*v*,*T*) *t*) $\cup$ *tvsT T* = *tvs t* $\cup$ *tvsT T*
⟨*proof*⟩

lemma *tvs-mk-all'*: *tvs* (*mk-all idn ty B*) = *tvs B* $\cup$ *tvsT ty*
⟨*proof*⟩

lemma *tvs-mk-all-list*:
tvs (*mk-all-list vs B*) = *tvs B* $\cup$ *tvsT-Set* (*snd* ‘ *set vs*)
⟨*proof*⟩

lemma *tvs-occs*: *occs v t* $\Longrightarrow$ *tvs v* $\subseteq$ *tvs t*
⟨*proof*⟩

lemma *tvs-forall-intro-vars*: *tvs* (*forall-intro-vars B Hs*) = *tvs B*
⟨*proof*⟩

lemma *strip-all-single-var B* = *Some τ* ⟹ *strip-all-single-body B* ≠ *B*
⟨*proof*⟩

lemma *strip-all-body-unchanged-iff-strip-all-single-body-unchanged*:
strip-all-body B = *B* ⟷ *strip-all-single-body B* = *B*
⟨*proof*⟩

lemma *strip-all-body-unchanged-imp-strip-all-vars-no*:
assumes *strip-all-body B* = *B*
shows *strip-all-vars B* = []
⟨*proof*⟩

lemma *strip-all-body-unchanged-imp-strip-all-single-body-unchanged*:
strip-all-body B = *B* ⟹ *strip-all-single-body B* = *B*
⟨*proof*⟩

lemma *strip-all-single-body-unchanged-imp-strip-all-body-unchanged*:
strip-all-single-body B = *B* ⟹ *strip-all-body B* = *B*
⟨*proof*⟩

lemma *strip-all-single-var-np-imp-strip-all-body-single-unchanged*:
strip-all-single-var B = *None* ⟹ *strip-all-single-body B* = *B*
⟨*proof*⟩

lemma *strip-all-single-form*: *strip-all-single-var B* = *Some τ*
⟹ *Ct STR ''Pure.all''* ((*τ* → *propT*) → *propT*) $ *Abs τ* (*strip-all-single-body B*) = *B*
⟨*proof*⟩

lemma *proves-strip-all-single*:
assumes Θ, Γ ⊢ *B strip-all-single-var B* = *Some τ*
typ-of t = *Some τ term-ok* Θ *t*
shows Θ, Γ ⊢ *subst-bv t* (*strip-all-single-body B*)
⟨*proof*⟩

corollary *proves-strip-all-single-Fv*:
assumes Θ, Γ ⊢ *B strip-all-single-var B* = *Some τ*
shows Θ, Γ ⊢ *subst-bv* (*Fv x τ*) (*strip-all-single-body B*)
⟨*proof*⟩

lemma *strip-all-vars-no-strip-all-body-unchanged*[*simp*]:
strip-all-vars B = [] ⟹ *strip-all-body B* = *B*
⟨*proof*⟩

lemma *strip-all-vars B* = (*τs*@[*τ*]) ⟹ *strip-all-body B*
= *strip-all-single-body* (*Ct STR ''Pure.all''* ((*τ* → *propT*) → *propT*) $ *Abs τ*

(*strip-all-body B*))
⟨*proof*⟩

lemma *strip-all-vars-incr-bv*: *strip-all-vars* (*incr-bv inc lev t*) = *strip-all-vars t*
⟨*proof*⟩
lemma *strip-all-vars-incr-boundvars*: *strip-all-vars* (*incr-boundvars inc t*) = *strip-all-vars t*
⟨*proof*⟩

lemma *strip-all-vars-subst-bv1-Fv*:
strip-all-vars (*subst-bv1 B lev* (*Fv x τ*)) = *strip-all-vars B*
⟨*proof*⟩
lemma *strip-all-vars-subst-bv-Fv*:
strip-all-vars (*subst-bv* (*Fv x τ*) *B*) = *strip-all-vars B*
⟨*proof*⟩

lemma *strip-all-single-var B* = *Some τ*
⟹ *strip-all-vars* (*subst-bv* (*Fv x τ*) (*strip-all-single-body B*)) = *tl* (*strip-all-vars B*)
⟨*proof*⟩

corollary *proves-strip-all-vars-Fv*:
assumes *length xs* = *length* (*strip-all-vars B*) Θ, Γ ⊢ *B*
shows Θ, Γ ⊢ *fold* (λ(*x*,*τ*). *subst-bv* (*Fv x τ*) *o strip-all-single-body*)
(*zip xs* (*strip-all-vars B*)) *B*
⟨*proof*⟩

lemma *trivial-pre-depr*: *term-ok* Θ *c* ⟹ *typ-of c* = *Some propT* ⟹ Θ, {*c*} ⊢ *c*
⟨*proof*⟩

lemma *trivial-pre*:
assumes *wf-theory* Θ *term-ok* Θ *c typ-of c* = *Some propT*
shows Θ, {} ⊢ *c* ⟼ *c*
⟨*proof*⟩

lemma *inst-var*:
assumes *wf-theory*: *wf-theory* Θ
assumes *B*: Θ, Γ ⊢ *B*
assumes *a-ok*: *term-ok* Θ *a*
assumes *typ-a*: *typ-of a* = *Some τ*
assumes *free*: (*x*,*τ*) ∉ *FV* Γ
shows Θ, Γ ⊢ *subst-term* [((*x*, *τ*), *a*)] *B*
⟨*proof*⟩

lemma *subst-term-single-no-change*[*simp*]:
assumes *nvar*: (*x*,*τ*)∉*fv B*

shows *subst-term* [((*x*,*τ*), *t*)] *B* = *B*
⟨*proof*⟩

lemma *fv-subst-term-single*:
assumes *var*: (*x*,*τ*)∈*fv B*
assumes ⋀*p* . *p* ∈ *fv t* ⟹ *p* ~= (*x*,*τ*)
shows *fv* (*subst-term* [((*x*,*τ*), *t*)] *B*) = *fv B* − {(*x*,*τ*)} ∪ *fv t*
⟨*proof*⟩

lemma *inst-vars-pre*:
assumes *wf-theory*: *wf-theory* Θ
assumes *B*: Θ, Γ ⊢ *B*

assumes *vars-ok*: *list-all* (*term-ok* Θ) (*map snd insts*)
assumes *typs-ok*: *list-all* (λ((*idx*, *ty*), *t*) . *typ-of t* = *Some ty*) *insts*
assumes *free*: *list-all* (λ((*idx*, *ty*), *t*) . (*idx*, *ty*) ∉ *FV* Γ) *insts*
assumes *typ-a*: *typ-of a* = *Some τ*
assumes *distinct*: *distinct* (*map fst insts*)
assumes *no-overlap*: ⋀*x* . *x* ∈ (⋃*t* ∈ *snd* ‘ (*set insts*) . *fv t*) ⟹ *x* ∉ *fst* ‘ (*set insts*)
shows Θ, Γ ⊢ *fold* (λ*single*. *subst-term* [*single*]) *insts B*
⟨*proof*⟩

lemma *subterm-term-ok′*:
is-std-sig Σ ⟹ *term-ok′* Σ *t* ⟹ *is-closed st* ⟹ *occs st t* ⟹ *term-ok′* Σ *st*
⟨*proof*⟩

lemma *infinite-fv-UNIV*: *infinite* (*UNIV* :: (*indexname* × *typ*) *set*)
⟨*proof*⟩

lemma *implies-intro′-pre*:
assumes *wf-theory* Θ Θ, Γ ⊢ *B term-ok* Θ *A typ-of A* = *Some propT A* ∉ Γ
shows Θ, Γ ⊢ *A* ⟼ *B*
⟨*proof*⟩

lemma *implies-intro′-pre2*:
assumes *wf-theory* Θ Θ, Γ ⊢ *B term-ok* Θ *A typ-of A* = *Some propT A* ∈ Γ
shows Θ, Γ ⊢ *A* ⟼ *B*
⟨*proof*⟩

lemma *subst-term-preserves-typ-of1* [*simp*]:
typ-of1 Ts (*subst-term* [((*x*, *τ*), *Fv y τ*)] *t*) = *typ-of1 Ts t*
⟨*proof*⟩

lemma *subst-term-preserves-typ-of*[*simp*]:
typ-of (*subst-term* [((*x*, *τ*), *Fv y τ*)] *t*) = *typ-of t*
⟨*proof*⟩

lemma *subst-term-preserves-term-ok'*[*simp*]:
term-ok' Σ (*subst-term* [((*x*, *τ*), *Fv y τ*)] *t*) ⟷ *term-ok'* Σ *t*
⟨*proof*⟩

lemma *subst-term-preserves-term-ok*[*simp*]:
term-ok Θ (*subst-term* [((*x*, *τ*), *Fv y τ*)] *A*) ⟷ *term-ok* Θ *A*
⟨*proof*⟩

lemma *not-in-FV-in-fv-not-in*: (*x*,*τ*) ∉ *FV* Γ ⟹ (*x*,*τ*) ∈ *fv t* ⟹ *t* ∉ Γ
⟨*proof*⟩

lemma *subst-term-fv*: *fv* (*subst-term* [((*x*, *τ*), *Fv y τ*)] *t*)
= (*if* (*x*,*τ*) ∈ *fv t then insert* (*y*,*τ*) *else id*) (*fv t* − {(*x*,*τ*)})
⟨*proof*⟩

lemma *rename-free*:
assumes *wf-theory*: *wf-theory* Θ
assumes *B*: Θ, Γ ⊢ *B*
assumes *free*: (*x*,*τ*)∉ *FV* Γ
shows Θ, Γ ⊢ *subst-term* [((*x*, *τ*), *Fv y τ*)] *B*
⟨*proof*⟩

lemma *tvs-subst-term-single*[*simp*]: *tvs* (*subst-term* [((*x*, *τ*), *Fv y τ*)] *A*) = *tvs A*
⟨*proof*⟩

lemma *weaken-proves'*: Θ, Γ ⊢ *B* ⟹ *term-ok* Θ *A* ⟹ *typ-of A* = *Some propT*
⟹ *A* ∉ Γ
⟹ *finite* Γ
⟹ Θ, *insert A* Γ ⊢ *B*
⟨*proof*⟩
corollary *weaken-proves*: Θ, Γ ⊢ *B* ⟹ *term-ok* Θ *A* ⟹ *typ-of A* = *Some propT*
⟹ *finite* Γ
⟹ Θ, *insert A* Γ ⊢ *B*
⟨*proof*⟩

lemma *weaken-proves-set*: *finite* Γ*2* ⟹ Θ, Γ ⊢ *B* ⟹ ∀ *A*∈Γ*2* . *term-ok* Θ *A* ⟹
∀ *A*∈Γ*2* . *typ-of A* = *Some propT*
⟹ *finite* Γ
⟹ Θ, Γ ∪ Γ*2* ⊢ *B*
⟨*proof*⟩

lemma *no-tvsT-imp-subst-typ-unchanged*: *tvsT T* = *empty* ⟹ *subst-typ insts T*
= *T*

⟨*proof*⟩

lemma *subst-typ-fv*:
shows *apsnd* (*subst-typ insts*) ‘ *fv B* = *fv* (*subst-typ′ insts B*)
⟨*proof*⟩

lemma *subst-typ-fv-point*:
assumes (*x*, *τ*) ∈ *fv B*
shows (*x*, *subst-typ insts τ*) ∈ *fv* (*subst-typ′ insts B*)
⟨*proof*⟩

lemma *subst-typ-typ-ok*:
assumes *typ-ok-sig* Σ *τ*
assumes *list-all* (*typ-ok-sig* Σ) (*map snd insts*)
shows *typ-ok-sig* Σ (*subst-typ insts τ*)
⟨*proof*⟩

lemma *subst-typ-comp-single-left*: *subst-typ* [*single*] (*subst-typ insts T*)
= *subst-typ* (*map* (*apsnd* (*subst-typ* [*single*])) *insts*@[*single*]) *T*
⟨*proof*⟩

lemma *subst-typ-comp-single-left-stronger*: *subst-typ* [*single*] (*subst-typ insts T*)
= *subst-typ* (*map* (*apsnd* (*subst-typ* [*single*])) *insts*
@ (*if fst single* ∈ *set* (*map fst insts*) *then* [] *else* [*single*])) *T*
⟨*proof*⟩

lemma *subst-typ′-comp-single-left*: *subst-typ′* [*single*] (*subst-typ′ insts t*)
= *subst-typ′* (*map* (*apsnd* (*subst-typ* [*single*])) *insts*@[*single*]) *t*
⟨*proof*⟩

lemma *subst-typ′-comp-single-left-stronger*: *subst-typ′* [*single*] (*subst-typ′ insts t*)
= *subst-typ′* (*map* (*apsnd* (*subst-typ* [*single*])) *insts*
@ (*if fst single* ∈ *set* (*map fst insts*) *then* [] *else* [*single*])) *t*
⟨*proof*⟩

lemma *subst-typ-preserves-typ-ok*:
assumes *wf-theory* Θ
assumes *typ-ok* Θ *T*
assumes *list-all* (*typ-ok* Θ) (*map snd insts*)
shows *typ-ok* Θ (*subst-typ insts T*)
⟨*proof*⟩

lemma *typ-ok-Ty*[*simp*]: *typ-ok* Θ (*Ty n Ts*) ⟹ *list-all* (*typ-ok* Θ) *Ts*
⟨*proof*⟩
lemma *typ-ok-sig-Ty*[*simp*]: *typ-ok-sig* Σ (*Ty n Ts*) ⟹ *list-all* (*typ-ok-sig* Σ) *Ts*
⟨*proof*⟩

lemma *wf-theory-imp-wf-osig*: *wf-theory* Θ ⟹ *wf-osig* (*osig* (*sig* Θ))
⟨*proof*⟩

lemma *the-lift2-option-Somes*[*simp*]: *the* (*lift2-option f* (*Some a*) (*Some b*)) = *f a b* ⟨*proof*⟩

lemma *class-les-mgd*:
assumes *wf-osig oss*
assumes *tcsigs oss type* = *Some mgd*
assumes *mgd C′* = *Some Ss′*
assumes *class-les* (*subclass oss*) *C′ C*
shows *mgd C* ≠ *None*
⟨*proof*⟩

lemma *has-sort-sort-leq-osig*:
assumes *wf-osig* (*sub*, *tcs*) *has-sort* (*sub*,*tcs*) *T S sort-leq sub S S′*
shows *has-sort* (*sub*,*tcs*) *T S′*
⟨*proof*⟩

lemma *has-sort-sort-leq*: *wf-theory* Θ ⟹ *has-sort* (*osig* (*sig* Θ)) *T S*
⟹ *sort-leq* (*subclass* (*osig* (*sig* Θ))) *S S′*
⟹ *has-sort* (*osig* (*sig* Θ)) *T S′*
⟨*proof*⟩

lemma *subst-typ-preserves-has-sort*:
assumes *wf-theory* Θ
assumes *has-sort* (*osig* (*sig* Θ)) *T S*
assumes *list-all* (λ((*idn*, *S*), *T*). *has-sort* (*osig* (*sig* Θ)) *T S*) *insts*
shows *has-sort* (*osig* (*sig* Θ)) (*subst-typ insts T*) *S*
⟨*proof*⟩

lemma *subst-typ-preserves-Some-typ-of1*:
assumes *typ-of1 Ts t* = *Some T*
shows *typ-of1* (*map* (*subst-typ insts*) *Ts*) (*subst-typ′ insts t*)
= *Some* (*subst-typ insts T*)
⟨*proof*⟩

corollary *subst-typ-preserves-Some-typ-of*:
assumes *typ-of t* = *Some T*
shows *typ-of* (*subst-typ′ insts t*)
= *Some* (*subst-typ insts T*)
⟨*proof*⟩

lemma *subst-typ′-incr-bv*:
subst-typ′ insts (*incr-bv inc lev t*) = *incr-bv inc lev* (*subst-typ′ insts t*)
⟨*proof*⟩

lemma *subst-typ′-incr-boundvars*:
subst-typ′ insts (*incr-boundvars lev t*) = *incr-boundvars lev* (*subst-typ′ insts t*)
⟨*proof*⟩

lemma *subst-typ'-subst-bv1*: *subst-typ' insts* (*subst-bv1 t n u*)
= *subst-bv1* (*subst-typ' insts t*) *n* (*subst-typ' insts u*)
⟨*proof*⟩

lemma *subst-typ'-subst-bv*: *subst-typ' insts* (*subst-bv t u*)
= *subst-bv* (*subst-typ' insts t*) (*subst-typ' insts u*)
⟨*proof*⟩

lemma *subst-typ-no-tvsT-unchanged*:
∀(*f*, *s*) ∈ *set insts* . *f* ∉ *tvsT T* ⟹ *subst-typ insts T* = *T*
⟨*proof*⟩

lemma *subst-typ'-no-tvs-unchanged*:
∀(*f*, *s*) ∈ *set insts* . *f* ∉ *tvs t* ⟹ *subst-typ' insts t* = *t*
⟨*proof*⟩

lemma *subst-typ'-preserves-term-ok'*:
assumes *wf-theory* Θ
assumes *inst-ok* Θ *insts*
assumes *term-ok'* (*sig* Θ) *t*
shows *term-ok'* (*sig* Θ) (*subst-typ' insts t*)
⟨*proof*⟩

lemma *subst-typ'-preserves-term-ok*:
assumes *wf-theory* Θ
assumes *inst-ok* Θ *insts*
assumes *term-ok* Θ *t*
shows *term-ok* Θ (*subst-typ' insts t*)
⟨*proof*⟩

lemma *subst-typ-rename-vars-cancel*:
assumes *y* ∉ *fst* ‘ *tvsT T*
shows *subst-typ* [((*y*,*S*), *Tv x S*)] (*subst-typ* [((*x*,*S*), *Tv y S*)] *T*) = *T*
⟨*proof*⟩

lemma *subst-typ'-rename-tvars-cancel*:
assumes *y* ∉ *fst* ‘ *tvs t* **assumes** *y* ∉ *fst* ‘ *tvsT τ*
shows *subst-typ'* [((*y*,*S*), *Tv x S*)] ((*bind-fv2* (*x*, *subst-typ* [((*x*,*S*), *Tv y S*)] *τ*))
lev (*subst-typ'* [((*x*,*S*), *Tv y S*)] *t*))
= *bind-fv2* (*x*, *τ*) *lev t*
⟨*proof*⟩

lemma *bind-fv2-renamed-var*:
assumes *y* ∉ *fst* ‘ *fv t*
shows *bind-fv2* (*y*, *τ*) *i* (*subst-term* [((*x*, *τ*), *Fv y τ*)] *t*)
= *bind-fv2* (*x*, *τ*) *i t*
⟨*proof*⟩

lemma *bind-fv-renamed-var*:
assumes $y \notin fst\ `\ fv\ t$
shows *bind-fv* (y, τ) (*subst-term* $[((x, \tau), Fv\ y\ \tau)]\ t$)
$= bind\text{-}fv\ (x, \tau)\ t$
⟨*proof*⟩

lemma *subst-typ′-rename-tvar-bind-fv2*:
assumes $y \notin fst\ `\ fv\ t$
assumes $(b, S) \notin tvs\ t$
assumes $(b, S) \notin tvsT\ \tau$
shows *bind-fv2* $(y,$ *subst-typ* $[((a, S), Tv\ b\ S)]\ \tau)\ i$
(*subst-typ′* $[((a,S), Tv\ b\ S)]$ (*subst-term* $[((x, \tau), Fv\ y\ \tau)]\ t$))
$= subst\text{-}typ'\ [((a,S), Tv\ b\ S)]$ (*bind-fv2* $(x, \tau)\ i\ t$)
⟨*proof*⟩

lemma *subst-typ′-rename-tvar-bind-fv*:
assumes $y \notin fst\ `\ fv\ t$
assumes $(b, S) \notin tvs\ t$
assumes $(b, S) \notin tvsT\ \tau$
shows *bind-fv* $(y,$ *subst-typ* $[((a,S), Tv\ b\ S)]\ \tau)$
(*subst-typ′* $[((a,S), Tv\ b\ S)]$ (*subst-term* $[((x, \tau), Fv\ y\ \tau)]\ t$))
$= subst\text{-}typ'\ [((a,S), Tv\ b\ S)]$ (*bind-fv* $(x, \tau)\ t$)
⟨*proof*⟩

lemma *tvar-in-fv-in-tvs*: $(a, \tau) \in fv\ B \Longrightarrow (x, S) \in tvsT\ \tau \Longrightarrow (x, S) \in tvs\ B$
⟨*proof*⟩

lemma *tvs-bind-fv2-subset*: *tvs* (*bind-fv2* $(a, \tau)\ i\ B) \subseteq tvs\ B$
⟨*proof*⟩

lemma *tvs-bind-fv-subset*: *tvs* (*bind-fv* $(a, \tau)\ B) \subseteq tvs\ B$
⟨*proof*⟩

lemma *subst-typ-rename-tvar-preserves-eq*:
$(y, S) \notin tvsT\ T \Longrightarrow (y, S) \notin tvsT\ \tau \Longrightarrow$
subst-typ $[((x, S), Tv\ y\ S)]\ T =$ *subst-typ* $[((x, S), Tv\ y\ S)]\ \tau \Longrightarrow T{=}\tau$
⟨*proof*⟩

lemma *subst-typ′-subst-term-rename-var-swap*:
assumes $b \notin fst\ `\ fv\ B$
assumes $(y, S) \notin tvs\ B$
assumes $(y, S) \notin tvsT\ \tau$
shows *subst-typ′* $[((x, S), Tv\ y\ S)]$ (*subst-term* $[((a, \tau), Fv\ b\ \tau)]\ B$)
$=$ *subst-term* $[((a,$ (*subst-typ* $[((x, S), Tv\ y\ S)]\ \tau)), Fv\ b$ (*subst-typ* $[((x, S), Tv\ y\ S)]\ \tau))]$
(*subst-typ′* $[((x, S), Tv\ y\ S)]\ B$)
⟨*proof*⟩

lemma *tvar-not-in-term-imp-free-not-in-term*:
(*y*, *S*) ∈ *tvsT τ* ⟹ (*y*,*S*) ∉ *tvs t* ⟹ (*a*, *τ*) ∉ *fv t*
⟨*proof*⟩

lemma *tvar-not-in-term-imp-free-not-in-term-set*:
finite Γ ⟹ (*y*, *S*) ∈ *tvsT τ* ⟹ (*y*,*S*) ∉ *tvs-Set* Γ ⟹ (*a*, *τ*) ∉ *FV* Γ
⟨*proof*⟩

lemma *inst-var-multiple*:
assumes *wf-theory*: *wf-theory* Θ
assumes *B*: Θ, Γ ⊢ *B*
assumes *vars*: ∀ (*x*,*τ*)∈*fst* ‘ *set insts* . *term-ok* Θ (*Fv x τ*)
assumes *a-ok*: ∀ *a*∈*snd* ‘ *set insts* . *term-ok* Θ *a*
assumes *typ-a*: ∀ ((-,*τ*), *a*)∈*set insts* . *typ-of a* = *Some τ*
assumes *free*: ∀ (*v*, -)∈*set insts* . *v* ∉ *FV* Γ
assumes *distinct*: *distinct* (*map fst insts*)
assumes *finite*: *finite* Γ
shows Θ, Γ ⊢ *subst-term insts B*
⟨*proof*⟩

lemma *term-ok-eta-red-step*:
¬ *is-dependent t* ⟹ *term-ok* Θ (*Abs T* (*t* $ *Bv 0*)) ⟹ *term-ok* Θ (*decr 0 t*)
⟨*proof*⟩

end

11 Derived rules on equality and normalization

theory *EqualityProof*
imports *Logic*
begin

lemma *proves-eq-reflexive-pre*:
assumes *wf-theory* Θ
assumes *term-ok* Θ *t*
shows Θ, {} ⊢ *mk-eq t t*
⟨*proof*⟩

lemma *unsimp-context*: Γ = {} ∪ Γ
⟨*proof*⟩

lemma *proves-eq-reflexive*:
assumes *wf-theory* Θ
assumes *term-ok* Θ *t*
assumes *finite* Γ ∀ *A*∈Γ. *term-ok* Θ *A* ∀ *A*∈Γ. *typ-of A* = *Some propT*

shows Θ, $\Gamma \vdash$ *mk-eq t t*
⟨*proof*⟩

lemma *proves-eq-symmetric-pre*:
assumes *wf-theory* Θ
assumes *term-ok* Θ *t*
assumes *term-ok* Θ *s*
assumes *typ-of s* = *typ-of t*
shows Θ, {} $\vdash$ *mk-eq s t* $\longmapsto$ *mk-eq t s*
⟨*proof*⟩

lemma *proves-eq-symmetric*:
assumes *wf-theory* Θ
assumes *term-ok* Θ *t*
assumes *term-ok* Θ *s*
assumes *typ-of s* = *typ-of t*
assumes *finite* Γ $\forall A{\in}\Gamma$. *term-ok* Θ *A* $\forall A{\in}\Gamma$. *typ-of A* = *Some propT*
shows Θ, $\Gamma \vdash$ *mk-eq s t* $\longmapsto$ *mk-eq t s*
⟨*proof*⟩

lemma *proves-eq-symmetric2'*:
assumes *wf-theory* Θ
assumes *term-ok* Θ (*mk-eq s t*)
assumes *finite* Γ $\forall A{\in}\Gamma$. *term-ok* Θ *A* $\forall A{\in}\Gamma$. *typ-of A* = *Some propT*
shows Θ, $\Gamma \vdash$ *mk-eq s t* $\longmapsto$ *mk-eq t s*
⟨*proof*⟩

lemma *proves-eq-symmetric-rule*:
assumes *wf-theory* Θ
assumes *term-ok* Θ *t*
assumes *term-ok* Θ *s*
assumes *typ-of s* = *typ-of t*
assumes Θ, $\Gamma \vdash$ *mk-eq s t*
assumes *ctxt*: *finite* Γ $\forall A{\in}\Gamma$. *term-ok* Θ *A* $\forall A{\in}\Gamma$. *typ-of A* = *Some propT*
shows Θ, $\Gamma \vdash$ *mk-eq t s*
⟨*proof*⟩

lemma *proves-eq-transitive-pre*:
assumes *wf-theory* Θ
assumes *term-ok* Θ *s*
assumes *term-ok* Θ *t*
assumes *term-ok* Θ *u*
assumes *typ-of s* = *typ-of t typ-of t* = *typ-of u*
shows Θ, {} $\vdash$ *mk-eq s t* $\longmapsto$ *mk-eq t u* $\longmapsto$ *mk-eq s u*
⟨*proof*⟩

lemma *proves-eq-transitive*:
assumes *wf-theory* Θ
assumes *term-ok* Θ *s*

assumes *term-ok* Θ *t*
assumes *term-ok* Θ *u*
assumes *typ-of s* = *typ-of t typ-of t* = *typ-of u*
assumes *ctxt*: *finite* Γ $\forall A{\in}\Gamma$. *term-ok* Θ *A* $\forall A{\in}\Gamma$. *typ-of A* = *Some propT*
shows Θ, $\Gamma \vdash$ *mk-eq s t* $\longmapsto$ *mk-eq t u* $\longmapsto$ *mk-eq s u*
⟨*proof*⟩

lemma *proves-eq-transitive2*:
assumes *wf-theory* Θ
assumes *term-ok* Θ (*mk-eq s t*)
assumes *term-ok* Θ (*mk-eq t u*)
assumes *ctxt*: *finite* Γ $\forall A{\in}\Gamma$. *term-ok* Θ *A* $\forall A{\in}\Gamma$. *typ-of A* = *Some propT*
shows Θ, $\Gamma \vdash$ *mk-eq s t* $\longmapsto$ *mk-eq t u* $\longmapsto$ *mk-eq s u*
⟨*proof*⟩

lemma *proves-eq-transitive-rule*:
assumes *wf-theory* Θ
assumes *term-ok* Θ *s*
assumes *term-ok* Θ *t*
assumes *term-ok* Θ *u*
assumes *typ-of s* = *typ-of t typ-of t* = *typ-of u*
assumes Θ, $\Gamma \vdash$ *mk-eq s t* Θ, $\Gamma \vdash$ *mk-eq t u*
assumes *ctxt*: *finite* Γ $\forall A{\in}\Gamma$. *term-ok* Θ *A* $\forall A{\in}\Gamma$. *typ-of A* = *Some propT*
shows Θ, $\Gamma \vdash$ *mk-eq s u*
⟨*proof*⟩

lemma *proves-eq-intr-pre*:
assumes *thy*: *wf-theory* Θ
assumes *A*: *term-ok* Θ *A typ-of A* = *Some propT*
assumes *B*: *term-ok* Θ *B typ-of B* = *Some propT*
shows Θ, $\{\} \vdash (A \longmapsto B) \longmapsto (B \longmapsto A) \longmapsto$ *mk-eq A B*
⟨*proof*⟩

lemma *proves-eq-intr*:
assumes *thy*: *wf-theory* Θ
assumes *A*: *term-ok* Θ *A typ-of A* = *Some propT*
assumes *B*: *term-ok* Θ *B typ-of B* = *Some propT*
assumes *ctxt*: *finite* Γ $\forall A{\in}\Gamma$. *term-ok* Θ *A* $\forall A{\in}\Gamma$. *typ-of A* = *Some propT*
shows Θ, $\Gamma \vdash (A \longmapsto B) \longmapsto (B \longmapsto A) \longmapsto$ *mk-eq A B*
⟨*proof*⟩

lemma *proves-eq-intr-rule*:
assumes *thy*: *wf-theory* Θ
assumes *A*: *term-ok* Θ *A typ-of A* = *Some propT*
assumes *B*: *term-ok* Θ *B typ-of B* = *Some propT*
assumes Θ, $\Gamma \vdash (A \longmapsto B)$ Θ, $\Gamma \vdash (B \longmapsto A)$
assumes *ctxt*: *finite* Γ $\forall A{\in}\Gamma$. *term-ok* Θ *A* $\forall A{\in}\Gamma$. *typ-of A* = *Some propT*
shows Θ, $\Gamma \vdash$ *mk-eq A B*
⟨*proof*⟩

lemma *proves-eq-elim-pre*:
assumes *thy*: *wf-theory* Θ
assumes A: *term-ok* Θ A *typ-of* A = *Some propT*
assumes B: *term-ok* Θ B *typ-of* B = *Some propT*
shows Θ, $\{\} \vdash$ *mk-eq* A $B \longmapsto A \longmapsto B$
⟨*proof*⟩

lemma *proves-eq-elim*:
assumes *thy*: *wf-theory* Θ
assumes A: *term-ok* Θ A *typ-of* A = *Some propT*
assumes B: *term-ok* Θ B *typ-of* B = *Some propT*
assumes *ctxt*: *finite* Γ $\forall A{\in}\Gamma$. *term-ok* Θ A $\forall A{\in}\Gamma$. *typ-of* A = *Some propT*
shows Θ, $\Gamma \vdash$ *mk-eq* A $B \longmapsto A \longmapsto B$
⟨*proof*⟩

lemma *proves-eq-elim-rule*:
assumes *thy*: *wf-theory* Θ
assumes A: *term-ok* Θ A *typ-of* A = *Some propT*
assumes B: *term-ok* Θ B *typ-of* B = *Some propT*
assumes Θ, $\Gamma \vdash$ *mk-eq* A B
assumes *ctxt*: *finite* Γ $\forall A{\in}\Gamma$. *term-ok* Θ A $\forall A{\in}\Gamma$. *typ-of* A = *Some propT*
shows Θ, $\Gamma \vdash A \longmapsto B$
⟨*proof*⟩

lemma *proves-eq-elim2-rule*:
assumes *thy*: *wf-theory* Θ
assumes A: *term-ok* Θ A *typ-of* A = *Some propT*
assumes B: *term-ok* Θ B *typ-of* B = *Some propT*
assumes Θ, $\Gamma \vdash$ *mk-eq* A B
assumes *ctxt*: *finite* Γ $\forall A{\in}\Gamma$. *term-ok* Θ A $\forall A{\in}\Gamma$. *typ-of* A = *Some propT*
shows Θ, $\Gamma \vdash B \longmapsto A$
⟨*proof*⟩

lemma *proves-eq-combination-pre*:
assumes *thy*: *wf-theory* Θ
assumes f: *term-ok* Θ f *typ-of* f = *Some* $(\tau \to \tau')$
assumes g: *term-ok* Θ g *typ-of* g = *Some* $(\tau \to \tau')$
assumes x: *term-ok* Θ x *typ-of* x = *Some* τ
assumes y: *term-ok* Θ y *typ-of* y = *Some* τ
shows Θ, $\{\} \vdash$ *mk-eq* f $g \longmapsto$ *mk-eq* x $y \longmapsto$ *mk-eq* $(f \ \$ \ x)$ $(g \ \$ \ y)$
⟨*proof*⟩

lemma *proves-eq-combination*:
assumes *thy*: *wf-theory* Θ
assumes f: *term-ok* Θ f *typ-of* f = *Some* $(\tau \to \tau')$
assumes g: *term-ok* Θ g *typ-of* g = *Some* $(\tau \to \tau')$
assumes x: *term-ok* Θ x *typ-of* x = *Some* τ
assumes y: *term-ok* Θ y *typ-of* y = *Some* τ

assumes *ctxt*: *finite* Γ $\forall A{\in}\Gamma$. *term-ok* Θ A $\forall A{\in}\Gamma$. *typ-of* A = *Some propT*
shows Θ, $\Gamma \vdash$ *mk-eq f g* $\longmapsto$ *mk-eq x y* $\longmapsto$ *mk-eq* (*f* \$ *x*) (*g* \$ *y*)
⟨*proof*⟩

lemma *proves-eq-combination-rule*:
assumes *thy*: *wf-theory* Θ
assumes *f*: *term-ok* Θ *f typ-of f* = *Some* ($\tau \to \tau'$)
assumes *g*: *term-ok* Θ *g typ-of g* = *Some* ($\tau \to \tau'$)
assumes *x*: *term-ok* Θ *x typ-of x* = *Some* τ
assumes *y*: *term-ok* Θ *y typ-of y* = *Some* τ
assumes Θ, $\Gamma \vdash$ *mk-eq f g* Θ, $\Gamma \vdash$ *mk-eq x y*
assumes *ctxt*: *finite* Γ $\forall A{\in}\Gamma$. *term-ok* Θ A $\forall A{\in}\Gamma$. *typ-of* A = *Some propT*
shows Θ, $\Gamma \vdash$ *mk-eq* (*f* \$ *x*) (*g* \$ *y*)
⟨*proof*⟩

lemma *proves-eq-combination-rule-better*:
assumes *thy*: *wf-theory* Θ
assumes Θ, $\Gamma \vdash$ *mk-eq f g* Θ, $\Gamma \vdash$ *mk-eq x y*
assumes *f*: *typ-of f* = *Some* ($\tau \to \tau'$)
assumes *x*: *typ-of x* = *Some* τ
assumes *ctxt*: *finite* Γ $\forall A{\in}\Gamma$. *term-ok* Θ A $\forall A{\in}\Gamma$. *typ-of* A = *Some propT*
shows Θ, $\Gamma \vdash$ *mk-eq* (*f* \$ *x*) (*g* \$ *y*)
⟨*proof*⟩

lemma *proves-eq-mp-rule*:
assumes *thy*: *wf-theory* Θ
assumes *A*: *term-ok* Θ *A typ-of A* = *Some propT*
assumes *B*: *term-ok* Θ *B typ-of B* = *Some propT*
assumes *eq*: Θ, $\Gamma \vdash$ *mk-eq A B*
assumes *pA*: Θ, $\Gamma \vdash$ *A*
assumes *ctxt*: *finite* Γ $\forall A{\in}\Gamma$. *term-ok* Θ A $\forall A{\in}\Gamma$. *typ-of* A = *Some propT*
shows Θ, $\Gamma \vdash$ *B*
⟨*proof*⟩

lemma *proves-eq-mp-rule-better*:
assumes *thy*: *wf-theory* Θ
assumes *eq*: Θ, $\Gamma \vdash$ *mk-eq A B*
assumes *pA*: Θ, $\Gamma \vdash$ *A*
assumes *ctxt*: *finite* Γ $\forall A{\in}\Gamma$. *term-ok* Θ A $\forall A{\in}\Gamma$. *typ-of* A = *Some propT*
shows Θ, $\Gamma \vdash$ *B*
⟨*proof*⟩

lemma *proves-subst-rule*:
assumes *thy*: *wf-theory* Θ
assumes *x*: *term-ok* Θ *x typ-of x* = *Some* τ
assumes *y*: *term-ok* Θ *y typ-of y* = *Some* τ
assumes *P*: *term-ok* Θ *P typ-of P* = *Some* ($\tau \to$ *propT*)
assumes *ctxt*: *finite* Γ $\forall A{\in}\Gamma$. *term-ok* Θ A $\forall A{\in}\Gamma$. *typ-of* A = *Some propT*

assumes *eq*: $\Theta, \Gamma \vdash$ *mk-eq x y*
shows $\Theta, \Gamma \vdash$ *mk-eq* (*P* $ *x*) (*P* $ *y*)
⟨*proof*⟩

lemma *proves-beta-step-rule*:
assumes *thy*: *wf-theory* Θ
assumes *abs*: *term-ok* Θ (*Abs T t*) $\Theta, \Gamma \vdash$ (*Abs T t*) $ *x*
assumes *x*: *term-ok* Θ *x typ-of x* = *Some T*
assumes *ctxt*: *finite* Γ $\forall A{\in}\Gamma$. *term-ok* Θ *A* $\forall A{\in}\Gamma$. *typ-of A* = *Some propT*
shows $\Theta, \Gamma \vdash$ *subst-bv x t*
⟨*proof*⟩

lemma *proves-add-param-rule*:
assumes *thy*: *wf-theory* Θ
assumes *ctxt*: *finite* Γ
assumes *eq*: $\Theta, \Gamma \vdash$ *mk-eq f g typ-of f* = *Some* ($\tau \to \tau'$)
assumes *type*: *typ-ok* Θ τ
assumes *ctxt*: *finite* Γ $\forall A{\in}\Gamma$. *term-ok* Θ *A* $\forall A{\in}\Gamma$. *typ-of A* = *Some propT*
shows $\Theta, \Gamma \vdash$ (*Ct STR ''Pure.all''* (($\tau \to$ *propT*) $\to$ *propT*) $
(*Abs* τ (*mk-eq'* τ' (*f* $ *Bv 0*) (*g* $ *Bv 0*))))
⟨*proof*⟩

lemma *proves-add-abs-rule*:
assumes *thy*: *wf-theory* Θ
assumes *ctxt*: *finite* Γ
assumes *eq*: $\Theta, \Gamma \vdash$ *mk-eq f g typ-of f* = *Some* ($\tau \to \tau'$)
assumes *type*: *typ-ok* Θ τ
assumes *ctxt*: *finite* Γ $\forall A{\in}\Gamma$. *term-ok* Θ *A* $\forall A{\in}\Gamma$. *typ-of A* = *Some propT*
shows $\Theta, \Gamma \vdash$ *mk-eq* (*Abs* τ (*f* $ *Bv 0*)) (*Abs* τ (*g* $ *Bv 0*))
⟨*proof*⟩

lemma *proves-inst-bound-rule*:
assumes *thy*: *wf-theory* Θ
assumes *ctxt*: *finite* Γ $\forall A{\in}\Gamma$. *term-ok* Θ *A* $\forall A{\in}\Gamma$. *typ-of A* = *Some propT*
assumes *eq*: $\Theta, \Gamma \vdash$ *mk-eq* (*Abs* τ *f*) (*Abs* τ *g*) *typ-of* (*Abs* τ *f*) = *Some* ($\tau \to \tau'$)
assumes *x*: *term-ok* Θ *x typ-of x* = *Some* τ
assumes *ctxt*: *finite* Γ $\forall A{\in}\Gamma$. *term-ok* Θ *A* $\forall A{\in}\Gamma$. *typ-of A* = *Some propT*
shows $\Theta, \Gamma \vdash$ *mk-eq* (*subst-bv x f*) (*subst-bv x g*)
⟨*proof*⟩

lemma *proves-descend-abs-rule*:
assumes *thy*: *wf-theory* Θ
assumes *eq*: $\Theta, \Gamma \vdash$ *mk-eq* (*Abs* τ' (*bind-fv* (x, τ') *s*)) (*Abs* τ' (*bind-fv* (x, τ') *t*))
is-closed s is-closed t
assumes *x*: (x, τ') $\notin$ *FV* Γ *typ-ok* Θ τ'

assumes *ctxt*: *finite* Γ $\forall A{\in}\Gamma$. *term-ok* Θ A $\forall A{\in}\Gamma$. *typ-of* A = *Some propT*
shows Θ, $\Gamma \vdash$ *mk-eq s t*
⟨*proof*⟩

lemma *obtain-fresh-variable*:
assumes *finite* Γ
obtains x **where** $(x,\tau) \notin$ *fv t* $\cup$ *FV* Γ
⟨*proof*⟩
lemma *obtain-fresh-variable'*:
assumes *finite* Γ
obtains x **where** $(x,\tau) \notin$ *fv t* $\cup$ *fv u* $\cup$ *FV* Γ
⟨*proof*⟩

lemma *proves-eq-abstract-rule-pre*:
assumes *thy*: *wf-theory* Θ
assumes *A*: *term-ok* Θ *f typ-of f* = *Some* $(\tau \to \tau')$
assumes *B*: *term-ok* Θ *g typ-of g* = *Some* $(\tau \to \tau')$
shows Θ, $\{\} \vdash$ (*Ct STR ''Pure.all''* $((\tau \to propT) \to propT)$ $ *Abs* τ (*mk-eq'* τ' (*f* $ *Bv 0*) (*g* $ *Bv 0*)))
$\longmapsto$ *mk-eq* (*Abs* τ (*f* $ *Bv 0*)) (*Abs* τ (*g* $ *Bv 0*))
⟨*proof*⟩

lemma *proves-eq-abstract-rule*:
assumes *thy*: *wf-theory* Θ
assumes *A*: *term-ok* Θ *f typ-of f* = *Some* $(\tau \to \tau')$
assumes *B*: *term-ok* Θ *g typ-of g* = *Some* $(\tau \to \tau')$
assumes *ctxt*: *finite* Γ $\forall A{\in}\Gamma$. *term-ok* Θ A $\forall A{\in}\Gamma$. *typ-of* A = *Some propT*
shows Θ, $\Gamma \vdash$ (*Ct STR ''Pure.all''* $((\tau \to propT) \to propT)$ $ *Abs* τ (*mk-eq'* τ' (*f* $ *Bv 0*) (*g* $ *Bv 0*)))
$\longmapsto$ *mk-eq* (*Abs* τ (*f* $ *Bv 0*)) (*Abs* τ (*g* $ *Bv 0*))
⟨*proof*⟩

lemma *proves-eq-abstract-rule-rule*:
assumes *thy*: *wf-theory* Θ
assumes *A*: *term-ok* Θ *f typ-of f* = *Some* $(\tau \to \tau')$
assumes *B*: *term-ok* Θ *g typ-of g* = *Some* $(\tau \to \tau')$
assumes Θ, $\Gamma \vdash$ (*Ct STR ''Pure.all''* $((\tau \to propT) \to propT)$ $ *Abs* τ (*mk-eq'* τ' (*f* $ *Bv 0*) (*g* $ *Bv 0*)))
assumes *ctxt*: *finite* Γ $\forall A{\in}\Gamma$. *term-ok* Θ A $\forall A{\in}\Gamma$. *typ-of* A = *Some propT*
shows Θ, $\Gamma \vdash$ *mk-eq* (*Abs* τ (*f* $ *Bv 0*)) (*Abs* τ (*g* $ *Bv 0*))
⟨*proof*⟩

lemma *proves-eq-ext-rule*:
assumes *thy*: *wf-theory* Θ
assumes *f*: *term-ok* Θ *f typ-of f* = *Some* $(\tau \to \tau')$
assumes *g*: *term-ok* Θ *g typ-of g* = *Some* $(\tau \to \tau')$
assumes *prem*: Θ, $\Gamma \vdash$ *Ct STR ''Pure.all''* $((\tau \to propT) \to propT)$ $ *Abs* τ (*mk-eq'* τ' (*f* $ *Bv 0*) (*g* $ *Bv 0*))

assumes *ctxt*: *finite* Γ $\forall A{\in}\Gamma$. *term-ok* Θ A $\forall A{\in}\Gamma$. *typ-of* A = *Some propT*
shows Θ, $\Gamma \vdash$ *mk-eq f g*
⟨*proof*⟩

lemma *bind-fv2-idem*[*simp*]:
bind-fv2 (x, τ) *lev1* (*bind-fv2* (x, τ) *lev2 t*) = *bind-fv2* (x, τ) *lev2 t*
⟨*proof*⟩
corollary *bind-fv-idem*[*simp*]:
bind-fv (x, τ) (*bind-fv* (x, τ) *t*) = *bind-fv* (x, τ) *t*
⟨*proof*⟩
corollary *bind-fv-Abs-fv*[*simp*]: *bind-fv* (x, τ) (*Abs-fv x* τ *t*) = *Abs-fv x* τ *t*
⟨*proof*⟩

lemma *bind-fv2* (x,τ) *lev* (*mk-eq'* τ' *s t*) = *mk-eq'* τ' (*bind-fv2* (x,τ) *lev s*) (*bind-fv2* (x,τ) *lev t*)
⟨*proof*⟩
lemma *bind-fv* (x,τ) (*mk-eq'* τ' *s t*) = *mk-eq'* τ' (*bind-fv* (x,τ) *s*) (*bind-fv* (x,τ) *t*)
⟨*proof*⟩

lemma *term-ok-Abs-fvI*: *term-ok* Θ *s* $\Longrightarrow$ *typ-ok* Θ τ $\Longrightarrow$ *term-ok* Θ (*Abs-fv x* τ *s*)
⟨*proof*⟩

lemma *proves-eq-abstract-rule-derived-rule*:
assumes *thy*: *wf-theory* Θ
assumes *x*: $(x, \tau) \notin$ *FV* Γ *typ-ok* Θ τ
assumes *ctxt*: *finite* Γ $\forall A{\in}\Gamma$. *term-ok* Θ A $\forall A{\in}\Gamma$. *typ-of* A = *Some propT*
assumes *eq*: Θ, $\Gamma \vdash$ *mk-eq s t*
shows Θ, $\Gamma \vdash$ *mk-eq* (*Abs* τ (*bind-fv* (x, τ) *s*)) (*Abs* τ (*bind-fv* (x, τ) *t*))
⟨*proof*⟩

lemma *proves-descend-abs-rule-iff*:
assumes *thy*: *wf-theory* Θ
assumes *ok*: *is-closed s is-closed t*
assumes *x*: $(x, \tau') \notin$ *FV* Γ *typ-ok* Θ τ'
assumes *ctxt*: *finite* Γ $\forall A{\in}\Gamma$. *term-ok* Θ A $\forall A{\in}\Gamma$. *typ-of* A = *Some propT*
shows Θ, $\Gamma \vdash$ *mk-eq s t*
$\longleftrightarrow$ Θ, $\Gamma \vdash$ *mk-eq* (*Abs* τ' (*bind-fv* (x, τ') *s*)) (*Abs* τ' (*bind-fv* (x, τ') *t*))
⟨*proof*⟩

lemma *proves-descend-abs-rule'*:
assumes *thy*: *wf-theory* Θ
assumes *eq*: Θ, $\Gamma \vdash$ *mk-eq* (*Abs* τ' *s*) (*Abs* τ' *t*)
assumes *x*: $(x, \tau') \notin$ *FV* Γ *typ-ok* Θ τ'
assumes *ctxt*: *finite* Γ $\forall A{\in}\Gamma$. *term-ok* Θ A $\forall A{\in}\Gamma$. *typ-of* A = *Some propT*
shows Θ, $\Gamma \vdash$ *mk-eq* (*subst-bv* (*Fv x* τ') *s*) (*subst-bv* (*Fv x* τ') *t*)
⟨*proof*⟩

lemma *proves-ascend-abs-rule'*:
assumes *thy*: *wf-theory* Θ
assumes *x*: $(x, \tau') \notin FV\ \Gamma\ (x,\tau') \notin fv\ (mk\text{-}eq\ (Abs\ \tau'\ s)\ (Abs\ \tau'\ t))\ typ\text{-}ok\ \Theta\ \tau'$
assumes *eq*: $\Theta, \Gamma \vdash mk\text{-}eq\ (subst\text{-}bv\ (Fv\ x\ \tau')\ s)\ (subst\text{-}bv\ (Fv\ x\ \tau')\ t)$
assumes *ctxt*: *finite* $\Gamma\ \forall A{\in}\Gamma.\ term\text{-}ok\ \Theta\ A\ \forall A{\in}\Gamma.\ typ\text{-}of\ A = Some\ propT$
shows $\Theta, \Gamma \vdash mk\text{-}eq\ (Abs\ \tau'\ s)\ (Abs\ \tau'\ t)$
⟨*proof*⟩

lemma *proves-descend-abs-rule-iff'*:
assumes *thy*: *wf-theory* Θ
assumes *x*: $(x, \tau') \notin FV\ \Gamma\ (x, \tau') \notin fv\ (mk\text{-}eq\ (Abs\ \tau'\ s)\ (Abs\ \tau'\ t))\ typ\text{-}ok\ \Theta\ \tau'$
assumes *ctxt*: *finite* $\Gamma\ \forall A{\in}\Gamma.\ term\text{-}ok\ \Theta\ A\ \forall A{\in}\Gamma.\ typ\text{-}of\ A = Some\ propT$
shows $\Theta, \Gamma \vdash mk\text{-}eq\ (subst\text{-}bv\ (Fv\ x\ \tau')\ s)\ (subst\text{-}bv\ (Fv\ x\ \tau')\ t)$
$\longleftrightarrow \Theta, \Gamma \vdash mk\text{-}eq\ (Abs\ \tau'\ s)\ (Abs\ \tau'\ t)$
⟨*proof*⟩

lemma *proves-beta-step-pre*:
assumes *thy*: *wf-theory* Θ
assumes *finite*: *finite* Γ
assumes *free*: $\forall (x,\tau) \in set\ vs\ .\ (x,\tau) \notin fv\ t \cup FV\ \Gamma$
assumes *term-ok'*: $term\text{-}ok\ \Theta\ (subst\text{-}bvs\ (map\ (case\text{-}prod\ Fv)\ vs)\ t)$
assumes *beta*: $t \rightarrow_\beta u$
assumes *ctxt*: $\forall A{\in}\Gamma.\ term\text{-}ok\ \Theta\ A\ \forall A{\in}\Gamma.\ typ\text{-}of\ A = Some\ propT$
shows $\Theta, \Gamma \vdash mk\text{-}eq$
$(subst\text{-}bvs\ (map\ (case\text{-}prod\ Fv)\ vs)\ t)$
$(subst\text{-}bvs\ (map\ (case\text{-}prod\ Fv)\ vs)\ u)$
⟨*proof*⟩

lemma *subst-bvs-empty*[*simp*]: $subst\text{-}bvs\ []\ t = t$
⟨*proof*⟩

lemma *proves-beta-step*:
assumes *thy*: *wf-theory* Θ
assumes *finite*: *finite* Γ
assumes *term-ok*: $term\text{-}ok\ \Theta\ t$
assumes *beta*: $t \rightarrow_\beta u$
assumes *ctxt*: $\forall A{\in}\Gamma.\ term\text{-}ok\ \Theta\ A\ \forall A{\in}\Gamma.\ typ\text{-}of\ A = Some\ propT$
shows $\Theta, \Gamma \vdash mk\text{-}eq\ t\ u$
⟨*proof*⟩

lemma *proves-beta-steps*:
assumes *thy*: *wf-theory* Θ
assumes *finite*: *finite* Γ
assumes *term-ok*: $term\text{-}ok\ \Theta\ t$
assumes *beta*: $t \rightarrow_\beta^* u$
assumes *ctxt*: $\forall A{\in}\Gamma.\ term\text{-}ok\ \Theta\ A\ \forall A{\in}\Gamma.\ typ\text{-}of\ A = Some\ propT$
shows $\Theta, \Gamma \vdash mk\text{-}eq\ t\ u$

⟨*proof*⟩

lemma *proves-beta-norm*:
assumes *thy*: *wf-theory* Θ
assumes *finite*: *finite* Γ
assumes *term-ok*: *term-ok* Θ *t*
assumes *beta*: *beta-norm t* = *Some u*
assumes *ctxt*: ∀ *A*∈Γ. *term-ok* Θ *A* ∀ *A*∈Γ. *typ-of A* = *Some propT*
shows Θ, Γ ⊢ *mk-eq t u*
⟨*proof*⟩

lemma *beta-norm-preserves-proves*:
assumes *thy*: *wf-theory* Θ
assumes *finite*: *finite* Γ
assumes *term-ok*: Θ, Γ ⊢ *t*
assumes *beta*: *beta-norm t* = *Some u*
assumes *ctxt*: ∀ *A*∈Γ. *term-ok* Θ *A* ∀ *A*∈Γ. *typ-of A* = *Some propT*
shows Θ, Γ ⊢ *u*
⟨*proof*⟩

lemma *proves-eta-step-pre*:
assumes *thy*: *wf-theory* Θ
assumes *finite*: *finite* Γ
assumes *free*: ∀ (*x*,τ) ∈ *set vs* . (*x*,τ) ∉ *fv t* ∪ *FV* Γ
assumes *term-ok′*: *term-ok* Θ (*subst-bvs* (*map* (*case-prod Fv*) *vs*) *t*)
assumes *eta*: *t* →$_\eta$ *u*
assumes *ctxt*: ∀ *A*∈Γ. *term-ok* Θ *A* ∀ *A*∈Γ. *typ-of A* = *Some propT*
shows Θ, Γ ⊢ *mk-eq*
(*subst-bvs* (*map* (*case-prod Fv*) *vs*) *t*)
(*subst-bvs* (*map* (*case-prod Fv*) *vs*) *u*)
⟨*proof*⟩

lemma *proves-eta-step*:
assumes *thy*: *wf-theory* Θ
assumes *finite*: *finite* Γ
assumes *term-ok*: *term-ok* Θ *t*
assumes *eta*: *t* →$_\eta$ *u*
assumes *ctxt*: ∀ *A*∈Γ. *term-ok* Θ *A* ∀ *A*∈Γ. *typ-of A* = *Some propT*
shows Θ, Γ ⊢ *mk-eq t u*
⟨*proof*⟩

lemma *proves-eta-steps*:
assumes *thy*: *wf-theory* Θ
assumes *finite*: *finite* Γ
assumes *term-ok*: *term-ok* Θ *t*
assumes *eta*: *t* →$_\eta^*$ *u*
assumes *ctxt*: ∀ *A*∈Γ. *term-ok* Θ *A* ∀ *A*∈Γ. *typ-of A* = *Some propT*
shows Θ, Γ ⊢ *mk-eq t u*
⟨*proof*⟩

lemma *proves-eta-norm*:
assumes *thy*: *wf-theory* Θ
assumes *finite*: *finite* Γ
assumes *term-ok*: *term-ok* Θ *t*
assumes *eta*: *eta-norm t* = *u*
assumes *ctxt*: $\forall A{\in}\Gamma$. *term-ok* Θ *A* $\forall A{\in}\Gamma$. *typ-of A* = *Some propT*
shows Θ, $\Gamma \vdash$ *mk-eq t u*
⟨*proof*⟩

lemma *eta-norm-preserves-proves*:
assumes *thy*: *wf-theory* Θ
assumes *finite*: *finite* Γ
assumes *term-ok*: Θ, $\Gamma \vdash$ *t*
assumes *eta*: *eta-norm t* = *u*
assumes *ctxt*: $\forall A{\in}\Gamma$. *term-ok* Θ *A* $\forall A{\in}\Gamma$. *typ-of A* = *Some propT*
shows Θ, $\Gamma \vdash$ *u*
⟨*proof*⟩

lemma *beta-eta-norm-preserves-proves*:
assumes *thy*: *wf-theory* Θ
assumes *finite*: *finite* Γ
assumes *term-ok*: Θ, $\Gamma \vdash$ *t*
assumes *beta-eta*: *beta-eta-norm t* = *Some u*
assumes *ctxt*: $\forall A{\in}\Gamma$. *term-ok* Θ *A* $\forall A{\in}\Gamma$. *typ-of A* = *Some propT*
shows Θ, $\Gamma \vdash$ *u*
⟨*proof*⟩

lemma *forall-elim'*:
assumes *thy*: *wf-theory* Θ
assumes *all*: Θ, $\Gamma \vdash$ *Ct STR ''Pure.all''* $((\tau \to propT) \to propT)$ $ *B*
assumes *a*: *has-typ a* τ *wf-term* (*sig* Θ) *a*
assumes *ctxt*: *finite* Γ $\forall A{\in}\Gamma$. *term-ok* Θ *A* $\forall A{\in}\Gamma$. *typ-of A* = *Some propT*
shows Θ, $\Gamma \vdash$ *B* · *a*
⟨*proof*⟩
end

12 Proof Terms and proof checker

theory *ProofTerm*
imports *Term Logic Term-Subst SortConstants EqualityProof*
begin

type-synonym *tyinst* = (*variable* × *sort*) × *typ*
type-synonym *tinst* = (*variable* × *typ*) × *term*

datatype *proofterm* = *PAxm term tyinst list*
| *PBound nat*

```
  | Abst typ proofterm
  | AbsP term proofterm
  | Appt proofterm term
  | AppP proofterm proofterm
  | OfClass typ class
  | Hyp term


fun depth :: proofterm ⇒ nat where
  depth (Abst - P) = Suc (depth P)
| depth (AbsP - P) = Suc (depth P)
| depth (Appt P -) = Suc (depth P)
| depth (AppP P1 P2) = Suc (max (depth P1) (depth P2))
| depth - = 1
fun size :: proofterm ⇒ nat where
  size (Abst - P) = Suc (size P)
| size (AbsP - P) = Suc (size P)
| size (Appt P -) = Suc (size P)
| size (AppP P1 P2) = Suc (size P1 + size P2)
| size - = 1

lemma depth P > 0
  ⟨proof⟩
lemma size P > 0
  ⟨proof⟩
lemma size P ≥ depth P
  ⟨proof⟩

fun partial-nth :: 'a list ⇒ nat ⇒ 'a option where
  partial-nth [] - = None
| partial-nth (x#xs) 0 = Some x
| partial-nth (x#xs) (Suc n) = partial-nth xs n

definition [simp]: partial-nth' xs n ≡ if n < length xs then Some (nth xs n) else
None

lemma partial-nth xs n ≡ partial-nth' xs n
  ⟨proof⟩

lemma partial-nth-Some-imp-elem: partial-nth l n = Some x ⟹ x∈set l
  ⟨proof⟩
```

The core of the proof checker

```
fun replay' :: theory ⇒ (variable × typ) list ⇒ variable set
  ⇒ term list ⇒ proofterm ⇒ term option where
  replay' thy - - Hs (PAxm t Tis) = (if inst-ok thy Tis ∧ term-ok thy t
    then if t ∈ axioms thy
      then Some (forall-intro-vars (subst-typ' Tis t) [])
    else None else None)
```

```
| replay' thy - - Hs (PBound n) = partial-nth Hs n
| replay' thy vs ns Hs (Abst T p) = (if typ-ok thy T
    then (let (s',ns') = variant-variable (Free STR ''default'') ns in
      map-option (mk-all s' T) (replay' thy ((s', T) # vs) ns' Hs p))
    else None)
| replay' thy vs ns Hs (Appt p t) =
    (let rep = replay' thy vs ns Hs p in
    let t' = subst-bvs (map (λ(x,y) . Fv x y) vs) t in
      case (rep, typ-of t') of
        (Some (Ct s (Ty fun1 [Ty fun2 [τ, Ty propT1 Nil], Ty propT2 Nil]) $ b),
Some τ') ⇒
          if s = STR ''Pure.all'' ∧ fun1 = STR ''fun'' ∧ fun2 = STR ''fun''
            ∧ propT1 = STR ''prop'' ∧ propT2 = STR ''prop''
             ∧ τ=τ' ∧ term-ok thy t'
            then Some (b · t') else None
      | - ⇒ None)
| replay' thy vs ns Hs (AbsP t p) =
    (let t' = subst-bvs (map (λ(x,y) . Fv x y) vs) t in
    let rep = replay' thy vs ns (t'#Hs) p in
      (if typ-of t' = Some propT ∧ term-ok thy t' then map-option (mk-imp t') rep
else None))
| replay' thy vs ns Hs (AppP p1 p2) =
    (let rep1 = Option.bind (replay' thy vs ns Hs p1) beta-eta-norm in
    let rep2 = Option.bind (replay' thy vs ns Hs p2) beta-eta-norm in
      (case (rep1, rep2) of (
        Some (Ct imp (Ty fn1 [Ty prp1 [], Ty fn2 [Ty prp2 [], Ty prp3 []]]) $ A $
B),
        Some A') ⇒
          if imp = STR ''Pure.imp'' ∧ fn1 = STR ''fun'' ∧ fn2 = STR ''fun''
            ∧ prp1 = STR ''prop'' ∧ prp2 = STR ''prop'' ∧ prp3 = STR ''prop'' ∧
A=A'
          then Some B else None
        | - ⇒ None))
| replay' thy vs ns Hs (OfClass ty c) = (if has-sort (osig (sig thy)) ty {c}
    ∧ typ-ok thy ty
    then (case const-type (sig thy) (const-of-class c) of
      Some (Ty fun [Ty it [ity], Ty prop []]) ⇒
        if ity = tvariable STR '''a'' ∧ fun = STR ''fun'' ∧ prop = STR ''prop'' ∧
it = STR ''itself''
          then Some (mk-of-class ty c) else None | - ⇒ None) else None)
| replay' thy vs ns Hs (Hyp t) = (if t∈set Hs then Some t else None)
```

lemma *fv-subst-bv1*:
fv (subst-bv1 t lev u) = fv t ∪ (if loose-bvar1 t lev then fv u else {})
⟨*proof*⟩

corollary *fv-subst-bvs-upper-bound*:
assumes *is-closed t*

shows *fv* (*subst-bvs us t*) ⊆ *fv t* ∪ (⋃ *x*∈*set us* . (*fv x*))
⟨*proof*⟩

lemma *fv-subst-bvs1-upper-bound*:
fv (*subst-bvs1 t lev us*) ⊆ *fv t* ∪ (⋃ *x*∈*set us* . (*fv x*))
⟨*proof*⟩

lemma *typ-of-axiom*: *wf-theory thy* ⟹ *t* ∈ *axioms thy* ⟹ *typ-of t* = *Some propT*

⟨*proof*⟩

fun *fv-Proof* :: *proofterm* ⇒ (*variable* × *typ*) *set* **where**
fv-Proof (*PAxm t* -) = *fv t*
| *fv-Proof* (*PBound* -) = *empty*
| *fv-Proof* (*Abst* - *p*) = *fv-Proof p*
| *fv-Proof* (*AbsP t p*) = *fv t* ∪ *fv-Proof p*
| *fv-Proof* (*Appt p t*) = *fv-Proof p* ∪ *fv t*
| *fv-Proof* (*AppP p1 p2*) = *fv-Proof p1* ∪ *fv-Proof p2*
| *fv-Proof* (*OfClass* - -) = *empty*
| *fv-Proof* (*Hyp t*) = *fv t*

lemma *typ-ok-Tv*[*simp*]: *typ-ok thy* (*Tv idn S*) = *wf-sort* (*subclass* (*osig* (*sig thy*))) *S*
⟨*proof*⟩

lemma *typ-ok-contained-tvars-typ-ok*: *typ-ok thy ty* ⟹ (*idn*, *S*) ∈ *tvsT ty* ⟹ *typ-ok thy* (*Tv idn S*)
⟨*proof*⟩

lemma *typ-ok-sig-contained-tvars-typ-ok-sig*:
typ-ok-sig Σ *ty* ⟹ (*idn*, *S*) ∈ *tvsT ty* ⟹ *typ-ok-sig* Σ (*Tv idn S*)
⟨*proof*⟩

lemma *term-ok′-contained-tvars-typ-ok-sig*:
term-ok′ Σ *t* ⟹ (*idn*, *S*) ∈ *tvs t* ⟹ *typ-ok-sig* Σ (*Tv idn S*)

⟨*proof*⟩

lemma *term-ok-contained-tvars-typ-ok*:
term-ok thy t ⟹ (*idn*, *S*) ∈ *tvs t* ⟹ *typ-ok thy* (*Tv idn S*)
⟨*proof*⟩

lemma *typ-ok-subst-typ*:
typ-ok thy T ⟹ ∀ (-, *ty*) ∈ *set insts* . *typ-ok thy ty* ⟹ *typ-ok thy* (*subst-typ insts T*)
⟨*proof*⟩

lemma *typ-ok-sig-subst-typ*:
typ-ok-sig Σ *T* ⟹ ∀ (-, *ty*) ∈ *set insts* . *typ-ok-sig* Σ *ty* ⟹ *typ-ok-sig* Σ (*subst-typ*

insts T)
⟨*proof*⟩

lemma *typ-ok-sig-imp-sortsT-ok-sig*: *typ-ok-sig* Σ *T* ⟹ *S* ∈ *SortsT T* ⟹ *wf-sort* (*subclass* (*osig* Σ)) *S*
⟨*proof*⟩

lemma *term-ok'-imp-Sorts-ok-sig*: *term-ok'* Σ *t* ⟹ *S* ∈ *Sorts t* ⟹ *wf-sort* (*subclass* (*osig* Σ)) *S*
⟨*proof*⟩

lemma *replay'-sound-pre*:
assumes *thy*: *wf-theory thy*

assumes *HS-invs*:
⋀*x*. *x*∈*set Hs* ⟹ *term-ok thy x*
⋀*x*. *x*∈*set Hs* ⟹ *typ-of x* = *Some propT*

assumes *ns-invs*:
finite ns
fst ‘ *FV* (*set Hs*) ⊆ *ns*
fst ‘ *fv-Proof P* ⊆ *ns*

assumes *vs-invs*:
fst ‘ *set vs* ⊆ *ns*

assumes *replay' thy vs ns Hs P* = *Some res*
shows *thy*, (*set Hs*) ⊢ *res*
⟨*proof*⟩

lemma *finite-fv-Proof*: *finite* (*fv-Proof P*)
⟨*proof*⟩

abbreviation *replay'' thy vs ns Hs P* ≡ *Option.bind* (*replay' thy vs ns Hs P*) *beta-eta-norm*

lemma *replay''-sound*:
assumes *wf-theory thy*

assumes *HS-invs*:
⋀*x*. *x*∈*set Hs* ⟹ *term-ok thy x*
⋀*x*. *x*∈*set Hs* ⟹ *typ-of x* = *Some propT*

assumes *ns-invs*:
finite ns
fst ‘ *FV* (*set Hs*) ⊆ *ns*
fst ‘ *fv-Proof P* ⊆ *ns*

assumes *vs-invs*:
fst ‘ set vs ⊆ *ns*

assumes *replay''* *thy vs ns Hs P* = *Some res*
shows *thy*, (*set Hs*) ⊢ *res*
⟨*proof*⟩

lemma
assumes *wf-theory thy*
assumes *replay''* *thy* [] (*fst ‘ fv-Proof P*) [] *P* = *Some res*
shows *thy*, *set* [] ⊢ *res*
⟨*proof*⟩

fun *hyps* :: *proofterm* ⇒ *term list* **where**
hyps (*Abst* - *p*) = *hyps p*
| *hyps* (*AbsP* - *p*) = *hyps p*
| *hyps* (*Appt p* -) = *hyps p*
| *hyps* (*AppP p1 p2*) = *List.union* (*hyps p1*) (*hyps p2*)
| *hyps* (*Hyp t*) = [*t*]
| *hyps* - = []

lemma *replay''-sound-pre-hyps*:
assumes *wf-theory thy*

assumes ⋀*x*. *x* ∈ *set* (*hyps P*) ⟹ *term-ok thy x*
assumes ⋀*x*. *x* ∈ *set* (*hyps P*) ⟹ *typ-of x* = *Some propT*
assumes *replay''* *thy* [] (*fst ‘* (*fv-Proof P* ∪ *FV* (*set* (*hyps P*)))) (*hyps P*) *P* = *Some res*
shows *thy*, *set* (*hyps P*) ⊢ *res*
⟨*proof*⟩

definition [*simp*]: *replay thy P* ≡
(*if* ∀ *x*∈*set* (*hyps P*) . *term-ok thy x* ∧ *typ-of x* = *Some propT then*
replay'' *thy* [] (*fst ‘* (*fv-Proof P* ∪ *FV* (*set* (*hyps P*)))) (*hyps P*) *P else None*)

lemma *replay-sound-pre-hyps*:
assumes *wf-theory thy*
assumes *replay thy P* = *Some res*
shows *thy*, *set* (*hyps P*) ⊢ *res*
⟨*proof*⟩

definition *check-proof thy P res* ≡ *wf-theory thy* ∧ *replay thy P* = *Some res*

lemma *check-proof-sound*:
shows *check-proof thy P res* ⟹ *thy*, *set* (*hyps P*) ⊢ *res*
⟨*proof*⟩

lemma *check-proof-really-sound*:
  assumes *check-proof thy P res*
  shows *thy*, *set* (*hyps P*) ⊩ *res*
⟨*proof*⟩

end

13 Executable Sorts

theory *SortsExe*
  imports *Sorts*
begin

type-synonym *exeosig* = (*class* × *class*) *list* × (*name* × (*class* × *sort list*) *list*) *list*

abbreviation (*input*) *execlasses* ≡ *fst*
abbreviation (*input*) *exetcsigs* ≡ *snd*

abbreviation *alist-conds* :: ('*k*::*linorder* × '*v*) *list* ⇒ *bool* **where**
  alist-conds al ≡ *distinct* (*map fst al*)

definition *exe-ars-conds* :: (*name* × (*class* × *sort list*) *list*) *list* ⇒ *bool* **where**
  exe-ars-conds arss ⟷ *alist-conds arss* ∧ (∀ *ars* ∈ *snd* ‘ *set arss* . *alist-conds ars*)

fun *exe-ars-conds'* :: (('*k1*::*linorder*) × (('*k2*::*linorder*) × '*s list*) *list*) *list* ⇒ *bool* **where**
  exe-ars-conds' arss ⟷ *alist-conds arss* ∧ (∀ *ars* ∈ *snd* ‘ *set arss* . *alist-conds ars*)

lemma [*code*]: *exe-ars-conds arss* ⟷ *exe-ars-conds' arss*
  ⟨*proof*⟩

definition *exe-class-conds* :: (*class* × *class*) *list* ⇒ *bool* **where**
  exe-class-conds cs ≡ *distinct cs*

definition *exe-osig-conds* :: *exeosig* ⇒ *bool* **where**
  exe-osig-conds a ≡ *exe-class-conds* (*execlasses a*) ∧ *exe-ars-conds* (*exetcsigs a*)

fun *translate-ars* :: (*name* × (*class* × *sort list*) *list*) *list* ⇒ *name* ⇀ (*class* ⇀ *sort list*) **where**
  translate-ars ars = *map-of* (*map* (*apsnd map-of*) *ars*)

abbreviation *illformed-osig* ≡ ({}, *Map.empty*(*STR* ''*A*'' ↦ *Map.empty*(*STR* ''*A*'' ↦ [{*STR* ''*A*''}])))

lemma *illformed-osig-not-wf-osig*: ¬ *wf-osig illformed-osig*

⟨*proof*⟩

fun *translate-osig* :: *exeosig* ⇒ *osig* **where**
translate-osig (*cs*, *arss*) = (*if exe-osig-conds* (*cs*, *arss*)
then (*set cs*, *translate-ars arss*)
else illformed-osig)

definition *exe-consistent-length-tcsigs arss* ≡ (∀ *ars* ∈ *snd* ‘ *set arss* .
∀ *ss*$_1$ ∈ *snd* ‘ *set ars*. ∀ *ss*$_2$ ∈ *snd* ‘ *set ars*. *length ss*$_1$ = *length ss*$_2$)

lemma *in-alist-imp-in-map-of*: *distinct* (*map fst arss*)
⟹ (*name*, *ars*) ∈ *set arss* ⟹ *translate-ars arss name* = *Some* (*map-of ars*)
⟨*proof*⟩

lemma *exe-ars-conds arss* ⟹ ∃ *name* . *map-of* (*map* (*apsnd map-of*) *arss*) *name*
= *Some ars*
⟹ ∃ *name arsl* . (*name*, *arsl*) ∈ *set arss* ∧ *map-of arsl* = *ars*
⟨*proof*⟩

lemma *exe-ars-conds arss*
⟹ (*name*, *arsl*) ∈ *set arss* ∧ *map-of arsl* = *ars*
⟹ *map-of* (*map* (*apsnd map-of*) *arss*) *name* = *Some ars*
⟨*proof*⟩

lemma *consistent-length-tcsigs-imp-exe-consistent-length-tcsigs*:
exe-ars-conds arss ⟹ *consistent-length-tcsigs* (*translate-ars arss*)
⟹ *exe-consistent-length-tcsigs arss*
⟨*proof*⟩

lemma *exe-consistent-length-tcsigs-imp-consistent-length-tcsigs*:
assumes *exe-ars-conds arss exe-consistent-length-tcsigs arss*
shows *consistent-length-tcsigs* (*translate-ars arss*)
⟨*proof*⟩

lemma *consistent-length-tcsigs-iff-exe-consistent-length-tcsigs*:
exe-ars-conds arss ⟹
consistent-length-tcsigs (*translate-ars arss*) ⟷ *exe-consistent-length-tcsigs arss*
⟨*proof*⟩

definition *exe-complete-tcsigs cs arss*
≡ (∀ *ars* ∈ *snd* ‘ *set arss* .
∀ (*c*$_1$, *c*$_2$) ∈ *set cs* . *c*$_1$∈*fst* ‘ *set ars* ⟶ *c*$_2$∈*fst* ‘ *set ars*)

lemma *exe-complete-tcsigs-imp-complete-tcsigs*:
assumes *exe-ars-conds arss exe-complete-tcsigs cs arss*
shows *complete-tcsigs* (*set cs*) (*translate-ars arss*)
⟨*proof*⟩

lemma *complete-tcsigs-imp-exe-complete-tcsigs*: *exe-ars-conds arss* $\Longrightarrow$
complete-tcsigs (*set cs*) (*translate-ars arss*) $\Longrightarrow$ *exe-complete-tcsigs cs arss*
⟨*proof*⟩

lemma *exe-complete-tcsigs-iff-complete-tcsigs*:
exe-ars-conds arss $\Longrightarrow$
complete-tcsigs (*set cs*) (*translate-ars arss*) $\longleftrightarrow$ *exe-complete-tcsigs cs arss*
⟨*proof*⟩

definition *exe-coregular-tcsigs* (*cs* :: (*class* × *class*) *list*) *arss*
≡ (∀ *ars* ∈ *snd* ' *set arss* .
∀ c_1 ∈ *fst* ' *set ars*. ∀ c_2 ∈ *fst* ' *set ars*.
(*class-leq* (*set cs*) c_1 c_2 ⟶
list-all2 (*sort-leq* (*set cs*)) (*the* (*lookup* (λ*x*. *x*=c_1) *ars*)) (*the* (*lookup* (λ*x*. *x*=c_2) *ars*))))

lemma *exe-coregular-tcsigs-imp-coregular-tcsigs*:
assumes *exe-ars-conds arss exe-coregular-tcsigs cs arss*
shows *coregular-tcsigs* (*set cs*) (*translate-ars arss*)
⟨*proof*⟩

lemma *coregular-tcsigs-imp-exe-coregular-tcsigs*:
assumes *exe-ars-conds arss coregular-tcsigs* (*set cs*) (*translate-ars arss*)
shows *exe-coregular-tcsigs cs arss*
⟨*proof*⟩

lemma *coregular-tcsigs-iff-exe-coregular-tcsigs*:
exe-ars-conds arss $\Longrightarrow$ *coregular-tcsigs* (*set cs*) (*translate-ars arss*) $\longleftrightarrow$ *exe-coregular-tcsigs cs arss*
⟨*proof*⟩

lemma *wf-subclass sub* $\Longrightarrow$ *Field sub* = *Domain sub*
⟨*proof*⟩

definition [*simp*]: *exefield rel* = *List.union* (*map fst rel*) (*map snd rel*)
lemma *Field-set-code*: *Field* (*set rel*) = *set* (*exefield rel*)
⟨*proof*⟩

lemma *class-ex-rec*: *finite r* $\Longrightarrow$ *class-ex* (*insert* (*a*,*b*) *r*) *c* = (*a*=*c* ∨ *b*=*c* ∨ *class-ex r c*)
⟨*proof*⟩

definition [*simp*]: *execlass-ex rel c* = *List.member* (*exefield rel*) *c*
lemma *execlass-ex-code*: *class-ex* (*set rel*) *c* = *execlass-ex rel c*
⟨*proof*⟩

definition [*simp*]: *exesort-ex rel S* = (∀ *x*∈*S* . (*List.member* (*exefield rel*) *x*))

lemma *sort-ex-code*: *sort-ex* (*set rel*) *S* = *exesort-ex rel S*
⟨*proof*⟩

definition [*simp*]: *execlass-les cs c1 c2* = (*List.member cs* (*c1*,*c2*) ∧ ¬ *List.member cs* (*c2*,*c1*))
lemma *execlass-les-code*: *class-les* (*set cs*) *c1 c2* = *execlass-les cs c1 c2*
⟨*proof*⟩

definition [*simp*]: *exenormalize-sort cs* (*s*::*sort*)
= {*c* ∈ *s* . ¬ (∃ *c'* ∈ *s* . *execlass-les cs c' c*)}
definition [*simp*]: *exenormalized-sort cs s* ≡ (*exenormalize-sort cs s*) = *s*

lemma *normalize-sort-code*[*code*]: *normalize-sort* (*set cs*) *s* = *exenormalize-sort cs s*
⟨*proof*⟩

lemma *normalized-sort-code*[*code*]: *normalized-sort* (*set cs*) *s* = *exenormalized-sort cs s*
⟨*proof*⟩

definition [*simp*]: *exewf-sort sub S* ≡ *exenormalized-sort sub S* ∧ *exesort-ex sub S*
lemma *wf-sort-code*:
assumes *exe-class-conds sub*
shows *wf-sort* (*set sub*) *S* = *exewf-sort sub S*
⟨*proof*⟩

lemma [*code*]: *exewf-sort sub S* ≡ (*S* = {} ∨ *exenormalized-sort sub S* ∧ *exesort-ex sub S*)
⟨*proof*⟩

definition *exe-all-normalized-and-ex-tcsigs cs arss*
≡ (∀ *ars* ∈ *snd* ‘ *set arss* . ∀ *ss* ∈ *snd* ‘ *set ars* . ∀ *s* ∈ *set ss*. *exewf-sort cs s*)

lemma *all-normalized-and-ex-tcsigs-imp-exe-all-normalized-and-ex-tcsigs*:
assumes *exe-ars-conds arss all-normalized-and-ex-tcsigs* (*set cs*) (*translate-ars arss*)
shows *exe-all-normalized-and-ex-tcsigs cs arss*
⟨*proof*⟩

lemma *exe-all-normalized-and-ex-tcsigs-imp-all-normalized-and-ex-tcsigs*:
assumes *exe-ars-conds arss exe-all-normalized-and-ex-tcsigs cs arss*
shows *all-normalized-and-ex-tcsigs* (*set cs*) (*translate-ars arss*)
⟨*proof*⟩

lemma *all-normalized-and-ex-tcsigs-iff-exe-all-normalized-and-ex-tcsigs*:
exe-ars-conds arss ⟹ *all-normalized-and-ex-tcsigs* (*set cs*) (*translate-ars arss*)
⟷ *exe-all-normalized-and-ex-tcsigs cs arss*
⟨*proof*⟩

definition [*simp*]: *exe-wf-tcsigs* (*cs* :: (*class* × *class*) *list*) *arss* ≡
  exe-coregular-tcsigs cs arss
 ∧ *exe-complete-tcsigs cs arss*
 ∧ *exe-consistent-length-tcsigs arss*
 ∧ *exe-all-normalized-and-ex-tcsigs cs arss*

lemma *wf-tcsigs-iff-exe-wf-tcsigs*:
 exe-ars-conds arss ⟹ *wf-tcsigs* (*set cs*) (*translate-ars arss*) ⟷ *exe-wf-tcsigs cs arss*
 ⟨*proof*⟩

fun *exe-antisym* :: ('*a* × '*a*) *list* ⇒ *bool* **where**
 exe-antisym [] ⟷ *True*
| *exe-antisym* ((*x*,*y*)#*r*) ⟷ ((*y*,*x*)∈*set r* ⟶ *x*=*y*) ∧ *exe-antisym r*

lemma *exe-antisym-imp-antisym*: *exe-antisym l* ⟹ *antisym* (*set l*)
 ⟨*proof*⟩

lemma *antisym-imp-exe-antisym*: *antisym* (*set l*) ⟹ *exe-antisym l*
⟨*proof*⟩

lemma *antisym-iff-exe-antisym*: *antisym* (*set l*) = *exe-antisym l*
 ⟨*proof*⟩

definition *exe-wf-subclass cs* ⟷
 set cs ⊆ *Field* (*set cs*) × *Field* (*set cs*) ∧ *trans* (*set cs*) ∧ *exe-antisym cs* ∧ *Refl* (*set cs*)

lemma *wf-classes-iff-exe-wf-classes*: *wf-subclass* (*set cs*) ⟷ *exe-wf-subclass cs*
 ⟨*proof*⟩

definition [*simp*]: *exe-wf-osig oss* ≡ *exe-wf-subclass* (*execlasses oss*)
 ∧ *exe-wf-tcsigs* (*execlasses oss*) (*exetcsigs oss*) ∧ *exe-osig-conds oss*

lemma *exe-wf-osig-imp-wf-osig*: *exe-wf-osig oss* ⟹ *wf-osig* (*translate-osig oss*)
 ⟨*proof*⟩

lemma *classes-translate*: *exe-osig-conds oss* ⟹ *subclass* (*translate-osig oss*) = *set* (*execlasses oss*)
 ⟨*proof*⟩

lemma *tcsigs-translate*: *exe-osig-conds oss*
 ⟹ *tcsigs* (*translate-osig oss*) = *translate-ars* (*exetcsigs oss*)
 ⟨*proof*⟩

lemma *wf-osig-translate-imp-exe-osig-conds*:
 wf-osig (*translate-osig oss*) ⟹ *exe-osig-conds oss*
 ⟨*proof*⟩

lemma *wf-osig-imp-exe-wf-osig*:
assumes *wf-osig* (*translate-osig oss*) **shows** *exe-wf-osig oss*
⟨*proof*⟩

lemma *wf-osig-iff-exe-wf-osig*: *wf-osig* (*translate-osig oss*) ⟷ *exe-wf-osig oss*
⟨*proof*⟩

end

14 Executable Instance Relations

theory *Instances*
imports *Term*
begin

fun *raw-match* :: *typ* ⇒ *typ* ⇒ ((*variable* × *sort*) ⇀ *typ*) ⇒ ((*variable* × *sort*) ⇀ *typ*) *option*
and *raw-matches* :: *typ list* ⇒ *typ list* ⇒ ((*variable* × *sort*) ⇀ *typ*) ⇒ ((*variable* × *sort*) ⇀ *typ*) *option*
where
raw-match (*Tv v S*) *T subs* =
(*case subs* (*v*,*S*) *of*
None ⇒ *Some* (*subs*((*v*,*S*) := *Some T*))
| *Some U* ⇒ (*if U* = *T then Some subs else None*))
| *raw-match* (*Ty a Ts*) (*Ty b Us*) *subs* =
(*if a*=*b then raw-matches Ts Us subs else None*)
| *raw-match* - - - = *None*
| *raw-matches* (*T*#*Ts*) (*U*#*Us*) *subs* = *Option.bind* (*raw-match T U subs*) (*raw-matches Ts Us*)
| *raw-matches* [] [] *subs* = *Some subs*
| *raw-matches* - - *subs* = *None*

function (*sequential*) *raw-match*′
:: *typ* ⇒ *typ* ⇒ ((*variable* × *sort*) ⇀ *typ*) ⇒ ((*variable* × *sort*) ⇀ *typ*) *option*
where
raw-match′ (*Tv v S*) *T subs* =
(*case subs* (*v*,*S*) *of*
None ⇒ *Some* (*subs*((*v*,*S*) := *Some T*))
| *Some U* ⇒ (*if U* = *T then Some subs else None*))
| *raw-match*′ (*Ty a Ts*) (*Ty b Us*) *subs* =
(*if a*=*b* ∧ *length Ts* = *length Us*
then fold (λ(*T*, *U*) *subs* . *Option.bind subs* (*raw-match*′ *T U*)) (*zip Ts Us*) (*Some subs*)
else None)
| *raw-match*′ *T U subs* = (*if T* = *U then Some subs else None*)

⟨*proof*⟩
termination ⟨*proof*⟩

lemma *length-neq-imp-not-raw-matches*: *length Ts* ≠ *length Us* ⟹ *raw-matches Ts Us subs* = *None*
⟨*proof*⟩

lemma *raw-match T U subs* = *raw-match′ T U subs*
⟨*proof*⟩

lemma *raw-match′-map-le*: *raw-match′ T U subs* = *Some subs′* ⟹ *map-le subs subs′*
⟨*proof*⟩

lemma *fold-matches-first-step-not-None*:
assumes
fold (λ(*T*, *U*) *subs* . *Option.bind subs* (*raw-match′ T U*)) (*zip* (*x*#*xs*) (*y*#*ys*)) (*Some subs*) = *Some subs′*
obtains *point* **where**
raw-match′ x y subs = *Some point*
fold (λ(*T*, *U*) *subs* . *Option.bind subs* (*raw-match′ T U*)) (*zip* (*xs*) (*ys*)) (*Some point*) = *Some subs′*
⟨*proof*⟩
lemma *fold-matches-last-step-not-None*:
assumes
length xs = *length ys*
fold (λ(*T*, *U*) *subs* . *Option.bind subs* (*raw-match′ T U*)) (*zip* (*xs*@[*x*]) (*ys*@[*y*])) (*Some subs*) = *Some subs′*
obtains *point* **where**
fold (λ(*T*, *U*) *subs* . *Option.bind subs* (*raw-match′ T U*)) (*zip* (*xs*) (*ys*)) (*Some subs*) = *Some point*
raw-match′ x y point = *Some subs′*
⟨*proof*⟩

corollary *raw-match′-Type-conds*:
assumes *raw-match′* (*Ty a Ts*) (*Ty b Us*) *subs* = *Some subs′*
shows *a*=*b length Ts* = *length Us*
⟨*proof*⟩

corollary *fold-matches-first-step-not-None′*:
assumes *length xs* = *length ys*
fold (λ(*T*, *U*) *subs* . *Option.bind subs* (*raw-match′ T U*)) (*zip* (*x*#*xs*) (*y*#*ys*)) (*Some subs*) = *Some subs′*
shows *raw-match′ x y subs* ~= *None*
⟨*proof*⟩

corollary *raw-match′-hd-raw-match′*:

assumes *raw-match′ (Ty a (T#Ts)) (Ty b (U#Us)) subs = Some subs′*
shows *raw-match′ T U subs ~= None*
⟨*proof*⟩

corollary *raw-match′-eq-Some-at-point-not-None′*:
assumes *length Ts = length Us*
assumes *raw-match′ (Ty a (Ts@Ts′)) (Ty b (Us@Us′)) subs = Some subs′*
shows *raw-match′ (Ty a (Ts)) (Ty b (Us)) subs ~= None*
⟨*proof*⟩

lemma *raw-match′-tvsT-subset-dom-res*: *raw-match′ T U subs = Some subs′ ⟹ tvsT T ⊆ dom subs′*
⟨*proof*⟩

lemma *raw-match′-dom-res-subset-tvsT*:
raw-match′ T U subs = Some subs′ ⟹ dom subs′ ⊆ tvsT T ∪ dom subs
⟨*proof*⟩

corollary *raw-match′-dom-res-eq-tvsT*:
raw-match′ T U subs = Some subs′ ⟹ dom subs′ = tvsT T ∪ dom subs
⟨*proof*⟩

corollary *raw-match′-dom-res-eq-tvsT-empty*:
raw-match′ T U (λx. None) = Some subs′ ⟹ dom subs′ = tvsT T
⟨*proof*⟩

lemma *raw-match′-map-defined*: *raw-match′ T U subs = Some subs′ ⟹ p∈tvsT T ⟹ subs′ p ~= None*
⟨*proof*⟩

lemma *raw-match′-extend-map-preserve*:
raw-match′ T U subs = Some subs′ ⟹ map-le subs′ subs″ ⟹ p∈tvsT T ⟹ subs″ p = subs′ p
⟨*proof*⟩

abbreviation *convert-subs subs* ≡ *(λv S . the-default (Tv v S) (subs (v, S)))*

lemma *map-eq-on-tvsT-imp-map-eq-on-typ*:
(⋀p . p∈tvsT T ⟹ subs p = subs′ p)
⟹ tsubstT T (convert-subs subs)
= tsubstT T (convert-subs subs′)
⟨*proof*⟩

lemma *raw-match′-extend-map-preserve′*:
assumes *raw-match′ T U subs = Some subs′ map-le subs′ subs″*
shows *tsubstT T (convert-subs subs′)*
= tsubstT T (convert-subs subs″)

⟨*proof*⟩

lemma *raw-match′-produces-matcher*:
raw-match′ T U subs = *Some subs′*
⟹ *tsubstT T* (*convert-subs subs′*) = *U*
⟨*proof*⟩

lemma *tsubstT-matcher-imp-raw-match′-unchanged*:
tsubstT T ϱ = *U* ⟹ *raw-match′ T U* (λ(*idx*, *S*). *Some* (*ϱ idx S*)) = *Some* (λ(*idx*, *S*). *Some* (*ϱ idx S*))
⟨*proof*⟩

lemma *raw-match′-imp-raw-match′-on-map-le*:
assumes *raw-match′ T U subs* = *Some subs′*
assumes *map-le lesubs subs*
shows ∃ *lesubs′*. *raw-match′ T U lesubs* = *Some lesubs′* ∧ *map-le lesubs′ subs′*
⟨*proof*⟩

lemma *map-le-same-dom-imp-same-map*: *dom f* = *dom g* ⟹ *map-le f g* ⟹ *f* = *g*
⟨*proof*⟩

corollary *map-le-produces-same-raw-match′*:
assumes *raw-match′ T U subs* = *Some subs′*
assumes *dom subs* ⊆ *tvsT T*
assumes *map-le lesubs subs*
shows *raw-match′ T U lesubs* = *Some subs′*
⟨*proof*⟩

corollary *raw-match′ T U subs* = *Some subs′* ⟹ *dom subs* ⊆ *tvsT T* ⟹
raw-match′ T U (λ*p* . *None*) = *Some subs′*
⟨*proof*⟩

lemma *raw-match′-restriction*:
assumes *raw-match′ T U subs* = *Some subs′*
assumes *tvsT T* ⊆ *restriction*
shows *raw-match′ T U* (*subs*|‘*restriction*) = *Some* (*subs′*|‘*restriction*)
⟨*proof*⟩

corollary *raw-match′-restriction-on-tvsT*:
assumes *raw-match′ T U subs* = *Some subs′*
shows *raw-match′ T U* (*subs*|‘*tvsT T*) = *Some* (*subs′*|‘*tvsT T*)
⟨*proof*⟩

lemma *tinstT-imp-ex-raw-match′*:
assumes *tinstT T1 T2*
shows ∃ *subs*. *raw-match′ T2 T1* (λ*p* . *None*) = *Some subs*
⟨*proof*⟩

lemma *ex-raw-match'-imp-tinstT*:
assumes ∃ *subs*. *raw-match' T2 T1* (λ*p* . *None*) = *Some subs*
shows *tinstT T1 T2*
⟨*proof*⟩

corollary *tinstT-iff-ex-raw-match'*:
tinstT T1 T2 ⟷ (∃ *subs*. *raw-match' T2 T1* (λ*p* . *None*) = *Some subs*)
⟨*proof*⟩

function (*sequential*) *raw-match-term*
:: *term* ⇒ *term* ⇒ ((*variable* × *sort*) ⇀ *typ*) ⇒ ((*variable* × *sort*) ⇀ *typ*) *option*
where
raw-match-term (*Ct a T*) (*Ct b U*) *subs* = (*if a* = *b then raw-match' T U subs else None*)
| *raw-match-term* (*Fv a T*) (*Fv b U*) *subs* = (*if a* = *b then raw-match' T U subs else None*)
| *raw-match-term* (*Bv i*) (*Bv j*) *subs* = (*if i* = *j then Some subs else None*)
| *raw-match-term* (*Abs T t*) (*Abs U u*) *subs* =
Option.bind (*raw-match' T U subs*) (*raw-match-term t u*)
| *raw-match-term* (*f* $ *u*) (*f'* $ *u'*) *subs* = *Option.bind* (*raw-match-term f f' subs*) (*raw-match-term u u'*)
| *raw-match-term* - - - = *None*
⟨*proof*⟩
termination ⟨*proof*⟩

lemma *raw-match-term-map-le*: *raw-match-term t u subs* = *Some subs'* ⟹ *map-le subs subs'*
⟨*proof*⟩

lemma *raw-match-term-tvs-subset-dom-res*:
raw-match-term t u subs = *Some subs'* ⟹ *tvs t* ⊆ *dom subs'*
⟨*proof*⟩

lemma *raw-match-term-dom-res-subset-tvs*:
raw-match-term t u subs = *Some subs'* ⟹ *dom subs'* ⊆ *tvs t* ∪ *dom subs*
⟨*proof*⟩

corollary *raw-match-term-dom-res-eq-tvs*:
raw-match-term t u subs = *Some subs'* ⟹ *dom subs'* = *tvs t* ∪ *dom subs*
⟨*proof*⟩

lemma *raw-match-term-extend-map-preserve*:
raw-match-term t u subs = *Some subs'* ⟹ *map-le subs' subs''* ⟹ *p*∈*tvs t* ⟹ *subs'' p* = *subs' p*
⟨*proof*⟩

lemma *map-eq-on-tvs-imp-map-eq-on-term*:

$(\bigwedge p\ .\ p{\in}tvs\ t \Longrightarrow subs\ p = subs'\ p)$
$\Longrightarrow tsubst\ t\ (convert\text{-}subs\ subs)$
$= tsubst\ t\ (convert\text{-}subs\ subs')$
⟨*proof*⟩

lemma *raw-match-extend-map-preserve'*:
assumes *raw-match-term t u subs = Some subs' map-le subs' subs''*
shows *tsubst t (convert-subs subs')*
$= tsubst\ t\ (convert\text{-}subs\ subs'')$
⟨*proof*⟩

lemma *raw-match-term-produces-matcher*:
raw-match-term t u subs = Some subs'
$\Longrightarrow tsubst\ t\ (convert\text{-}subs\ subs') = u$
⟨*proof*⟩

lemma *ex-raw-match-term-imp-tinst*:
assumes $\exists$ *subs. raw-match-term t2 t1* $(\lambda p\ .\ None) = Some\ subs$
shows *tinst t1 t2*
⟨*proof*⟩

lemma *tsubst-matcher-imp-raw-match-term-unchanged*:
$tsubst\ t\ \varrho = u \Longrightarrow raw\text{-}match\text{-}term\ t\ u\ (\lambda(idx,\ S).\ Some\ (\varrho\ idx\ S)) = Some$
$(\lambda(idx,\ S).\ Some\ (\varrho\ idx\ S))$
⟨*proof*⟩

lemma *raw-match-term-restriction*:
assumes *raw-match-term t u subs = Some subs'*
assumes *tvs t* $\subseteq$ *restriction*
shows *raw-match-term t u (subs|`restriction) = Some (subs'|`restriction)*
⟨*proof*⟩

corollary *raw-match-term-restriction-on-tvs*:
assumes *raw-match-term t u subs = Some subs'*
shows *raw-match-term t u (subs|`tvs t) = Some (subs'|`tvs t)*
⟨*proof*⟩

lemma *raw-match-term-imp-raw-match-term-on-map-le*:
assumes *raw-match-term t u subs = Some subs'*
assumes *map-le lesubs subs*
shows $\exists$ *lesubs'. raw-match-term t u lesubs = Some lesubs'* $\wedge$ *map-le lesubs' subs'*
⟨*proof*⟩

corollary *map-le-produces-same-raw-match-term*:
assumes *raw-match-term t u subs = Some subs'*
assumes *dom subs* $\subseteq$ *tvs t*
assumes *map-le lesubs subs*
shows *raw-match-term t u lesubs = Some subs'*
⟨*proof*⟩

lemma *tinst-imp-ex-raw-match-term*:
assumes *tinst t1 t2*
shows $\exists$ *subs. raw-match-term t2 t1* (λ*p . None*) = *Some subs*
⟨*proof*⟩

corollary *tinst-iff-ex-raw-match-term*:
tinst t1 t2 $\longleftrightarrow$ ($\exists$ *subs. raw-match-term t2 t1* (λ*p . None*) = *Some subs*)
⟨*proof*⟩

function (*sequential*) *assoc-match*
:: *typ* ⇒ *typ* ⇒ ((*variable* × *sort*) × *typ*) *list* ⇒ ((*variable* × *sort*) × *typ*) *list option* **where**
assoc-match (*Tv v S*) *T subs* =
(*case lookup* (λ*x. x*=(*v*,*S*)) *subs of*
None ⇒ *Some* (((*v*,*S*), *T*) # *subs*)
| *Some U* ⇒ (*if U* = *T then Some subs else None*))
| *assoc-match* (*Ty a Ts*) (*Ty b Us*) *subs* =
(*if a*=*b* ∧ *length Ts* = *length Us*
then fold (λ(*T*, *U*) *subs . Option.bind subs* (*assoc-match T U*)) (*zip Ts Us*) (*Some subs*)
else None)
| *assoc-match T U subs* = (*if T* = *U then Some subs else None*)
⟨*proof*⟩
termination ⟨*proof*⟩

corollary *assoc-match-Type-conds*:
assumes *assoc-match* (*Ty a Ts*) (*Ty b Us*) *subs* = *Some subs′*
shows *a*=*b length Ts* = *length Us*
⟨*proof*⟩

lemma *fold-assoc-matches-first-step-not-None*:
assumes
fold (λ(*T*, *U*) *subs . Option.bind subs* (*assoc-match T U*)) (*zip* (*x*#*xs*) (*y*#*ys*)) (*Some subs*) = *Some subs′*
obtains *point* **where**
assoc-match x y subs = *Some point*
fold (λ(*T*, *U*) *subs . Option.bind subs* (*assoc-match T U*)) (*zip* (*xs*) (*ys*)) (*Some point*) = *Some subs′*
⟨*proof*⟩

lemma *assoc-match-subset*: *assoc-match T U subs* = *Some subs′* ⟹ *set subs* ⊆ *set subs′*
⟨*proof*⟩

lemma *assoc-match-distinct*: *assoc-match T U subs* = *Some subs′* ⟹ *distinct* (*map fst subs*)

$\Longrightarrow$ *distinct* (*map fst subs'*)
⟨*proof*⟩

lemma *lookup-eq-map-of-ap*:
shows *lookup* (λ*x*. *x*=*k*) *subs* = *map-of subs k*
⟨*proof*⟩

lemma *raw-match'-assoc-match*:
shows *raw-match' T U* (*map-of subs*) = *map-option map-of* (*assoc-match T U subs*)
⟨*proof*⟩

lemma *dom-eq-and-eq-on-dom-imp-eq*: *dom m* = *dom m'* $\Longrightarrow$ ∀ *x*∈*dom m* . *m x* = *m' x* $\Longrightarrow$ *m* = *m'*
⟨*proof*⟩

lemma *list-of-map*:
assumes *finite* (*dom subs*)
shows ∃ *l*. *map-of l* = *subs*
⟨*proof*⟩

corollary *tinstT-iff-assoc-match*[*code*]: *tinstT T1 T2* $\longleftrightarrow$ *assoc-match T2 T1* [] ~= *None*
⟨*proof*⟩

function (*sequential*) *assoc-match-term*
:: *term* ⇒ *term* ⇒ ((*variable* × *sort*) × *typ*) *list* ⇒ ((*variable* × *sort*) × *typ*) *list option*
where
assoc-match-term (*Ct a T*) (*Ct b U*) *subs* = (*if a* = *b then assoc-match T U subs else None*)
| *assoc-match-term* (*Fv a T*) (*Fv b U*) *subs* = (*if a* = *b then assoc-match T U subs else None*)
| *assoc-match-term* (*Bv i*) (*Bv j*) *subs* = (*if i* = *j then Some subs else None*)
| *assoc-match-term* (*Abs T t*) (*Abs U u*) *subs* =
Option.bind (*assoc-match T U subs*) (*assoc-match-term t u*)
| *assoc-match-term* (*f* $ *u*) (*f'* $ *u'*) *subs* = *Option.bind* (*assoc-match-term f f' subs*) (*assoc-match-term u u'*)
| *assoc-match-term* - - - = *None*
⟨*proof*⟩
termination ⟨*proof*⟩

lemma *raw-match-term-assoc-match-term*:
raw-match-term t u (*map-of subs*) = *map-option map-of* (*assoc-match-term t u subs*)
⟨*proof*⟩

corollary *tinst-iff-assoc-match-term*[*code*]: *tinst t1 t2* $\longleftrightarrow$ *assoc-match-term t2 t1* [] $\neq$ *None*
⟨*proof*⟩

hide-fact *fold-matches-first-step-not-None fold-matches-last-step-not-None*

end

15 Executable Signature and Theory

theory *TheoryExe*
imports *SortsExe Theory Instances*
begin

datatype *exesignature* = *ExeSignature*
(*execonst-type-of*: (*name* × *typ*) *list*)
(*exetyp-arity-of*: (*name* × *nat*) *list*)
(*exesorts*: *exeosig*)

lemma *exe-const-type-of-ok*:
alist-conds cto ⟹
(∀ *ty* ∈ *Map.ran* (*map-of cto*) . *typ-ok-sig* (*map-of cto*, *ta*, *sa*) *ty*)
⟷ (∀ *ty* ∈ *snd* ‘ *set cto* . *typ-ok-sig* (*map-of cto*, *ta*, *sa*) *ty*)
⟨*proof*⟩

fun *exe-wf-sig* **where**
exe-wf-sig (*ExeSignature cto tao sa*) = (*exe-wf-osig sa* ∧
fst ‘ *set* (*exetcsigs sa*) = *fst* ‘ *set tao*
∧ (∀ *type* ∈ *fst* ‘ *set* (*exetcsigs sa*).
(∀ *ars* ∈ *snd* ‘ *set* (*the* (*lookup* (λ*k*. *k*=*type*) (*exetcsigs sa*))) .
the (*lookup* (λ*k*. *k*=*type*) *tao*) = *length ars*))
∧ (∀ *ty* ∈ *snd* ‘ *set cto* . *typ-ok-sig* (*map-of cto*, *map-of tao*, *translate-osig sa*) *ty*))

lemma *exe-wf-sig-imp-wf-sig*:
assumes *alist-conds cto alist-conds tao exe-osig-conds sa* (*exe-wf-osig sa*
∧ *fst* ‘ *set* (*exetcsigs sa*) = *fst* ‘ *set tao*
∧ (∀ *type* ∈ *fst* ‘ *set* (*exetcsigs sa*).
(∀ *ars* ∈ *snd* ‘ *set* (*the* (*lookup* (λ*k*. *k*=*type*) (*exetcsigs sa*))) .
the (*lookup* (λ*k*. *k*=*type*) *tao*) = *length ars*)))
∧ (∀ *ty* ∈ *snd* ‘ *set cto* . *typ-ok-sig* (*map-of cto*, *map-of tao*, *translate-osig sa*) *ty*)
shows *wf-sig* (*map-of cto*, *map-of tao*, *translate-osig sa*)
⟨*proof*⟩

lemma *wf-sig-imp-exe-wf-sig*:
assumes *alist-conds cto alist-conds tao exe-osig-conds sa*
wf-sig (*map-of cto*, *map-of tao*, *translate-osig sa*)
shows (*exe-wf-osig sa*

$\wedge$ *fst ‘ set* (*exetcsigs sa*) = *fst ‘ set tao*
$\wedge$ ($\forall$ *type* $\in$ *fst ‘ set* (*exetcsigs sa*).
(∀ *ars* ∈ *snd ‘ set* (*the* (*lookup* (λ*k*. *k*=*type*) (*exetcsigs sa*))) .
the (*lookup* (λ*k*. *k*=*type*) *tao*) = *length ars*)))
$\wedge$ ($\forall$ *ty* $\in$ *snd ‘ set cto* . *typ-ok-sig* (*map-of cto*, *map-of tao*, *translate-osig sa*) *ty*)
⟨*proof*⟩

lemma *wf-sig-iff-exe-wf-sig-pre*: *alist-conds cto* $\Longrightarrow$ *alist-conds tao* $\Longrightarrow$ *exe-osig-conds sa*
$\Longrightarrow$ *wf-sig* (*map-of cto*, *map-of tao*, *translate-osig sa*) = (*exe-wf-osig sa*
$\wedge$ *fst ‘ set* (*exetcsigs sa*) = *fst ‘ set tao*
$\wedge$ ($\forall$ *type* $\in$ *fst ‘ set* (*exetcsigs sa*).
(∀ *ars* ∈ *snd ‘ set* (*the* (*lookup* (λ*k*. *k*=*type*) (*exetcsigs sa*))) .
the (*lookup* (λ*k*. *k*=*type*) *tao*) = *length ars*))
$\wedge$ ($\forall$ *ty* $\in$ *snd ‘ set cto* . *typ-ok-sig* (*map-of cto*, *map-of tao*, *translate-osig sa*) *ty*))
⟨*proof*⟩

lemma *wf-sig-iff-exe-wf-sig*: *alist-conds cto* $\Longrightarrow$ *alist-conds tao* $\Longrightarrow$ *exe-osig-conds sa*
$\Longrightarrow$ *wf-sig* (*map-of cto*, *map-of tao*, *translate-osig sa*)
$\longleftrightarrow$ *exe-wf-sig* (*ExeSignature cto tao sa*)
⟨*proof*⟩

fun *translate-signature* :: *exesignature* $\Rightarrow$ *signature* **where**
translate-signature (*ExeSignature cto tao sa*)
= (*map-of cto*, *map-of tao*, *translate-osig sa*)

fun *exetyp-ok-sig* :: *exesignature* $\Rightarrow$ *typ* $\Rightarrow$ *bool* **where**
exetyp-ok-sig Σ (*Ty c Ts*) = (*case lookup* (λ*k*. *k*=*c*) (*exetyp-arity-of* Σ) *of*
None $\Rightarrow$ *False*
| *Some ar* $\Rightarrow$ *length Ts* = *ar* $\wedge$ *list-all* (*exetyp-ok-sig* Σ) *Ts*)
| *exetyp-ok-sig* Σ (*Tv* - *S*) = *exewf-sort* (*execlasses* (*exesorts* Σ)) *S*

thm *exewf-sort-def*
definition [*simp*]: *exesort-ok-sig* Σ *S* $\equiv$ *exesort-ex* (*execlasses* (*exesorts* Σ)) *S*
$\wedge$ *exenormalized-sort* (*execlasses* (*exesorts* Σ)) *S*

lemma *typ-arity-lookup-code*: *type-arity* (*translate-signature* Σ) *n* = *lookup* (λ*k*. *k* = *n*) (*exetyp-arity-of* Σ)
⟨*proof*⟩

lemma *typ-ok-sig-code*:
assumes *exe-osig-conds* (*exesorts* Σ)
shows *typ-ok-sig* (*translate-signature* Σ) *ty* = *exetyp-ok-sig* Σ *ty*
⟨*proof*⟩

fun *exe-wf-sig′* **where**
exe-wf-sig′ (*ExeSignature cto tao sa*) = (*exe-wf-osig sa* $\wedge$

fst ‘ set (exetcsigs sa) = fst ‘ set tao
∧ (∀ type ∈ fst ‘ set (exetcsigs sa).
(∀ ars ∈ snd ‘ set (the (lookup (λk. k=type) (exetcsigs sa))) .
the (lookup (λk. k=type) tao) = length ars))
∧ (∀ ty ∈ snd ‘ set cto . exetyp-ok-sig (ExeSignature cto tao sa) ty))

lemma *exe-wf-sig-code*[*code*]: *exe-wf-sig Σ = exe-wf-sig′ Σ*
⟨*proof*⟩

fun *exeterm-ok′* :: *exesignature ⇒ term ⇒ bool* **where**
exeterm-ok′ Σ (Fv - T) = exetyp-ok-sig Σ T
| *exeterm-ok′ Σ (Bv -) = True*
| *exeterm-ok′ Σ (Ct s T) = (case lookup (λk. k=s) (execonst-type-of Σ) of*
None ⇒ False
| *Some ty ⇒ exetyp-ok-sig Σ T ∧ tinstT T ty)*
| *exeterm-ok′ Σ (t $ u) ⟷ exeterm-ok′ Σ t ∧ exeterm-ok′ Σ u*
| *exeterm-ok′ Σ (Abs T t) ⟷ exetyp-ok-sig Σ T ∧ exeterm-ok′ Σ t*

lemma *const-type-of-lookup-code*: *const-type (translate-signature Σ) n = lookup (λk. k = n) (execonst-type-of Σ)*
⟨*proof*⟩

lemma *wt-term-code*:
assumes *exe-osig-conds (exesorts Σ)*
shows *term-ok′ (translate-signature Σ) t = exeterm-ok′ Σ t*
⟨*proof*⟩

datatype *exetheory = ExeTheory* (*exesig*: *exesignature*) (*exeaxioms-of*: *term list*)

lemma *exetheory-full-exhaust*: (⋀*const-type typ-arity sorts axioms.*
Θ = (ExeTheory (ExeSignature const-type typ-arity sorts) axioms) ⟹ P)
⟹ P
⟨*proof*⟩

definition *exe-sig-conds Σ ≡ alist-conds (execonst-type-of Σ) ∧ alist-conds (exetyp-arity-of Σ)*
∧ exe-osig-conds (exesorts Σ)

abbreviation *illformed-theory ≡* ((*Map.empty*, *Map.empty*, *illformed-osig*), {})

lemma *illformed-theory-not-wf-theory*: *¬ wf-theory illformed-theory*
⟨*proof*⟩

fun *translate-theory* :: *exetheory ⇒ theory* **where**
translate-theory (ExeTheory Σ ax) = (if exe-sig-conds Σ then
(translate-signature Σ, set ax) else illformed-theory)

fun *exe-wf-theory* **where** *exe-wf-theory (ExeTheory (ExeSignature cto tao sa) ax)*
⟷

exe-sig-conds (*ExeSignature cto tao sa*) ∧
(∀ *p* ∈ *set ax* . *term-ok* (*translate-theory* (*ExeTheory* (*ExeSignature cto tao sa*) *ax*)) *p* ∧ *typ-of p* = *Some propT*)
∧ *is-std-sig* (*translate-signature* (*ExeSignature cto tao sa*))
∧ *exe-wf-sig* (*ExeSignature cto tao sa*)
∧ *eq-axs* ⊆ *set ax*

lemma *wf-sig-iff-exe-wf-sig'*: *exe-sig-conds* Σ ⟹
wf-sig (*translate-signature* Σ) ⟷
exe-wf-sig Σ
⟨*proof*⟩

lemma *wf-sig-imp-exe-wf-sig'*: *exe-sig-conds* Σ ⟹
wf-sig (*translate-signature* Σ) ⟹
exe-wf-sig Σ
⟨*proof*⟩

lemma *exe-wf-sig-imp-wf-sig'*: *exe-sig-conds* Σ ⟹
exe-wf-sig Σ
⟹ *wf-sig* (*translate-signature* Σ)
⟨*proof*⟩

lemma *wf-theory-translate-imp-exe-wf-theory*:
assumes *wf-theory* (*translate-theory a*) **shows** *exe-wf-theory a*
⟨*proof*⟩

lemma *exe-wf-theory-translate-imp-wf-theory*:
assumes *exe-wf-theory a* **shows** *wf-theory* (*translate-theory a*)
⟨*proof*⟩

lemma *wf-theory-translate-iff-exe-wf-theory*:
wf-theory (*translate-theory a*) ⟷ *exe-wf-theory a*
⟨*proof*⟩

fun *exeis-std-sig* **where** *exeis-std-sig* (*ExeSignature cto tao sorts*) ⟷
lookup (λ*k*. *k* = *STR ''fun''*) *tao* = *Some 2* ∧ *lookup* (λ*k*. *k* = *STR ''prop''*) *tao* = *Some 0*
∧ *lookup* (λ*k*. *k* = *STR ''itself''*) *tao* = *Some 1*
∧ *lookup* (λ*k*. *k* = *STR ''Pure.eq''*) *cto*
= *Some* ((*Tv* (*Var* (*STR '''a''*, *0*)) *full-sort*) → ((*Tv* (*Var* (*STR '''a''*, *0*)) *full-sort*) → *propT*))
∧ *lookup* (λ*k*. *k* = *STR ''Pure.all''*) *cto* = *Some* ((*Tv* (*Var* (*STR '''a''*, *0*)) *full-sort* → *propT*) → *propT*)
∧ *lookup* (λ*k*. *k* = *STR ''Pure.imp''*) *cto* = *Some* (*propT* → (*propT* → *propT*))
∧ *lookup* (λ*k*. *k* = *STR ''Pure.type''*) *cto* = *Some* (*itselfT* (*Tv* (*Var* (*STR '''a''*, *0*)) *full-sort*))

lemma *is-std-sig-code*: *is-std-sig* (*translate-signature* Σ) = *exeis-std-sig* Σ
⟨*proof*⟩

fun *exe-wf-theory′* **where** *exe-wf-theory′* (*ExeTheory* (*ExeSignature cto tao sa*) *ax*)
⟷
  exe-sig-conds (*ExeSignature cto tao sa*) ∧
    (∀ *p* ∈ *set ax* . *exeterm-ok′* (*ExeSignature cto tao sa*) *p* ∧ *typ-of p* = *Some propT*)
  ∧ *exeis-std-sig* (*ExeSignature cto tao sa*)
  ∧ *exe-wf-sig* (*ExeSignature cto tao sa*)
  ∧ *eq-axs* ⊆ *set ax*

lemma *term-ok′-code*:
  assumes *exe-osig-conds* (*exesorts* (*ExeSignature cto tao sa*))
  shows (*term-ok′* (*translate-signature* (*ExeSignature cto tao sa*)) *p* ∧ *typ-of p* = *Some propT*)
    = (*exeterm-ok′* (*ExeSignature cto tao sa*) *p* ∧ *typ-of p* = *Some propT*)
  ⟨*proof*⟩

lemma *term-ok-translate-code-step*:
  assumes *exe-sig-conds* (*ExeSignature cto tao sa*)
  shows (*term-ok* (*translate-theory* (*ExeTheory* (*ExeSignature cto tao sa*) *ax*)) *p* ∧ *typ-of p* = *Some propT*)
    = (*term-ok′* (*translate-signature* (*ExeSignature cto tao sa*)) *p* ∧ *typ-of p* = *Some propT*)
  ⟨*proof*⟩

lemma *term-ok-theory-cond-code*:
  assumes *exe-sig-conds* (*ExeSignature cto tao sa*)
  shows(∀ *p* ∈ *set ax* . *term-ok* (*translate-theory* (*ExeTheory* (*ExeSignature cto tao sa*) *ax*)) *p* ∧ *typ-of p* = *Some propT*)
    = (∀ *p* ∈ *set ax* . *exeterm-ok′* (*ExeSignature cto tao sa*) *p* ∧ *typ-of p* = *Some propT*)
  ⟨*proof*⟩

lemma *exe-wf-theory-code*[*code*]: *exe-wf-theory* Θ = *exe-wf-theory′* Θ
  ⟨*proof*⟩

end

theory *CheckerExe*
  imports *TheoryExe ProofTerm*
begin

abbreviation *exetyp-ok* Θ ≡ *exetyp-ok-sig* (*exesig* Θ)

lemma *typ-ok-code*:
  assumes *exe-wf-theory′* Θ
  shows *typ-ok* (*translate-theory* Θ) *ty* = *exetyp-ok* Θ *ty*
  ⟨*proof*⟩

definition [*simp*]: *execlass-leq cs c1 c2* = *List.member cs* (*c1*,*c2*)
lemma *execlass-leq-code*: *class-leq* (*set cs*) *c1 c2* = *execlass-leq cs c1 c2*
⟨*proof*⟩

definition *exesort-leq sub s1 s2* = (∀ c_2 ∈ *s2* . ∃ c_1 ∈ *s1*. *execlass-leq sub* c_1 c_2)
lemma *exesort-les-code*: *sort-leq* (*set cs*) *c1 c2* = *exesort-leq cs c1 c2*
⟨*proof*⟩

fun *exehas-sort* :: *exeosig* ⇒ *typ* ⇒ *sort* ⇒ *bool* **where**
exehas-sort oss (*Tv* - *S*) *S'* = *exesort-leq* (*execlasses oss*) *S S'* |
exehas-sort oss (*Ty a Ts*) *S* =
(*case lookup* (λ*k*. *k*=*a*) (*exetcsigs oss*) *of*
None ⇒ *False* |
Some mgd ⇒ (∀ *C* ∈ *S*.
case lookup (λ*k*. *k*=*C*) *mgd of*
None ⇒ *False*
| *Some Ss* ⇒ *list-all2* (*exehas-sort oss*) *Ts Ss*))

lemma *exehas-sort-imp-has-sort*:
assumes *exe-osig-conds* (*sub*, *tcs*)
shows *exehas-sort* (*sub*, *tcs*) *T S* ⟹ *has-sort* (*translate-osig* (*sub*, *tcs*)) *T S*
⟨*proof*⟩

lemma *has-sort-imp-exehas-sort*:
assumes *exe-osig-conds* (*sub*, *tcs*)
shows *has-sort* (*translate-osig* (*sub*, *tcs*)) *T S* ⟹ *exehas-sort* (*sub*, *tcs*) *T S*
⟨*proof*⟩

lemma *has-sort-code*:
assumes *exe-osig-conds oss*
shows *has-sort* (*translate-osig oss*) *T S* = *exehas-sort oss T S*
⟨*proof*⟩

lemma *has-sort-code'*:
assumes *exe-wf-theory'* Θ
shows *has-sort* (*osig* (*sig* (*translate-theory* Θ))) *T S*
= *exehas-sort* (*exesorts* (*exesig* Θ)) *T S*
⟨*proof*⟩

abbreviation *exeinst-ok* Θ *insts* ≡
distinct (*map fst insts*)
∧ *list-all* (*exetyp-ok* Θ) (*map snd insts*)
∧ *list-all* (λ((*idn*, *S*), *T*) . *exehas-sort* (*exesorts* (*exesig* Θ)) *T S*) *insts*

lemma *inst-ok-code1*:
assumes *exe-wf-theory'* Θ
shows *list-all* (*exetyp-ok* Θ) (*map snd insts*) = *list-all* (*typ-ok* (*translate-theory* Θ)) (*map snd insts*)

⟨*proof*⟩

lemma *inst-ok-code2*:
assumes *exe-wf-theory'* Θ
shows *list-all* (λ((*idn*, *S*), *T*) . *has-sort* (*osig* (*sig* (*translate-theory* Θ))) *T S*) *insts*
= *list-all* (λ((*idn*, *S*), *T*) . *exehas-sort* (*exesorts* (*exesig* Θ)) *T S*) *insts*
⟨*proof*⟩

lemma *inst-ok-code*:
assumes *exe-wf-theory'* Θ
shows *inst-ok* (*translate-theory* Θ) *insts* = *exeinst-ok* Θ *insts*
⟨*proof*⟩

definition [*simp*]: *exeterm-ok* Θ *t* ≡ *exeterm-ok'* (*exesig* Θ) *t* ∧ *typ-of t* ≠ *None*
lemma *term-ok-code*:
assumes *exe-wf-theory'* Θ
shows *term-ok* (*translate-theory* Θ) *t* = *exeterm-ok* Θ *t*
⟨*proof*⟩

fun *exereplay'* :: *exetheory* ⇒ (*variable* × *typ*) *list* ⇒ *variable set*
⇒ *term list* ⇒ *proofterm* ⇒ *term option* **where**
exereplay' thy - - Hs (*PAxm t Tis*) = (*if exeinst-ok thy Tis* ∧ *exeterm-ok thy t*
then if t ∈ *set* (*exeaxioms-of thy*)
then Some (*forall-intro-vars* (*subst-typ' Tis t*) [])
else None else None)
| *exereplay' thy - - Hs* (*PBound n*) = *partial-nth Hs n*
| *exereplay' thy vs ns Hs* (*Abst T p*) = (*if exetyp-ok thy T*
then (*let* (*s'*,*ns'*) = *variant-variable* (*Free STR ''default''*) *ns in*
map-option (*mk-all s' T*) (*exereplay' thy* ((*s'*, *T*) # *vs*) *ns' Hs p*))
else None)
| *exereplay' thy vs ns Hs* (*Appt p t*) =
(*let rep* = *exereplay' thy vs ns Hs p in*
let t' = *subst-bvs* (*map* (λ(*x*,*y*) . *Fv x y*) *vs*) *t in*
case (*rep*, *typ-of t'*) *of*
(*Some* (*Ct s* (*Ty fun1* [*Ty fun2* [τ, *Ty propT1 Nil*], *Ty propT2 Nil*]) $ *b*), *Some* τ') ⇒
if s = *STR ''Pure.all''* ∧ *fun1* = *STR ''fun''* ∧ *fun2* = *STR ''fun''*
∧ *propT1* = *STR ''prop''* ∧ *propT2* = *STR ''prop''*
∧ τ=τ' ∧ *exeterm-ok thy t'*
then Some (*b* ⋅ *t'*) *else None*
| - ⇒ *None*)
| *exereplay' thy vs ns Hs* (*AbsP t p*) =
(*let t'* = *subst-bvs* (*map* (λ(*x*,*y*) . *Fv x y*) *vs*) *t in*
let rep = *exereplay' thy vs ns* (*t'*#*Hs*) *p in*
(*if typ-of t'* = *Some propT* ∧ *exeterm-ok thy t' then map-option* (*mk-imp t'*) *rep else None*))
| *exereplay' thy vs ns Hs* (*AppP p1 p2*) =
(*let rep1* = *Option.bind* (*exereplay' thy vs ns Hs p1*) *beta-eta-norm in*

let rep2 = Option.bind (exereplay′ thy vs ns Hs p2) beta-eta-norm in
(case (rep1, rep2) of (
Some (Ct imp (Ty fn1 [Ty prp1 [], Ty fn2 [Ty prp2 [], Ty prp3 []]]) $ A $ B),
Some A′) ⇒
if imp = STR ″Pure.imp″ ∧ fn1 = STR ″fun″ ∧ fn2 = STR ″fun″
∧ prp1 = STR ″prop″ ∧ prp2 = STR ″prop″ ∧ prp3 = STR ″prop″ ∧ A=A′
then Some B else None
| - ⇒ None))
| exereplay′ thy vs ns Hs (OfClass ty c) = (if exehas-sort (exesorts (exesig thy)) ty {c}
∧ exetyp-ok thy ty
then (case lookup (λk. k=const-of-class c) (execonst-type-of (exesig thy)) of
Some (Ty fun [Ty it [ity], Ty prop []]) ⇒
if ity = tvariable STR ‴a″ ∧ fun = STR ″fun″ ∧ prop = STR ″prop″ ∧ it = STR ″itself″
then Some (mk-of-class ty c) else None | - ⇒ None) else None)
| exereplay′ thy vs ns Hs (Hyp t) = (if t∈set Hs then Some t else None)

lemma *of-class-code1*:
assumes *exe-wf-theory′ thy*
shows *(has-sort (osig (sig (translate-theory thy))) ty {c} ∧ typ-ok (translate-theory thy) ty)*
= (exehas-sort (exesorts (exesig thy)) ty {c} ∧ exetyp-ok thy ty)
⟨*proof*⟩

lemma *of-class-code2*:
assumes *exe-wf-theory′ thy*
shows *const-type (sig (translate-theory thy)) (const-of-class c)*
= lookup (λk. k=const-of-class c) (execonst-type-of (exesig thy))
⟨*proof*⟩

lemma *replay′-code*:
assumes *exe-wf-theory′ thy*
shows *replay′ (translate-theory thy) vs ns Hs P = exereplay′ thy vs ns Hs P*
⟨*proof*⟩

abbreviation *exereplay″ thy vs ns Hs P ≡ Option.bind (exereplay′ thy vs ns Hs P) beta-eta-norm*
lemma *replay″-code*:
assumes *exe-wf-theory′ thy*
shows *replay″ (translate-theory thy) vs ns Hs P = exereplay″ thy vs ns Hs P*
⟨*proof*⟩

definition [*simp*]: *exereplay thy P ≡*
(if ∀x∈set (hyps P) . exeterm-ok thy x ∧ typ-of x = Some propT then
exereplay″ thy [] (fst ‘ (fv-Proof P ∪ FV (set (hyps P)))) (hyps P) P else None)

lemma *replay-code*:
assumes *exe-wf-theory′ thy*
shows *replay* (*translate-theory thy*) *P* = *exereplay thy P*
⟨*proof*⟩

definition *exe-replay′ e P* = *exereplay″ e* [] (*fst* ‘ *fv-Proof P*) [] *P*

definition *exe-check-proof e P res* ≡
exe-wf-theory′ e ∧ *exereplay e P* = *Some res*

lemma *exe-check-proof-iff-check-proof*:
exe-check-proof e P res ⟷ *check-proof* (*translate-theory e*) *P res*
⟨*proof*⟩

lemma *check-proof-sound*:
shows *exe-check-proof e P res* ⟹ *translate-theory e*, *set* (*hyps P*) ⊢ *res*
⟨*proof*⟩

lemma *check-proof-really-sound*:
shows *exe-check-proof e P res* ⟹ *translate-theory e*, *set* (*hyps P*) ⊩ *res*
⟨*proof*⟩

end

16 Code Generation

theory *CodeGen*
imports *ProofTerm TheoryExe CheckerExe Instances*
HOL−Library.Code-Target-Int
HOL−Library.Code-Target-Nat
begin

declare *typ-of-def*[*code*]

export-code *exe-check-proof exereplay exe-wf-theory*
Bv PBound Tv Free ExeTheory ExeSignature
in *SML* **module-name** *ExportCheck* **file-prefix** *export*

end

References

[1] S. Berghofer and T. Nipkow. Proof terms for simply typed higher order logic. In J. Harrison and M. Aagaard, editors, *Theorem Proving in Higher*

Order Logics, volume 1869 of *Lect. Notes in Comp. Sci.*, pages 38–52. Springer, 2000.

[2] T. Nipkow and S. RoSSkopf. Isabelle's metalogic: Formalization and proof checker. In G. S. A. Platzer, editor, *28th International Conference on Automated Deduction (CADE-28)*, Lect. Notes in Comp. Sci. Springer, 2021.

[3] L. C. Paulson. The foundation of a generic theorem prover. *J. Automated Reasoning*, 5:363–397, 1989.

[4] M. Wenzel. The isabelle/isar implementation. https://isabelle.in.tum.de/doc/implementation.pdf.

[5] M. Wenzel. Type classes and overloading in higher-order logic. In E. L. Gunter and A. P. Felty, editors, *Theorem Proving in Higher Order Logics, TPHOLs'97*, volume 1275 of *Lect. Notes in Comp. Sci.*, pages 307–322. Springer, 1997.