Isabelle's Metalogic: Formalization and Proof Checker

Tobias Nipkow and Simon Roßkopf

May 14, 2024

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

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* (λσ. *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* = (*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* ≡ *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* $((\text{-},\text{-}) \vdash (\text{-})\ 50)$ **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-typ a* τ $\Longrightarrow$ *wf-term* (*sig* Θ) *a*
$\Longrightarrow$ Θ, $\Gamma \vdash$ *subst-bv a B*
| *implies-intro*: *wf-theory* Θ $\Longrightarrow$ Θ, $\Gamma \vdash B$ $\Longrightarrow$ *wf-term* (*sig* Θ) *A* $\Longrightarrow$ *has-typ A propT*
$\Longrightarrow$ Θ, $\Gamma - \{A\} \vdash A \longmapsto B$
| *implies-elim*: Θ, $\Gamma_1 \vdash A \longmapsto B$ $\Longrightarrow$ Θ, $\Gamma_2 \vdash A$ $\Longrightarrow$ Θ, $\Gamma_1 \cup \Gamma_2 \vdash B$
| *of-class*: *wf-theory* Θ
$\Longrightarrow$ *const-type* (*sig* Θ) (*const-of-class c*) = *Some* (*Core.itselfT aT* $\rightarrow$ *propT*)
$\Longrightarrow$ *wf-type* (*sig* Θ) *T*
$\Longrightarrow$ *has-sort* (*osig* (*sig* Θ)) *T* {*c*}
$\Longrightarrow$ Θ, $\Gamma \vdash$ *mk-of-class T c*

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

Ensure no garbage in Θ,Γ

definition *proves'* :: *theory* $\Rightarrow$ *term set* $\Rightarrow$ *term* $\Rightarrow$ *bool* ((-,-) $\Vdash$ (-) *51*) **where**
proves' Θ Γ *t* $\equiv$ *wf-theory* Θ $\wedge$ ($\forall h \in \Gamma$. *wf-term* (*sig* Θ) *h* $\wedge$ *has-typ h propT*) $\wedge$ Θ, $\Gamma \vdash t$

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* $\Rightarrow$ *'a option* $\Rightarrow$ *'a* **where**
the-default a None = *a*
| *the-default* - (*Some b*) = *b*

abbreviation *Or* :: *'a option* $\Rightarrow$ *'a option* $\Rightarrow$ *'a option* (**infixl** *OR 60*) **where**
e1 OR e2 $\equiv$ *case e1 of None* $\Rightarrow$ *e2* | *p* $\Rightarrow$ *p*

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

by(*auto split*: *option.split*)

lemma *Or-None*: (*e1 OR e2*) = *None* ⟷ *e1* = *None* ∧ *e2* = *None*
by(*auto split*: *option.split*)

fun *lift2-option* :: ('*a* ⇒ '*b* ⇒ '*c*) ⇒ '*a option* ⇒ '*b option* ⇒ '*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-option f x y* ≠ *None* ⟷ (*x* ≠ *None* ∧ *y* ≠ *None*)
using *lift2-option.elims* **by** *blast*
lemma *lift2-option-None*: *lift2-option f x y* = *None* ⟷ (*x* = *None* ∨ *y* = *None*)
using *lift2-option.elims* **by** *blast*

Lookup functions for assoc lists

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

lemma *findD*:
find f xs = *Some p* ⟹ ∃ *x* ∈ *set xs*. *f x* = *Some p*
by(*induction xs arbitrary*: *p*) (*auto split*: *option.splits*)

lemma *find-None*:
find f xs = *None* ⟷ (∀ *x* ∈ *set xs*. *f x* = *None*)
by(*induction xs*) (*auto split*: *option.splits*)

lemma *find-ListFind*: *find f l* = *Option.bind* (*List.find* (λ*x*. *case f x of None* ⇒ *False* | - ⇒ *True*) *l*) *f*
by (*induction l*) (*auto split*: *option.split*)

lemma *List.find P l* = *Some p* ⟹ ∃ *p* ∈ *set l* . *P p*
by (*induction l*) (*auto split*: *if-splits*)

lemma *find-the-pair*:
assumes *distinct* (*map fst pairs*)
and ⋀*x y*. *x*∈*set* (*map fst pairs*) ⟹ *y*∈*set* (*map fst pairs*) ⟹ *P x* ⟹ *P y* ⟹ *x* = *y*
and (*x*,*y*) ∈ *set pairs* **and** *P x*
shows *List.find* (λ(*x*,-) . *P x*) *pairs* = *Some* (*x*,*y*)
using *assms*(*1*−*3*)
proof (*induction pairs*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons pair pairs*)
thm *Cons.prems*

```
  show ?case
  proof(cases fst pair = x)
    case True
    then show ?thesis
      using eq-key-imp-eq-value[OF Cons.prems(1,3)] assms(4) by force
  next
    case False
    hence (x,y) ∈ set pairs
      using Cons.prems(3) by fastforce
    moreover have ⋀x y. x ∈ set (map fst pairs) ⟹ y ∈ set (map fst pairs) ⟹
P x ⟹ P y ⟹ x = y
      using Cons.prems(2) by (metis list.set-intros(2) list.simps(9))
    ultimately have I: List.find (λ(x,-) . P x) pairs = Some (x,y)
      using Cons.prems(1,3) by (auto intro!: Cons.IH)
    moreover have ⋀y. y ∈ set (map fst (pair # pairs)) ⟹ P y ⟹ x = y
      using Cons.prems(2,3) assms(4) by (metis set-zip-leftD zip-map-fst-snd)
    ultimately show ?thesis
      using False by fastforce
  qed
qed

fun remdups-on :: ('a ⇒ 'a ⇒ bool) ⇒ 'a list ⇒ 'a list where
  remdups-on _ [] = []
| remdups-on cmp (x # xs) =
    (if ∃x' ∈ 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) ⟷ ¬(∃x' ∈ set xs . cmp x x') ∧ distinct-on cmp xs

lemma remdups-on (=) xs = remdups xs
  by (induction xs) auto

lemma remdups-on-antimono:
  (⋀x y . f x y ⟹ g x y) ⟹ set (remdups-on g xs) ⊆ set (remdups-on f xs)
  by (induction xs) auto

lemma remdups-on-subset-input: set (remdups-on f xs) ⊆ set xs
  by (induction xs) auto

lemma distinct-on-remdups-on: distinct-on f (remdups-on f xs)
proof (induction xs)
  case Nil
  then show ?case
    by simp
next
  case (Cons x xs)
```

then show *?case*
using *remdups-on-subset-input* **by** *fastforce*
qed

lemma *distinct-on-no-compare*: $(\bigwedge x\ y\ .\ f\ x\ y \Longrightarrow f\ y\ x) \Longrightarrow$
distinct-on f xs $\Longrightarrow$ *x*$\in$*set xs* $\Longrightarrow$ *y*$\in$*set xs* $\Longrightarrow$ *x*$\neq$*y* $\Longrightarrow$ $\neg$ *f x y*
by (*induction xs*) *auto*

fun *lookup* :: ('*a* $\Rightarrow$ *bool*) $\Rightarrow$ ('*a* $\times$ '*b*) *list* $\Rightarrow$ '*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*) $\Longrightarrow$ (*k*, *v*) $\in$ *set al* $\longleftrightarrow$ *lookup* (λ*x*. *x*=*k*) *al* = *Some v*
by (*induction al*) (*auto simp add*: *rev-image-eqI split*: *if-splits*)

lemma *lookup-None-iff*: *lookup P xs* = *None* $\longleftrightarrow$ $\neg$ ($\exists$ *x*. *x* $\in$ *set* (*map fst xs*) $\wedge$ *P x*)
by (*induction xs*) (*auto split*: *if-splits*)

lemma *find-Some*: *List.find P l* = *Some p* $\Longrightarrow$ *p*$\in$*set l* $\wedge$ *P p*
by (*induction l*) (*auto split*: *if-splits*)

lemma *find-Some-imp-lookup-Some*:
List.find (λ(*k*,-). *P k*) *xs* = *Some* (*k*,*v*) $\Longrightarrow$ *lookup P xs* = *Some v*
by (*induction xs*) *auto*

lemma *lookup-Some-imp-find-Some*:
lookup P xs = *Some v* $\Longrightarrow$ $\exists$ *x*. *List.find* (λ(*k*,-). *P k*) *xs* = *Some* (*x*,*v*)
by (*induction xs*) *auto*

lemma *lookup-None-iff-find-None*: *lookup P xs* = *None* $\longleftrightarrow$ *List.find* (λ(*k*,-). *P k*) *xs* = *None*
by (*induction xs*) *auto*

lemma *lookup-eq-order-irrelevant*:
assumes *distinct* (*map fst pairs*) **and** *distinct* (*map fst pairs'*) **and** *set pairs* = *set pairs'*
shows *lookup* (λ*x*. *x*=*k*) *pairs* = *lookup* (λ*x*. *x*=*k*) *pairs'*
proof (*cases lookup* (λ*x*. *x*=*k*) *pairs*)
case *None*
then show *?thesis* **using** *lookup-None-iff*
by (*metis assms*(*3*) *set-map*)
next
case (*Some v*)
hence (*k*,*v*)$\in$*set pairs*
using *assms*(*1*) **by** (*simp add*: *lookup-present-eq-key*)

hence *el*: $(k,v) \in set\ pairs'$ **using** *assms*(*3*) **by** *blast*
show *?thesis* **using** *lookup-present-eq-key*[*OF assms*(*2*)] *el Some* **by** *simp*
qed

lemma *lookup-Some-append-back*:
$lookup\ (\lambda x.\ x{=}k)\ insts = Some\ v \Longrightarrow lookup\ (\lambda x.\ x{=}k)\ (insts@[(k,v')]) = Some\ v$
by (*induction insts arbitrary*:) *auto*

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

lemma *lookup-in-empty*[*simp*]: $lookup\ f\ [] = None$ **by** *simp*
lemma *lookup-in-single*[*simp*]: $lookup\ f\ [(k,\ v)] = (if\ f\ k\ then\ Some\ v\ else\ None)$ **by** *simp*

lemma *lookup-present-eq-key'*: $lookup\ (\lambda x.\ x{=}k)\ al = Some\ v \Longrightarrow (k,\ v) \in set\ al$
by (*induction al*) (*auto simp add*: *rev-image-eqI split*: *if-splits*)

lemma *lookup-present-eq-key''*: $distinct\ (map\ fst\ al) \Longrightarrow lookup\ (\lambda x.\ x{=}k)\ al = Some\ v \longleftrightarrow (k,\ v) \in set\ al$
by (*induction al*) (*auto simp add*: *rev-image-eqI split*: *if-splits*)

lemma *key-present-imp-eq-lookup-finds-value*: $k \in fst\ `\ set\ al \Longrightarrow \exists\, v\ .\ lookup\ (\lambda x.\ x{=}k)\ al = Some\ v$
by (*induction al*) (*auto simp add*: *rev-image-eqI*)

lemma *list-allI*: $(\bigwedge x.\ x{\in}set\ l \Longrightarrow P\ x) \Longrightarrow list\text{-}all\ P\ l$
by (*induction l*) *auto*

lemma *map2-sym*: $(\bigwedge x\ y\ .\ f\ x\ y = f\ y\ x) \Longrightarrow map2\ f\ xs\ ys = map2\ f\ ys\ xs$
proof (*induction xs arbitrary*: *ys*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons a xs*)
then show *?case* **by** (*induction ys*) *auto*
qed

lemma *idem-map2*: **assumes** $(\bigwedge x.\ f\ x\ x = x)$ **shows** $map2\ f\ l\ l = l$
proof−
have $length\ l = length\ l$ **by** *simp*
then show $map2\ f\ l\ l = l$ **by** (*induction l l rule*: *list-induct2*) (*use assms* **in** *auto*)
qed

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−
have $length\ (rev\ xs) = length\ (rev\ ys)$ **using** *assms*(*1*) **by** *simp*
hence $P\ (rev\ (rev\ xs))\ (rev\ (rev\ ys))$
using *assms*(*2*−*3*) **by** (*induction rule*: *list-induct2*[*of rev xs rev ys*]) *simp-all*
thus *?thesis* **by** *simp*
qed

lemma *alist-map-corr*: $distinct\ (map\ fst\ al) \Longrightarrow (k,v) \in set\ al \longleftrightarrow map\text{-}of\ al\ k = Some\ v$
by *simp*

lemma *distinct-fst-imp-distinct*: $distinct\ (map\ fst\ l) \Longrightarrow distinct\ l$
by (*induction l*) *auto*

lemma *length-alist*:
assumes $distinct\ (map\ fst\ al)$ **and** $distinct\ (map\ fst\ al')$ **and** $set\ al = set\ al'$
shows $length\ al = length\ al'$
using *assms* **by** (*metis distinct-card length-map set-map*)

lemma *same-map-of-imp-same-length*:
$distinct\ (map\ fst\ ars1) \Longrightarrow distinct\ (map\ fst\ ars2) \Longrightarrow map\text{-}of\ ars1 = map\text{-}of\ ars2$
$\Longrightarrow length\ ars1 = length\ ars2$

using *length-alist map-of-inject-set* **by** *blast*

lemma *in-range-if-ex-key*: $v \in ran\ m \longleftrightarrow (\exists\, k.\ m\ k = Some\ v)$
by (*auto simp add*: *ranI ran-def*)

lemma *set-AList-delete-bound*: $set\ (AList.delete\ a\ l) \subseteq set\ l$
by (*induction l*) *auto*

lemma *list-all-clearjunk-cons*:
$list\text{-}all\ P\ (x\#(AList.clearjunk\ l)) \Longrightarrow list\text{-}all\ P\ (AList.clearjunk\ (x\#l))$
by (*induction l rule*: *AList.clearjunk.induct*) (*auto simp add*: *delete-twist*)

lemma *lookup-AList-delete*: $k' \neq k \Longrightarrow lookup\ (\lambda x.\ x = k)\ al = lookup\ (\lambda x.\ x = k)\ (AList.delete\ k'\ al)$
by (*induction al*) *auto*

lemma *lookup-AList-clearjunk*: $lookup\ (\lambda x.\ x = k)\ al = lookup\ (\lambda x.\ x = k)\ (AList.clearjunk\ al)$
proof (*induction al*)
case *Nil*
then show *?case*
by *simp*

```
next
  case (Cons a al)
  then show ?case
  proof(cases fst a=k)
    case True
    then show ?thesis
      by (metis (full-types) clearjunk.simps(2) lookup.simps(2) prod.collapse)
  next
    case False
    have lookup (λx. x = k) (AList.clearjunk (a # al))
      = lookup (λx. x = k) (a # AList.clearjunk (AList.delete (fst a) al))
      by simp
    also have ... = lookup (λx. x = k) (AList.clearjunk (AList.delete (fst a) al))
      by (metis (full-types) False lookup.simps(2) surjective-pairing)
    also have ... = lookup (λx. x = k) (AList.clearjunk al)
      by (metis False clearjunk-delete lookup-AList-delete)
    also have ... = lookup (λx. x = k) al
      using Cons.IH by auto
    also have ... = lookup (λx. x = k) (a # al)
      by (metis (full-types) False lookup.simps(2) surjective-pairing)
    finally show ?thesis
      by simp
  qed
qed

definition diff-list xs ys ≡ fold removeAll ys xs

lemma diff-list-set[simp]: set (diff-list xs ys) = set xs − set ys
  unfolding diff-list-def by (induction ys arbitrary: xs) auto

lemma diff-list-set-from-Nil[simp]: diff-list [] ys = []
  using last-in-set by fastforce

lemma diff-list-set-remove-Nil[simp]: diff-list xs [] = xs
  unfolding diff-list-def by (induction xs) auto

lemma diff-list-rec: diff-list (x # xs) ys = (if x∈set ys then diff-list xs ys else
x#diff-list xs ys)
  unfolding diff-list-def by (induction ys arbitrary: x xs) auto
lemma diff-list-order-irr: set ys = set ys' ⟹ diff-list xs ys = diff-list xs ys'
proof (induction ys arbitrary: ys' xs)
  case Nil
  then show ?case by simp
next
  case (Cons y ys)
  then show ?case
    by (induction xs arbitrary: y ys ys') (simp-all add: diff-list-rec)
qed
```

lemma *fold-Option-bind-eq-Some-start-not-None*:
fold (λ*new option* . *Option.bind option* (*f new*)) *list start* = *Some res*
⟹ *start* ≠ *None*
by (*induction list arbitrary*: *start res*)
(*fastforce split*: *option.splits if-splits simp add*: *bind-eq-Some-conv*)+

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*
by (*induction l1 arbitrary*: *start res l2*) (*use fold-Option-bind-eq-Some-start-not-None*
in
‹*fastforce split*: *option.splits if-splits simp add*: *bind-eq-Some-conv*›)+

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 (*induction list arbitrary*: *start res*)
case *Nil*
then show *?case*
by *simp*
next
case (*Cons a list*)
then show *?case*
by (*fastforce split*: *option.splits if-splits prod.splits simp add*: *bind-eq-Some-conv*)
qed

lemma *fold-Option-bind-eq-None-start-None*:
fold (λ(*x*,*y*) *option* . *Option.bind option* (*f x y*)) *list None* = *None*
by (*induction list*) (*auto split*: *option.splits if-splits prod.splits*)

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 (*induction l1 arbitrary*: *start* *l2*)
case *Nil*
then show *?case* **using** *fold-Option-bind-eq-Some-start-not-None′* **by** *fastforce*
next
case (*Cons a l1*)
then show *?case* **by** *simp*
qed

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 (*induction l1 arbitrary*: *start res l2*)
case *Nil*

then show *?case*
using *fold-Option-bind-eq-Some-start-not-None'* **by** *fastforce*
next
case (*Cons a l1*)
then show *?case* **by** *simp*
qed

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*
using *assms fold-Option-bind-eq-Some-at-each-point-Some* **by** *fast*

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*
using *fold-Option-bind-eq-Some-at-each-point-Some* **by** *fast*

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'*
using *assms fold-Option-bind-eq-Some-start-not-None' not-None-eq* **by** *fastforce*

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'*
using *assms fold-Option-bind-eq-Some-at-each-point-Some'*[**where** *xs*=*zip xs ys* **and** *ys*=[(*x*,*y*)]
and *start*=*Some subs* **and** *res*=*subs'* **and** *f*=*f*] **by** *auto*

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*}
  by (*induction σ*) *fastforce+*

lemma *tsubstT-cong*:
  (∀(*v*,*S*) ∈ *tvsT σ*. *ϱ1 v* = *ϱ2 v*) ⟹ *tsubstT σ ϱ1* = *tsubstT σ ϱ2*
  by (*induction σ*) *fastforce+*

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

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

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*)
  by (*induction T*) *auto*
lemma *finite-Sorts*[*simp*]: *finite* (*Sorts t*)
  by (*induction t*) *auto*
lemma *finite-Types*[*simp*]: *finite* (*Types t*)
  by (*induction t*) *auto*
lemma *finite-tvsT*[*simp*]: *finite* (*tvsT T*)
  by (*induction T*) *auto*
lemma *no-tvsT-imp-tsubsT-unchanged*: *tvsT T* = {} ⟹ *tsubstT T* ϱ = *T*
  by (*induction T*) (*auto simp add*: *map-idI*)
lemma *finite-fv*[*simp*]: *finite* (*fv t*)
  by (*induction t*) *auto*
lemma *finite-tvs*[*simp*]: *finite* (*tvs t*)
  by (*induction t*) *auto*

lemma *finite-FV*: *finite S* ⟹ *finite* (*FV S*)
  by (*induction S rule*: *finite-induct*) *auto*
lemma *finite-tvs-Set*: *finite S* ⟹ *finite* (*tvs-Set S*)
  by (*induction S rule*: *finite-induct*) *auto*
lemma *finite-tvsT-Set*: *finite S* ⟹ *finite* (*tvsT-Set S*)
  by (*induction S rule*: *finite-induct*) *auto*

lemma *no-tvs-imp-tsubst-unchanged*: *tvs t* = {} ⟹ *tsubst t* ϱ = *t*
  by (*induction t*) (*auto simp add*: *map-idI no-tvsT-imp-tsubsT-unchanged*)
lemma *no-fv-imp-subst-unchanged*: *fv t* = {} ⟹ *subst t* ϱ = *t*
  by (*induction t*) (*auto simp add*: *map-idI*)

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) (λx. Some (T →
x))
| typ-of1 Ts (t $ u) = Option.bind (typ-of1 Ts u) (λU. Option.bind (typ-of1 Ts t)
(λT.
    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 ⟹ has-typ1 (Ts@[T]) t rT
proof (induction arbitrary: rule: has-typ1.induct)
  case (2 i Ts)
  hence has-typ1 (Ts @ [T]) (Bv i) ((Ts@[T]) ! i)
    by (auto intro: has-typ1.intros(2))
  then show ?case
    by (simp add: 2.hyps nth-append)
qed (auto intro: has-typ1.intros) thm less-Suc-eq nth-butlast

lemma has-typ1-imp-typ-of1: has-typ1 Ts t ty ⟹ typ-of1 Ts t = Some ty
  by (induction rule: has-typ1.induct) auto

lemma typ-of1-imp-has-typ1: typ-of1 Ts t = Some ty ⟹ has-typ1 Ts t ty
proof (induction t arbitrary: Ts ty)
  case (App t u)
  from this obtain U where U: typ-of1 Ts u = Some U by fastforce
  from this App obtain T where T: typ-of1 Ts t = Some T by fastforce
  from U T App obtain T2 where T = Ty STR ''fun'' [U, T2]
    by (auto simp add: bind-eq-Some-conv intro!: has-typ1.intros
        split: if-splits typ.splits list.splits)
  from this U T show ?case using App by (auto intro!: has-typ1.intros(5))
qed (auto simp add: bind-eq-Some-conv intro!: has-typ1.intros split: if-splits)

corollary has-typ1-iff-typ-of1[iff]: has-typ1 Ts t ty ⟷ typ-of1 Ts t = Some ty
  using has-typ1-imp-typ-of1 typ-of1-imp-has-typ1 by blast
corollary has-typ-iff-typ-of[iff]: has-typ t ty ⟷ typ-of t = Some ty
  by (force simp add: has-typ-def typ-of-def)

corollary typ-of-imp-has-typ: typ-of t = Some ty ⟹ has-typ t ty
  by simp

lemma typ-of1-weaken-Ts: typ-of1 Ts t = Some ty ⟹ typ-of1 (Ts@[T]) t = Some
```

ty
using *has-typ1-weaken-Ts* **by** *simp*

lemma *typ-of1-weaken*:
assumes *typ-of1 Ts t* = *Some T*
shows *typ-of1* (*Ts*@*Ts'*) *t* = *Some T*
using *assms* **by** (*induction Ts t arbitrary*: *Ts' T rule*: *typ-of1.induct*)
(*auto split*: *if-splits simp add*: *nth-append bind-eq-Some-conv*)

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 (*induction rule*: *has-typ1.induct*)
case (*2 i Ts*)

then show *?case* **using** *tsubstT-ith* **by** (*metis has-typ1.intros*(*2*) *length-map tsubst.simps*(*3*))
qed (*auto simp add*: *tsubstT-fun-typ-dist intro*: *has-typ1.intros*)

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*
using *assms*
by (*metis has-typ1-imp-typ-of1 option.inject*)

hide-fact *typ-of-def*

lemma *typ-of-def*: *typ-of t* ≡ *typ-of1* [] *t*
by (*smt has-typ1-iff-typ-of1 has-typ-def has-typ-iff-typ-of not-None-eq*)

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*
by (*induction t arbitrary*: *n*) *auto*
lemma *not-loose-bvar-imp-not-loose-bvar1*: ¬ *loose-bvar t n* ⟹ ¬ *loose-bvar1 t n*
by (*induction t arbitrary*: *n*) *auto*

lemma *loose-bvar-iff-exist-loose-bvar1*: *loose-bvar t lev* $\longleftrightarrow$ ($\exists$ *lev'*$\geq$*lev*. *loose-bvar1 t lev'*)
by (*induction t arbitrary*: *lev*) (*auto dest*: *Suc-le-D*)

definition *is-open t* $\equiv$ *loose-bvar t 0*
abbreviation *is-closed t* $\equiv$ $\neg$ *is-open t*
definition *is-dependent t* $\equiv$ *loose-bvar1 t 0*

lemma *loose-bvar-Suc*: *loose-bvar t* (*Suc k*) $\Longrightarrow$ *loose-bvar t k*
by (*induction t arbitrary*: *k*) *auto*
lemma *loose-bvar-leq*: *k*$\geq$*p* $\Longrightarrow$ *loose-bvar t k* $\Longrightarrow$ *loose-bvar t p*
by (*induction rule*: *inc-induct*) (*use loose-bvar-Suc* **in** *auto*)

lemma *has-typ1-imp-no-loose-bvar*: *has-typ1 Ts t ty* $\Longrightarrow$ $\neg$ *loose-bvar t* (*length Ts*)
by (*induction rule*: *has-typ1.induct*) *auto*

corollary *has-typ-imp-closed*: *has-typ t ty* $\Longrightarrow$ $\neg$ *is-open t*
unfolding *is-open-def has-typ-def* **using** *has-typ1-imp-no-loose-bvar* **by** *fastforce*

corollary *typ-of-imp-closed*: *typ-of t* = *Some ty* $\Longrightarrow$ $\neg$ *is-open t*
by (*simp add*: *has-typ-imp-closed*)

Subterms

fun *exists-subterm* :: (*term* $\Rightarrow$ *bool*) $\Rightarrow$ *term* $\Rightarrow$ *bool* **where**
exists-subterm P t $\longleftrightarrow$ *P t* $\vee$ (*case t of*
(*t* $ *u*) $\Rightarrow$ *exists-subterm P t* $\vee$ *exists-subterm P u*
| *Abs ty body* $\Rightarrow$ *exists-subterm P body*
| - $\Rightarrow$ *False*)

fun *exists-subterm'* :: (*term* $\Rightarrow$ *bool*) $\Rightarrow$ *term* $\Rightarrow$ *bool* **where**
exists-subterm' P (*t* $ *u*) $\longleftrightarrow$ *P* (*t* $ *u*) $\vee$ *exists-subterm' P t* $\vee$ *exists-subterm' P u*
| *exists-subterm' P* (*Abs ty body*) $\longleftrightarrow$ *P* (*Abs ty body*) $\vee$ *exists-subterm' P body*
| *exists-subterm' P t* $\longleftrightarrow$ *P t*

lemma *exists-subterm-iff-exists-subterm'*: *exists-subterm P t* $\longleftrightarrow$ *exists-subterm' P t*
by (*induction t*) *auto*
lemma *exists-subterm* (λ*t*. *t*=*Fv idx T*) *t* $\longleftrightarrow$ (*idx*, *T*) $\in$ *fv t*
by (*induction t*) *auto*

abbreviation *occs t u* $\equiv$ *exists-subterm* (λ*s*. *t* = *s*) *u*

lemma *occs-Fv-eq-elem-fv*: *occs* (*Fv v S*) *t* $\longleftrightarrow$ (*v*, *S*) $\in$ *fv t*
by (*induction t*) *auto*

lemma *bind-fv2-unchanged*:

$\neg$*loose-bvar tm lev* $\Longrightarrow$ *bind-fv2 v lev tm = tm* $\Longrightarrow$ *v* $\notin$ *fv tm*
by (*induction v lev tm rule*: *bind-fv2.induct*) *auto*
lemma *bind-fv2-unchanged'*:
$\neg$*loose-bvar tm lev* $\Longrightarrow$ *bind-fv2 v lev tm = tm* $\Longrightarrow$ $\neg$ *occs* (*case-prod Fv v*) *tm*
by (*induction v lev tm rule*: *bind-fv2.induct*) *auto*

lemma *bind-fv2-changed*:
bind-fv2 v lev tm $\neq$ *tm* $\Longrightarrow$ *v* $\in$ *fv tm*
by (*induction v lev tm rule*: *bind-fv2.induct*) (*auto split*: *if-splits*)
lemma *bind-fv2-changed'*:
bind-fv2 v lev tm $\neq$ *tm* $\Longrightarrow$ *occs* (*case-prod Fv v*) *tm*
by (*induction v lev tm rule*: *bind-fv2.induct*) (*auto split*: *if-splits*)

corollary *bind-fv-changed*: *bind-fv v tm* $\neq$ *tm* $\Longrightarrow$ *v* $\in$ *fv tm*
unfolding *is-open-def bind-fv-def* **using** *bind-fv2-changed* **by** *simp*
corollary *bind-fv-changed'*: *bind-fv v tm* $\neq$ *tm* $\Longrightarrow$ *occs* (*case-prod Fv v*) *tm*
unfolding *is-open-def bind-fv-def* **using** *bind-fv2-changed'* **by** *simp*

corollary *bind-fv-unchanged*: (*x*,τ) $\notin$ *fv t* $\Longrightarrow$ *bind-fv* (*x*,τ) *t = t*
using *bind-fv-changed* **by** *auto*

inductive-cases *has-typ1-app-elim*: *has-typ1 Ts* (*t* \$ *u*) *R*
lemma *has-typ1-arg-typ*: *has-typ1 Ts* (*t* \$ *u*) *R* $\Longrightarrow$ *has-typ1 Ts u U* $\Longrightarrow$ *has-typ1 Ts t* (*U* $\rightarrow$ *R*)
using *has-typ1-app-elim*
by (*metis has-typ1-imp-typ-of1 option.inject typ-of1-imp-has-typ1*)

lemma *has-typ1-fun-typ*: *has-typ1 Ts* (*t* \$ *u*) *R* $\Longrightarrow$ *has-typ1 Ts t* (*U* $\rightarrow$ *R*) $\Longrightarrow$ *has-typ1 Ts u U*
by (*cases rule*: *has-typ1-app-elim*[*of Ts t u R has-typ1 Ts u U*]) (*use has-typ1-unique* **in** *auto*)

lemma *typ-of1-arg-typ*:
typ-of1 Ts (*t* \$ *u*) = *Some R* $\Longrightarrow$ *typ-of1 Ts u = Some U* $\Longrightarrow$ *typ-of1 Ts t = Some* (*U* $\rightarrow$ *R*)
using *has-typ1-iff-typ-of1 has-typ1-arg-typ* **by** *simp*

corollary *typ-of-arg*: *typ-of* (*t*\$*u*) = *Some R* $\Longrightarrow$ *typ-of u = Some T* $\Longrightarrow$ *typ-of t = Some* (*T* $\rightarrow$ *R*)
by (*metis typ-of1-arg-typ typ-of-def*)

lemma *typ-of1-fun-typ*:
typ-of1 Ts (*t* \$ *u*) = *Some R* $\Longrightarrow$ *typ-of1 Ts t = Some* (*U* $\rightarrow$ *R*) $\Longrightarrow$ *typ-of1 Ts u = Some U*
using *has-typ1-iff-typ-of1 has-typ1-fun-typ* **by** *blast*

corollary *typ-of-fun*: *typ-of* (*t*\$*u*) = *Some R* $\Longrightarrow$ *typ-of t = Some* (*U* $\rightarrow$ *R*) $\Longrightarrow$ *typ-of u = Some U*
by (*metis typ-of1-fun-typ typ-of-def*)

lemma *typ-of-eta-expand*: *typ-of f* = *Some* ($\tau \to \tau'$) $\Longrightarrow$ *typ-of* (*Abs* τ (*f* $ *Bv* *0*)) = *Some* ($\tau \to \tau'$)
using *typ-of1-weaken* **by** (*fastforce simp add*: *bind-eq-Some-conv typ-of-def*)

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*
using *assms* **by** (*induction* (*v*, *T*) *length Ts t arbitrary*: *T Ts ty rule*: *bind-fv2.induct*)
(*force simp add*: *bind-eq-Some-conv nth-append split*: *if-splits*)+

lemma *typ-of-Abs-bind-fv*:
assumes *typ-of A* = *Some ty*
shows *typ-of* (*Abs bT* (*bind-fv* (*v*, *bT*) *A*)) = *Some* (*bT* $\to$ *ty*)
using *bind-fv2-preserves-type bind-fv-def assms typ-of-def* **by** *fastforce*

corollary *typ-of-Abs-fv*:
assumes *typ-of A* = *Some ty*
shows *typ-of* (*Abs-fv v bT A*) = *Some* (*bT* $\to$ *ty*)
using *assms typ-of-Abs-bind-fv typ-of-def* **by** *simp*

lemma *typ-of-mk-all*:
assumes *typ-of A* = *Some propT*
shows *typ-of* (*mk-all x ty A*) = *Some propT*
using *typ-of-Abs-bind-fv*[*OF assms*, *of ty*] **by** (*auto simp add*: *typ-of-def*)

fun *incr-bv* :: *nat* $\Rightarrow$ *nat* $\Rightarrow$ *term* $\Rightarrow$ *term* **where**
incr-bv inc n (*Bv i*) = (*if i* $\geq$ *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*
by (*induction t n rule*: *lift.induct*) *auto*

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

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

fun *decr* :: *nat* $\Rightarrow$ *term* $\Rightarrow$ *term* **where**
decr lev (*Bv i*) = (*if i* $\geq$ *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*
by (*induction t arbitrary*: *lev*) *auto*

lemma *loose-bvar-incr-bvar*: *loose-bvar t lev* $\longleftrightarrow$ *loose-bvar* (*incr-bv inc lev t*) (*lev+inc*)
by (*induction t arbitrary*: *inc lev*) *force+*

lemma *no-loose-bvar-no-incr*[*simp*]: $\neg$ *loose-bvar t lev* $\Longrightarrow$ *incr-bv inc lev t* = *t*
by (*induction t arbitrary*: *inc lev*) *auto*

lemma *is-close-no-incr-boundvars*[*simp*]: *is-closed t* $\Longrightarrow$ *incr-boundvars inc t* = *t*
using *no-loose-bvar-no-incr* **by** (*simp add*: *incr-boundvars-def is-open-def*)

lemma *fv-incr-bv* [*simp*]: *fv* (*incr-bv inc lev t*) = *fv t*
by (*induction inc lev t rule*: *incr-bv.induct*) *auto*
lemma *fv-incr-boundvars* [*simp*]: *fv* (*incr-boundvars inc t*) = *fv t*
by (*simp add*: *incr-boundvars-def*)

lemma *loose-bvar-decr*: $\neg$ *loose-bvar t k* $\Longrightarrow$ $\neg$ *loose-bvar* (*decr k t*) *k*
by (*induction t k rule*: *loose-bvar.induct*) *auto*
lemma *loose-bvar-decr-unchanged*[*simp*]: $\neg$ *loose-bvar t k* $\Longrightarrow$ *decr k t* = *t*
by (*induction t k rule*: *loose-bvar.induct*) *auto*
lemma *is-closed-decr-unchanged*[*simp*]: *is-closed t* $\Longrightarrow$ *decr 0 t* = *t*
by (*simp add*: *is-open-def*)

fun *subst-bv1* :: *term* $\Rightarrow$ *nat* $\Rightarrow$ *term* $\Rightarrow$ *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*
by (*induction s arbitrary*: *k*) *auto*

lemma *substn-subst-n* : *subst-bv1 t n s* = *subst-bv2 t n* (*incr-bv n 0 s*)
by (*induct t arbitrary*: *n*) (*auto simp add*: *incr-boundvars-def incr-bv-combine*)

theorem *substn-subst-0*: *subst-bv1 t 0 s* = *subst-bv2 t 0 s*
by (*simp add*: *substn-subst-n*)

corollary *substn-subst-0′*: *subst-bv s t* = *subst-bv2 t 0 s*
using *subst-bv-def substn-subst-0* **by** *simp*

lemma *subst-bv2-eq* [*simp*]: *subst-bv2* (*Bv k*) *k u* = *u*
by (*simp add*:)

lemma *subst-bv2-gt* [*simp*]: *i* < *j* $\Longrightarrow$ *subst-bv2* (*Bv j*) *i u* = *Bv* (*j* − *1*)
by (*simp add*:)

lemma *subst-bv2-subst-lt* [*simp*]: $j < i \Longrightarrow$ *subst-bv2* (*Bv j*) *i u* = *Bv j*
by (*simp add*:)

lemma *lift-lift*:
$i < k + 1 \Longrightarrow$ *lift* (*lift t i*) (*Suc k*) = *lift* (*lift t k*) *i*
by (*induct t arbitrary*: *i k*) *auto*

lemma *lift-subst* [*simp*]:
$j < i + 1 \Longrightarrow$ *lift* (*subst-bv2 t j s*) *i* = *subst-bv2* (*lift t* (*i* + *1*)) *j* (*lift s i*)
proof (*induction t arbitrary*: *i j s*)
case (*Abs T t*)
then show *?case*
by (*simp-all add*: *diff-Suc lift-lift split*: *nat.split*)
(*metis One-nat-def Suc-eq-plus1 lift-def lift-lift zero-less-Suc*)
qed (*simp-all add*: *diff-Suc lift-lift split*: *nat.split*)

lemma *lift-subst-bv2-subst-lt*:
$i < j + 1 \Longrightarrow$ *lift* (*subst-bv2 t j s*) *i* = *subst-bv2* (*lift t i*) (*j* + *1*) (*lift s i*)
proof (*induction t arbitrary*: *i j s*)
case (*Abs x1 t*)
then show *?case*
using *lift-lift* **by** *force*
qed (*auto simp add*: *lift-lift*)

lemma *subst-bv2-lift* [*simp*]:
subst-bv2 (*lift t k*) *k s* = *t*
by (*induct t arbitrary*: *k s*) *simp-all*

lemma *subst-bv2-subst-bv2*:
$i < j + 1 \Longrightarrow$ *subst-bv2* (*subst-bv2 t* (*Suc j*) (*lift v i*)) *i* (*subst-bv2 u j v*)
= *subst-bv2* (*subst-bv2 t i u*) *j v*
proof(*induction t arbitrary*: *i j u v*)
case (*Abs s T t*)
then show *?case*
by (*smt Suc-mono add.commute lift-lift lift-subst-bv2-subst-lt plus-1-eq-Suc subst-bv2.simps*(*2*) *zero-less-Suc*)
qed (*use subst-bv2-lift* **in** ‹*auto simp add*: *diff-Suc lift-lift* [*symmetric*] *lift-subst-bv2-subst-lt split*: *nat.split*›)

hide-fact (**open**) *subst-bv-def*
lemma *subst-bv-def*: *subst-bv u t* ≡ *subst-bv1 t 0 u*
by (*simp add*: *substn-subst-0′ substn-subst-n*)

fun *subst-bvs1* :: *term* ⇒ *nat* ⇒ *term list* ⇒ *term* **where**
subst-bvs1 (*Bv n*) *lev args* = (*if n* < *lev*
then Bv n
else if n − *lev* < *length args*

then incr-boundvars lev (nth args (n−lev))
else Bv (n − length args))
| *subst-bvs1 (Abs T body) lev args = Abs T (subst-bvs1 body (lev+1) args)*
| *subst-bvs1 (f $ t) lev args = subst-bvs1 f lev args $ subst-bvs1 t lev args*
| *subst-bvs1 t - - = t*

definition *subst-bvs args t ≡ subst-bvs1 t 0 args*

lemma *subst-bvs-App*[*simp*]: *subst-bvs args (s$t) = subst-bvs args s $ subst-bvs args t*
by (*auto simp add*: *subst-bvs-def*)

lemma *subst-bv1-special-case-subst-bvs1*: *subst-bvs1 t lev* [*x*] = *subst-bv1 t lev x*
by (*induction t lev* [*x*] *arbitrary*: *x rule*: *subst-bvs1.induct*) *auto*

lemma *no-loose-bvar-imp-no-subst-bv1*: ¬*loose-bvar t lev* ⟹ *subst-bv1 t lev u = t*
by (*induction t arbitrary*: *lev*) *auto*
lemma *no-loose-bvar-imp-no-subst-bvs1*: ¬*loose-bvar t lev* ⟹ *subst-bvs1 t lev us = t*
by (*induction t arbitrary*: *lev*) *auto*

lemma *subst-bvs1-step*:
assumes ¬ *loose-bvar t lev*
shows *subst-bvs1 t lev (args@*[*u*]*) = subst-bv1 (subst-bvs1 t lev args) lev u*
using *assms* **by** (*induction t arbitrary*: *lev args u*) *auto*

corollary *closed-subst-bv-no-change*: *is-closed t* ⟹ *subst-bv u t = t*
unfolding *is-open-def subst-bv-def no-loose-bvar-imp-no-subst-bv1* **by** *simp*

lemma *is-variable-imp-incr-bv-unchanged*: *incr-bv inc lev (Fv v T) = (Fv v T)*
by *simp*
lemma *is-variable-imp-incr-boundvars-unchganged*: *incr-boundvars inc (Fv v T) = (Fv v T)*
using *is-variable-imp-incr-bv-unchanged incr-boundvars-def* **by** *simp*

lemma *loose-bvar-subst-bv1*:
¬ *loose-bvar (subst-bv1 t lev u) lev* ⟹ ¬ *loose-bvar t (Suc lev)*
by (*induction t lev u rule*: *subst-bv1.induct*) *auto*
lemma *is-closed-subst-bv*: *is-closed (subst-bv u t)* ⟹ ¬ *loose-bvar t 1*
by (*simp add*: *is-open-def loose-bvar-subst-bv1 subst-bv-def*)

lemma *subst-bv1-bind-fv2*:
assumes ¬ *loose-bvar t lev*
shows *subst-bv1 (bind-fv2 (v, T) lev t) lev (Fv v T) = t*
using *assms* **by** (*induction t arbitrary*: *lev*) (*use is-variable-imp-incr-boundvars-unchganged* **in** *auto*)

corollary *subst-bv-bind-fv*:

assumes *is-closed t*
shows *subst-bv* (*Fv v T*) (*bind-fv* (*v*, *T*) *t*) = *t*
unfolding *bind-fv-def subst-bv-def* **using** *assms subst-bv1-bind-fv2 is-open-def*
by *blast*

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*
using *assms subst-bv-bind-fv* **by** *simp*

lemma *typ-of1-imp-no-loose-bvar*: *typ-of1 Ts t* = *Some ty* ⟹ ¬ *loose-bvar t* (*length Ts*)
by (*simp add*: *has-typ1-imp-no-loose-bvar*)

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*
using *assms*
proof (*induction f length Ts u arbitrary*: *uty fty Ts rule*: *subst-bv1.induct*)
case (*1 i arg*)
then show *?case*
using *no-loose-bvar-no-incr typ-of1-imp-no-loose-bvar typ-of1-weaken*
by (*force simp add*: *bind-eq-Some-conv incr-boundvars-def nth-append typ-of-def split*: *if-splits*)
next
case (*2 a T body arg*)
then show *?case*
by (*simp add*: *bind-eq-Some-conv typ-of-def*) (*smt append-Cons bind-eq-Some-conv length-Cons*)
qed (*auto simp add*: *bind-eq-Some-conv*)

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*)
by (*metis* (*no-types*, *lifting*) *bind.bind-lzero the-default.elims typ-of1.simps*(*5*) *typ-of1-arg-typ*)

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*
using *typ-of1-split-App assms* **by** *blast*

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
  using assms by (induction t arbitrary: ty Ts Ts' lev)
    (fastforce simp add: nth-append bind-eq-Some-conv min-def split: if-splits)+

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
  using assms typ-of1-incr-bv[where lev=0] by simp

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
  using assms
proof (induction t length Ts' u arbitrary: tty uty Ts Ts' rule: subst-bv1.induct)
next
  case (2 a T body arg)
  then show ?case
    by (simp add: bind-eq-Some-conv) (metis append-Cons length-Cons)
qed (auto simp add: bind-eq-Some-conv nth-append incr-boundvars-def
    typ-of1-incr-bv-lev0 split: if-splits)


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
  using assms
proof (induction f length Ts' u arbitrary: fty uty Ts Ts' rule: subst-bv1.induct)
  case (1 i arg)
  from 1 consider (LT) (length Ts' − 1) < i | (EQ) (length Ts' − 1) = i | (GT)
(length Ts' − 1) > i
    using linorder-neqE-nat by blast
  then show ?case
      by cases (metis 1.prems append-assoc append-butlast-last-id length-butlast
typ-of1-subst-bv-gen)+
next
  case (2 a T body arg)
  then show ?case
   by (metis append.assoc append-butlast-last-id length-butlast typ-of1-subst-bv-gen)
next
  case (3 f t arg)
  then show ?case
   by (auto simp add: bind-eq-Some-conv nth-append incr-boundvars-def subst-bv-def

       split: if-splits)
qed auto
```

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*
using *assms typ-of1-subst-bv-gen*
by (*metis append.left-neutral append-Cons list.size*(*3*))

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*
using *assms typ-of1-subst-bv-gen'*
by (*auto simp add*: *bind-eq-Some-conv subst-bv-def*)

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*
by (*induction t arbitrary*: *Ts Ts'*) (*simp-all*, *metis*+)

lemma *typ-of1-drop-extra-bounds*:
¬*loose-bvar t* (*length Ts*)
⟹ *typ-of1* (*Ts*@*rest*) *t* = *typ-of1 Ts t*
by (*induction Ts t arbitrary*: *rest rule*: *typ-of1.induct*) (*fastforce simp add*: *nth-append*)+

lemma *typ-of-betaply*:
assumes *typ-of t* = *Some* (*uty* → *tty*) *typ-of u* = *Some uty*
shows *typ-of* (*t* ∙ *u*) = *Some tty*
proof (*cases t*)
case (*Abs T t*)
then show *?thesis*
proof (*cases is-open t*)
case *True*
then show *?thesis*
unfolding *is-open-def* **using** *assms Abs typ-of1-subst-bv*
apply (*simp add*: *bind-eq-Some-conv subst-bv-def typ-of-def*)
by (*metis append-Nil list.size*(*3*) *typ-of-def*)
next
case *False*
hence *typ-of1* [*uty*] *t* = *Some tty* **using** *assms*(*1*)
by (*auto simp add*: *bind-eq-Some-conv typ-of-def is-open-def Abs*)

then show *?thesis*
using *assms False no-loose-bvar-imp-no-subst-bv1*
apply (*simp add*: *bind-eq-Some-conv typ-of-def is-open-def subst-bv-def Abs*)
using *no-Bv-Type-param-irrelevant-typ-of*
using *typ-of1-drop-extra-bounds*
by (*metis list.size*(*3*) *self-append-conv2*)

qed
qed (*use assms* **in** ‹*simp-all add*: *typ-of-def*›)

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*)) = (¬ *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 ¬ *loose-bvar t lev* ⟹ *decr lev t* = *t*
by (*induction t arbitrary*: *lev*) *auto*

lemma *decr-incr-bv1*: *decr lev* (*incr-bv 1 lev t*) = *t*
by (*induction t arbitrary*: *lev*) *auto*

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*
by (*induction lev t rule*: *decr.induct*) *auto*

lemma *loose-bvar1-decr*: *lev* > *0* ⟹ ¬ *loose-bvar1 t* (*Suc lev*) ⟹ ¬ *loose-bvar1* (*decr lev t*) *lev*
by (*induction lev t arbitrary*: *rule*: *decr.induct*) *auto*

lemma *loose-bvar1-decr′*:
¬ *loose-bvar1 t* (*Suc lev*) ⟹ ¬ *loose-bvar1 t lev* ⟹ ¬ *loose-bvar1* (*decr lev t*) *lev*
by (*induction lev t arbitrary*: *rule*: *decr.induct*) *auto*

lemma *eta-reducible-Abs1*: ¬ *eta-reducible* (*Abs T* (*t* $ *Bv 0*)) ⟹ ¬ *eta-reducible t* **by** *simp*

lemma *eta-reducible-Abs2*:
assumes ¬ (∃*f*. *t*=*f* $ *Bv 0*) ¬ *eta-reducible* (*Abs T t*)
shows ¬ *eta-reducible t*
proof (*cases t*)
case (*Abs T body*)
then show *?thesis* **using** *assms*(*2*) **by** (*cases body*) *auto*
next

```
  case (App f u)
 then show ?thesis using assms less-imp-Suc-add by (cases f; cases u) fastforce+

qed auto

lemma eta-reducible-Abs: ¬ eta-reducible (Abs T t) ⟹ ¬ eta-reducible t
  using eta-reducible-Abs1 eta-reducible-Abs2
  by (metis eta-reducible.simps(11) eta-reducible.simps(14))

lemma loose-bvar1-decr'': loose-bvar1 t lev ⟹ lev < lev'⟹ loose-bvar1 (decr
lev' t) lev
  by (induction t arbitrary: lev lev') auto
lemma loose-bvar1-decr''': loose-bvar1 t (Suc lev) ⟹ lev' ≤ lev ⟹ loose-bvar1
(decr lev' t) lev
  by (induction t arbitrary: lev lev') auto

lemma loose-bvar1-decr'''': ¬ loose-bvar1 t lev' ⟹ lev' ≤ lev  ⟹ ¬ loose-bvar1
t (Suc lev)
  ⟹ ¬ loose-bvar1 (decr lev' t) lev
  by (induction lev t arbitrary: lev' rule: decr.induct) auto

lemma not-eta-reducible-decr:
  ¬ eta-reducible t ⟹ ¬ loose-bvar1 t lev ⟹ ¬ eta-reducible (decr lev t)
proof (induction lev t arbitrary: rule: decr.induct)
  case (2 lev T body)
  hence ¬ eta-reducible body using eta-reducible-Abs by blast
  hence I: ¬ eta-reducible (decr (lev + 1) body) using 2.IH
    using 2.prems(2) by simp

  then show ?case
  proof(cases body)
    case (App f u)
    note app = this
    then show ?thesis
    proof (cases u)
      case (Bv n)
      then show ?thesis
      proof (cases n)
        case 0
        have is-dependent f ¬ eta-reducible f
          using 0 2.prems(1) App Bv eta-reducible.simps(1) by blast+
        hence loose-bvar1 f 0 by (simp add: is-dependent-def)
        hence loose-bvar1 (decr (Suc lev) f) 0 using loose-bvar1-decr'' by simp
       then show ?thesis using I by (auto simp add: 0 Bv App is-dependent-def)
      next
        case (Suc nat)
        then show ?thesis
          using 2 App Bv
       by (auto elim: eta-reducible.elims(2) simp add: Suc Bv App is-dependent-def)
```

```
    qed
  next
    case (Abs T t)
    then show ?thesis
      using I by (auto split: if-splits simp add: App is-dependent-def)
  qed (use I in ‹auto split: if-splits simp add: App is-dependent-def›)
 qed (auto split: if-splits simp add: is-dependent-def)
qed auto


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
  by pat-completeness auto

lemma eta-norm-reduces-depth: eta-norm-dom t ⟹ depth (eta-norm t) <= depth
t
  by (induction t rule: eta-norm.pinduct)
     (use depth-decr in ‹fastforce simp add: eta-norm.psimps eta-norm.domintros
is-dependent-def
      split: term.splits nat.splits›)+

termination eta-norm
proof (relation measure depth)
  fix T body t u n
   assume asms: eta-norm body = t $ u u = Bv n n = 0 ¬ is-dependent t
eta-norm-dom body
  have depth t < depth (t $ Bv 0) by auto
 moreover have depth (eta-norm body) ≤ depth body using asms eta-norm-reduces-depth
by blast
  ultimately show (t, Abs T body) ∈ measure depth using asms by (auto simp
add: eta-norm.psimps)
qed simp-all

lemma loose-bvar1-eta-norm: loose-bvar1 t lev ⟹ loose-bvar1 (eta-norm t) lev
  by (induction t arbitrary: lev rule: eta-norm.induct)
    (use loose-bvar1-decr''' in ‹(fastforce split: term.splits nat.splits)+›)

lemma loose-bvar1-eta-norm': ¬ loose-bvar1 t lev ⟹ ¬ loose-bvar1 (eta-norm t)
lev
proof (induction t arbitrary: lev rule: eta-norm.induct)
  case (1 T body)
  hence ¬ loose-bvar1 body (Suc lev) by simp
  hence I: ¬ loose-bvar1 (eta-norm body) (Suc lev) using 1 by simp
  then show ?case
  proof (cases body)
```

```
    case (Abs ty b)
    show ?thesis
      using I loose-bvar1-decr''''
    by (auto split: term.splits nat.splits if-splits simp add: 1.IH(2) is-dependent-def)
  next
    case (App T t)
    then show ?thesis using 1 I loose-bvar1-decr''''
      by (fastforce split: term.splits nat.splits if-splits simp add: is-dependent-def)
  qed (auto split: term.splits nat.splits simp add: is-dependent-def)
qed (auto split: term.splits nat.splits simp add: is-dependent-def)

lemma not-eta-reducible-eta-norm: ¬ eta-reducible (eta-norm t)
proof (induction t rule: eta-norm.induct)
  case (1 T body)
  then show ?case
  proof (cases eta-norm (body))
    case (Abs T t)
    then show ?thesis using 1 by auto
  next
    case (App f u)
    then show ?thesis
    proof (cases u = Bv 0)
      case True
      note u = this
      then show ?thesis
      proof (cases is-dependent f)
        case True
        then show ?thesis
            using 1 App u by (auto simp add: is-dependent-def split: term.splits
nat.splits if-splits)
      next
        case False
        have ¬ eta-reducible f using 1 App u by simp
        hence ¬ eta-reducible (eta-norm f)
          by (simp add: 1.IH(2) App False u)
        have ¬ loose-bvar1 f 0
          using False is-dependent-def by blast
        hence  ¬ loose-bvar1 (eta-norm f) 0
          using  loose-bvar1-eta-norm' by blast
        show ?thesis
             using 1 App u False not-eta-reducible-decr loose-bvar1-eta-norm ‹¬
loose-bvar1 (eta-norm f) 0›
          by (auto simp add: is-dependent-def split: term.splits nat.splits if-splits)
      qed
    next
      case False
      then show ?thesis using 1 App by (auto simp add: is-dependent-def
          split: term.splits nat.splits if-splits)
    qed
```

qed *auto*
qed *auto*

lemma *not-eta-reducible-imp-eta-norm-no-change*: ¬ *eta-reducible t* ⟹ *eta-norm t* = *t*
by (*induction t rule*: *eta-norm.induct*) (*auto simp add*: *eta-reducible-Abs is-dependent-def*

split: *term.splits nat.splits*)

lemma *eta-norm-collapse*: *eta-norm* (*eta-norm t*) = *eta-norm t*
using *not-eta-reducible-imp-eta-norm-no-change not-eta-reducible-eta-norm* **by** *blast*

lemma *typ-of1-decr*: *typ-of1* (*Ts*@[*T*]@*Ts'*) *t* = *Some ty* ⟹ ¬ *loose-bvar1 t* (*length Ts*)
⟹ *typ-of1* (*Ts*@*Ts'*) (*decr* (*length Ts*) *t*) = *Some ty*
proof (*induction t arbitrary*: *Ts T Ts' ty*)
case (*Abs bT t*)
then show *?case*
by (*simp add*: *bind-eq-Some-conv*) (*metis append-Cons length-Cons*)
qed (*auto split*: *if-splits simp add*: *bind-eq-Some-conv nth-append*)

lemma *typ-of1-decr-gen*: *typ-of1* (*Ts*@[*T*]@*Ts'*) *t* = *tyo* ⟹ ¬ *loose-bvar1 t* (*length Ts*)
⟹ *typ-of1* (*Ts*@*Ts'*) (*decr* (*length Ts*) *t*) = *tyo*
proof (*induction t arbitrary*: *Ts T Ts' tyo*)
case (*Abs T t*)
then show *?case*
by (*simp add*: *bind-eq-Some-conv*) (*metis append-Cons length-Cons*)
next
case (*App t1 t2*)
then show *?case* **by** *simp*
qed (*auto split*: *if-splits simp add*: *bind-eq-Some-conv nth-append*
split: *option.splits*)

lemma *typ-of1-decr-gen'*: *typ-of1* (*Ts*@*Ts'*) (*decr* (*length Ts*) *t*) = *tyo*⟹ ¬ *loose-bvar1 t* (*length Ts*)
⟹ *typ-of1* (*Ts*@[*T*]@*Ts'*) *t* = *tyo*
proof (*induction t arbitrary*: *Ts T Ts' tyo*)
case (*Abs T t*)
then show *?case*
by (*simp add*: *bind-eq-Some-conv*) (*metis append-Cons length-Cons*)
qed (*auto split*: *if-splits simp add*: *bind-eq-Some-conv nth-append*
split: *option.splits*)

lemma *typ-of1-eta-norm*: *typ-of1 Ts t* = *Some ty* ⟹ *typ-of1 Ts* (*eta-norm t*) = *Some ty*
proof (*induction Ts t arbitrary*: *ty rule*: *typ-of1.induct*)

```
  case (4 Ts T body)
  then show ?case
  proof(cases eta-norm body)
    case (App f u)
    then show ?thesis

    proof (cases u)
      case (Bv n)
      then show ?thesis
      proof (cases n)
        case 0
        then show ?thesis
        proof (cases is-dependent f)
          case True
          hence eta-norm (Abs T body) = Abs T (f $ Bv 0)
             by (auto simp add: App 0 4.IH Bv bind-eq-Some-conv is-dependent-def
split: nat.splits)
          then show ?thesis
           using 4 by (force simp add: 0 Bv App is-dependent-def bind-eq-Some-conv
split: if-splits)
        next
          case False

          hence simp: eta-norm (Abs T body) = decr 0 (eta-norm f)
           by (auto simp add: App 0 4.IH Bv bind-eq-Some-conv bind-eq-None-conv
               is-dependent-def split: nat.splits)

          obtain bT where bT: typ-of1 (T # Ts) body = Some bT
            using 4.prems by fastforce
          hence typ-of1 (T # Ts) (eta-norm body) = Some bT
            using 4.IH by blast
          moreover have T → bT = ty
            using 4.prems bT by auto
          ultimately have typ-of1 (T#Ts) f = Some ty
         by (metis 0 App Bv length-Cons nth-Cons-0 typ-of1.simps(2) typ-of1-arg-typ
zero-less-Suc)
          hence typ-of1 Ts (decr 0 f) = Some ty
              by (metis False append-Cons append-Nil is-dependent-def list.size(3)
typ-of1-decr)
          hence typ-of1 Ts (decr 0 (eta-norm f)) = Some ty
                by (metis App eta-reducible.simps(11) not-eta-reducible-eta-norm
not-eta-reducible-imp-eta-norm-no-change)

          then show ?thesis
            by(auto simp add: App 0 Bv False)
        qed
      next
        case (Suc nat)
        then show ?thesis
```

```
      using 4 apply (simp add: App 4.IH Bv bind-eq-Some-conv split: option.splits)
        using option.sel by fastforce
      qed
  qed (use 4 in ‹fastforce simp add: bind-eq-Some-conv nth-append split: if-splits›)+
 qed (use 4 in ‹fastforce simp add: bind-eq-Some-conv nth-append split: if-splits›)+
next
 case (5 Ts f u)
 then show ?case
  apply (clarsimp split: term.splits typ.splits if-splits nat.splits option.splits
      simp add: bind-eq-Some-conv)
  by blast
qed (auto split: term.splits typ.splits if-splits nat.splits option.splits
   simp add: bind-eq-Some-conv)

corollary typ-of-eta-norm: typ-of t = Some ty ⟹ typ-of (eta-norm t) = Some ty
 using typ-of1-eta-norm typ-of-def by simp

lemma typ-of-Abs-body-typ: typ-of1 Ts (Abs T t) = Some ty ⟹ ∃ rty. ty = (T →
rty)
 by (metis (no-types, lifting) bind-eq-Some-conv option.sel typ-of1.simps(4))
lemma typ-of-Abs-body-typ': typ-of1 Ts (Abs T t) = Some ty
 ⟹ ∃ rty. ty = (T → rty) ∧ typ-of1 (T#Ts) t = Some rty
 by (metis (no-types, lifting) bind-eq-Some-conv option.sel typ-of1.simps(4))

lemma typ-of-beta-redex-arg: typ-of (Abs T s $ t) ≠ None ⟹ typ-of t = Some T
 by (metis list.inject not-Some-eq typ.inject(1) typ-of1-split-App typ-of-Abs-body-typ'
typ-of-def)

lemma [partial-function-mono]: option.mono-body
       (λbeta-norm. map-option (Abs T) (beta-norm t))
 by (smt flat-ord-def fun-ord-def map-option-is-None monotone-def)
lemma [partial-function-mono]: option.mono-body
       (λbeta-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(standard, goal-cases)
 case (1 a b)
 then show ?case
 proof(cases a x; cases b x, simp-all add: flat-ord-def fun-ord-def, goal-cases)
  case (1 a)
```

```
    then show ?case
      by (metis option.discI)
  next
    case (2 r s)
    then show ?case
      apply (cases r; cases s)
      apply (simp-all add: flat-ord-def fun-ord-def)
    apply (metis option.distinct option.inject option.sel term.distinct term.inject)+
      done
  qed
qed


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]: beta-norm.simps
declare beta-norm-simps[code]

lemma not-beta-reducible-imp-beta-norm-unchanged: ¬ beta-reducible t ⟹ beta-norm
t = Some t
proof (induction t)
  case (App t u)
  then show ?case by (cases t) auto
qed auto

lemma not-beta-reducible-decr: ¬ beta-reducible t ⟹ ¬ beta-reducible (decr n t)
  by (induction t arbitrary: n rule: beta-reducible.induct) auto

lemma ¬ beta-reducible t ⟹ eta-norm t = t' ⟹ ¬ beta-reducible t'
proof (induction t arbitrary: t' rule: eta-norm.induct)
  case (1 T body)
  show ?case
  proof(cases eta-norm body)
    case (Abs T' t)
    then show ?thesis using 1 by fastforce
  next
    case (App f u)
    note oApp = this
    show ?thesis
    proof(cases u)
      case (Bv n)
```

```
      show ?thesis
      proof(cases n)
        case 0
        then show ?thesis
        proof(cases is-dependent f)
          case True
          then show ?thesis
            using 1 oApp Bv 0 apply simp
            using beta-reducible.simps(2) by blast
        next
          case False
          obtain body' where body': eta-norm body = body' by simp
          obtain f' where f': eta-norm f = f' by simp
          moreover have t': t' = decr 0 f' using 1.prems(2)[symmetric] oApp Bv
0 False f' by simp

          moreover have ¬ beta-reducible t'
          proof–
            have ¬ beta-reducible (f $ Bv 0)
              using 1.IH(1) 1 oApp Bv 0 by simp
            hence ¬ beta-reducible (decr 0 (f' $ Bv 0))
              by (metis eta-reducible.simps(11) f' not-beta-reducible-decr
                not-eta-reducible-eta-norm not-eta-reducible-imp-eta-norm-no-change
oApp)
            hence ¬ beta-reducible (decr 0 f' $ Bv 0) by simp
            hence ¬ beta-reducible (decr 0 f') by (auto elim: beta-reducible.elims)
            thus ?thesis using t' by simp
          qed
          ultimately show ?thesis by blast
        qed
      next
        case (Suc nat)
        then show ?thesis using 1 oApp Bv by auto
      qed
    qed (use 1 oApp in auto)
  qed (use 1 in auto)
next
  case (2 f u)
 hence ¬ beta-reducible f ¬ beta-reducible u by (blast elim!: beta-reducible.elims(3))+
  moreover obtain f' u' where eta-norm f = f' eta-norm u = u' by simp-all
  ultimately have ¬ beta-reducible f' ¬ beta-reducible u' using 2.IH by simp-all
  show ?case
  proof(cases t')
    case (App l r)
    then show ?thesis
      using 2.IH(2) 2.prems(2) ‹¬ beta-reducible u› ‹¬ beta-reducible f'› ‹eta-norm
f = f'› 2(3)
      by (auto elim: beta-reducible.elims(3))
  qed (use 2.prems(2) in auto)
```

qed *auto*

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*
  by (*induction t*) *auto*

lemma *fv-iff-occs*: (*x*,*τ*) ∈ *fv t* ⟷ *occs* (*Fv x τ*) *t*
  by (*induction t*) *auto*

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* **by** (*simp add*: *list-comb-def list-comb′-def*)

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*
by (*induction* (*t*,*ts*) *arbitrary*: *t ts rule*: *strip-comb-imp.induct*) *simp-all*
corollary *fst* (*strip-comb t*) = *head-of t*
using *fst-strip-comb-imp-eq-head-of* **by** (*simp add*: *strip-comb-def*)

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*)
by (*cases t*) *simp-all*
corollary *not-is-app-imp-strip-com-unchanged*: ¬ *is-app t* ⟹ *strip-comb t* = (*t*,[])

unfolding *strip-comb-def* **using** *not-is-app-imp-strip-com-imp-unchanged* **.**

lemma *list-comb-fuse*: *list-comb* (*list-comb* (*t*,*ts*), *ss*) = *list-comb* (*t*,*ts*@*ss*)
unfolding *list-comb-def* **by** *simp*

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
  by (induction rule: typ.induct) (simp-all add: map-idI)

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
  by (induction rule: term.induct) simp-all

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

lemma map-types-id[simp]: map-types id t = t
  by (induction t) simp-all
lemma map-types-id'[simp]: map-types (λa . a) t = t
  using map-types-id by (simp add: id-def)


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 (λx . 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 (λx . 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*
unfolding *add-tvarsT'-def*
proof (*induction* *T* *arbitrary*: *acc*)
case (*Ty* *n* *Ts*)
then show *?case* **by** (*induction* *Ts* *arbitrary*: *acc*) *auto*
qed *auto*

lemma *add-tvars'-tvs-pre*[*simp*]: *set* (*add-tvars'* *t* *acc*) = *set acc* ∪ *tvs* *t*
by (*induction* *t* *arbitrary*: *acc*) (*auto simp add*: *add-tvars'-def fold-types-def*)

lemma *add-tvarsT* *T* *acc* = *acc* ∪ *tvsT* *T*
unfolding *add-tvarsT-def*
proof (*induction* *T* *arbitrary*: *acc*)
case (*Ty* *n* *Ts*)
then show *?case* **by** (*induction* *Ts* *arbitrary*: *acc*) *auto*
qed *auto*

lemma *add-vars'-fv-pre*: *set* (*add-vars'* *t* *acc*) = *set acc* ∪ *fv* *t*
unfolding *add-vars'-def* **by** (*induction* *t* *arbitrary*: *acc*) *auto*
corollary *add-vars'-fv*: *set* (*add-vars'* *t* []) = *fv* *t*
using *add-vars'-fv-pre* **by** *simp*

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*
  by (*induction n t rule*: *strip-all-multiple-vars.induct*) *auto*
lemma *n*≥*length* (*strip-all-vars t*) ⟹ *strip-all-multiple-body n t* = *strip-all-body*
t
 by (*induction n t rule*: *strip-all-multiple-vars.induct*) (*auto elim*!: *strip-all-vars.elims*)

lemma *length-strip-all-multiple-vars*: *length* (*strip-all-multiple-vars n t*) ≤ *n*
  by (*induction n t rule*: *strip-all-multiple-vars.induct*) *auto*

lemma *prefix-strip-all-multiple-vars*: *prefix* (*strip-all-multiple-vars n t*) (*strip-all-vars*
t)
  unfolding *prefix-def* **by** (*induction n t rule*: *strip-all-multiple-vars.induct*) *auto*

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* **by** (*simp add*: *mk-all-list-def*)

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*
  apply (*cases t*) **apply** *simp-all*
 subgoal for *f u* **apply** (*cases f*; *cases u*) **by** (*auto elim*: *is-all.elims split*: *if-splits*)

  done

lemma *is-all t* ⟹ *hd* (*strip-all-vars t*) = *the* (*strip-all-single-var t*)
  by (*auto elim*: *is-all.elims*)

lemma *strip-all-body-single-simp*[*simp*]: *strip-all-body* (*strip-all-single-body t*) =
strip-all-body t

by (*induction t rule*: *strip-all-body.induct*) *auto*
lemma *strip-all-body-single-simp'*[*simp*]: *strip-all-single-body* (*strip-all-body t*) = *strip-all-body t*
by (*induction t rule*: *strip-all-body.induct*) *auto*

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*
by (*induction t arbitrary*: *T rule*: *strip-all-vars.induct*) (*auto split*: *if-splits*)

lemma *is-all-iff-strip-all-vars-not-empty*: *is-all t* ⟷ *strip-all-vars t* ≠ []
apply (*cases t*) **apply** *simp-all*
subgoal for *f u* **apply** (*cases f*; *cases u*) **by** (*auto elim*: *strip-all-vars.elims is-all.elims split*: *if-splits*)
done

lemma *strip-all-vars-bind-fv*:
strip-all-vars (*bind-fv2 v lev t*) = (*strip-all-vars t*)
by (*induction t arbitrary*: *lev rule*: *strip-all-vars.induct*) *auto*

lemma *strip-all-vars-mk-all*[*simp*]: *strip-all-vars* (*mk-all s ty t*) = *ty* # *strip-all-vars t*
using *bind-fv-def strip-all-vars-bind-fv typ-of-def* **by** *auto*

lemma *strip-all-vars-mk-all-list*:
¬*is-all t* ⟹ *strip-all-vars* (*mk-all-list l t*) = *rev* (*map snd l*)
proof (*induction l rule*: *rev-induct*)
case *Nil*
then show *?case* **using** *is-all-iff-strip-all-vars-not-empty* **by** *simp*
next
case (*snoc v vs*)
hence *I*: *strip-all-vars* (*mk-all-list vs t*) = *rev* (*map snd vs*) **by** *simp*
obtain *s ty* **where** *v*: *v* = (*s*,*ty*) **by** *fastforce*

have *strip-all-vars* (*mk-all-list* (*vs* @ [*v*]) *t*)
= *strip-all-vars* (*mk-all s ty* (*mk-all-list vs t*))
by (*auto simp add*: *mk-all-list-def v*)
also have ... = *ty* # *strip-all-vars* (*mk-all-list vs t*)
using *strip-all-vars-mk-all*[*of ty s mk-all-list vs t*] **by** *blast*
also have ... = *ty* # *rev* (*map snd vs*)
by (*simp add*: *I*)
also have ... = *rev* (*map snd* (*vs* @ [*v*]))
using *v* **by** *simp*
finally show *?case* **.**
qed

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*
using *assms* **proof** (*induction t arbitrary*: *lev*)
case (*Bv x*)
then show *?case*
using *loose-bvar-iff-exist-loose-bvar1 no-loose-bvar-imp-no-subst-bv1* **by** *presburger*
next
case (*Abs T t*)
then show *?case*
using *loose-bvar-iff-exist-loose-bvar1 no-loose-bvar-imp-no-subst-bv1* **by** *presburger*
qed *auto*

lemma *bind-fv2-no-occs-unchanged*:
assumes ¬ *occs* (*case-prod Fv v*) *t*
shows (*bind-fv2 v lev t*) = *t*
using *assms* **by** (*induction t arbitrary*: *lev*) *auto*

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*
using *assms* **proof** (*induction t arbitrary*: *lev*)
case (*Bv x*)
then show *?case*
using *linorder-neqE-nat*
by (*auto split*: *prod.splits simp add*: *is-variable-imp-incr-boundvars-unchganged*)
next
case (*Abs T t*)
hence *bind-fv2 v* (*lev*+*1*) (*subst-bv1 t* (*lev*+*1*) (*case-prod Fv v*)) = *t*
by (*auto elim*: *Suc-lessE*)
then show *?case* **by** *simp*
next

case (*App t1 t2*)
then show *?case*
proof(*cases loose-bvar1 t1 lev*)
case *True*
hence *I1*: *bind-fv2 v lev* (*subst-bv1 t1 lev* (*case-prod Fv v*)) = *t1* **using** *App* **by** *auto*
then show *?thesis*
proof(*cases loose-bvar1 t2 lev*)
case *True*
hence *bind-fv2 v lev* (*subst-bv1 t2 lev* (*case-prod Fv v*)) = *t2* **using** *App* **by** *auto*
then show *?thesis* **using** *I1 App.prems is-variable.elims*(*2*) **by** *auto*
next

```
      case False
      hence bind-fv2 v lev (subst-bv1 t2 lev  (case-prod Fv v)) = t2
      proof−
      have subst-bv1 t2 lev  (case-prod Fv v) = t2 using subst-bv-no-loose-unchanged
        using App.prems(1−2) False assms le-neq-implies-less loose-bvar1.simps(2)
         by (metis loose-bvar-iff-exist-loose-bvar1 no-loose-bvar-imp-no-subst-bv1)
        moreover have bind-fv2 v lev t2 = t2
         using App.prems(2) bind-fv2-no-occs-unchanged
         using App.prems(2) bind-fv2-changed′ exists-subterm′.simps(1)
           exists-subterm-iff-exists-subterm′ by blast
        ultimately show ?thesis by simp
      qed
      then show ?thesis using I1 App.prems is-variable.elims(2) by auto
    qed
  next
    case False
    hence I1: bind-fv2 v lev (subst-bv1 t1 lev (case-prod Fv v)) = t1
    proof−
     have subst-bv1 t1 lev (case-prod Fv v) = t1 using subst-bv-no-loose-unchanged
       using App.prems(1−2) False le-neq-implies-less loose-bvar1.simps(2)
         by (metis loose-bvar-iff-exist-loose-bvar1 no-loose-bvar-imp-no-subst-bv1)
      moreover have bind-fv2 v lev t1 = t1
       using App.prems(2) bind-fv2-no-occs-unchanged by auto
      ultimately show ?thesis by simp
    qed
    then show ?thesis
    proof(cases loose-bvar1 t2 lev)
      case True
       hence bind-fv2 v lev (subst-bv1 t2 lev (case-prod Fv v)) = t2 using App by
auto
      then show ?thesis using I1 App.prems is-variable.elims(2) by auto
    next
      case False
      hence bind-fv2 v lev (subst-bv1 t2 lev (case-prod Fv v)) = t2
      proof−
      have subst-bv1 t2 lev (case-prod Fv v) = t2 using subst-bv-no-loose-unchanged
        using App.prems(1−2) False assms le-neq-implies-less loose-bvar1.simps(2)

         by (metis loose-bvar-iff-exist-loose-bvar1 no-loose-bvar-imp-no-subst-bv1)
        moreover have bind-fv2 v lev t2 = t2
         using App.prems(2) bind-fv2-no-occs-unchanged by auto
        ultimately show ?thesis by simp
      qed
      then show ?thesis using I1 App.prems is-variable.elims(2) by auto
    qed
  qed
qed auto

lemma bind-fv-subst-bv-cancel:
```

assumes $\bigwedge x\ .\ x > 0 \Longrightarrow \neg$ *loose-bvar1 t x*
assumes $\neg$ *occs* (*case-prod Fv v*) *t*
shows *bind-fv v* (*subst-bv* (*case-prod Fv v*) *t*) = *t*
using *bind-fv2-subst-bv1-cancel bind-fv-def assms subst-bv-def* **by** *auto*

lemma *not-loose-bvar-imp-not-loose-bvar1-all-greater*: $\neg$ *loose-bvar t lev* $\Longrightarrow$ *x>lev*
$\Longrightarrow \neg$ *loose-bvar1 t x*
by (*simp add*: *loose-bvar-iff-exist-loose-bvar1*)

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*) $\notin$ *fv t*
shows *mk-all name T* (*subst-bv* (*Fv name T*) (*strip-all-single-body t*)) = *t*
proof−
from *assms*(*1*) **obtain** *t′* **where** *t′*: (*Ct STR ''Pure.all''* ((*T* → *propT*) → *propT*)
$ *Abs T t′*) = *t*
by (*auto elim*!: *strip-all-single-var.elims*
simp add: *bind-eq-Some-conv typ-of-def split*: *if-splits option.splits if-splits*)

hence *s*: *strip-all-single-body t* = *t′* **by** *auto*

have $\bigwedge x.\ x > 0 \Longrightarrow \neg$ *loose-bvar1 t x*
using *assms*(*2*) *is-open-def loose-bvar-iff-exist-loose-bvar1* **by** *blast*

have *0* < *x* $\Longrightarrow \neg$ *loose-bvar1 t′ x* **for** *x*
using *assms*(*2*) **by** (*auto simp add*: *is-open-def t′*[*symmetric*] *loose-bvar-iff-exist-loose-bvar1*
gr0-conv-Suc)

have *occs t′ t* **by** (*simp add*: *t′*[*symmetric*])

have *bind-fv* (*name*, *T*) (*subst-bv* (*Fv name T*) (*strip-all-single-body t*)) =
(*strip-all-single-body t*)
using *assms*(*2*−*3*) *bind-fv-subst-bv-cancel gr0-conv-Suc*
by (*force simp add*: *s is-open-def t′*[*symmetric*]
loose-bvar-iff-exist-loose-bvar1 fv-iff-occs intro!: *bind-fv-subst-bv-cancel*)
then show *?thesis* **using** *assms* **by** (*auto simp add*: *s typ-of-def t′*)
qed

lemma *not-is-all-imp-strip-all-body-unchanged*: $\neg$ *is-all t* $\Longrightarrow$ *strip-all-body t* = *t*
by (*auto elim*!: *is-all.elims split*: *if-splits*)

lemma *no-loose-bvar-imp-no-subst-bvs*: *is-closed t* $\Longrightarrow$ *subst-bvs* [] *t* = *t*
using *no-loose-bvar-imp-no-subst-bvs1 subst-bvs-def is-open-def* **by** *simp*

lemma *is-closed* (*Abs T t*) $\Longrightarrow \neg$ *loose-bvar t 1* **unfolding** *is-open-def* **by** *simp*

lemma *bind-fv2-Fv-fv*[*simp*]: *fv* (*bind-fv2* (*x*, τ) *lev t*) = *fv t* − {(*x*,τ)}
by (*induction* (*x*, τ) *lev t rule*: *bind-fv2.induct*) (*auto split*: *if-splits term.splits*)

corollary *mk-all-fv-unchanged*: *fv* (*mk-all x* τ *B*) = *fv B* − {(*x*,τ)}
using *bind-fv2-Fv-fv bind-fv-def* **by** *auto*

lemma *mk-all-list-fv-unchanged*: *fv* (*mk-all-list l B*) = *fv B* − *set l*
proof (*induction l arbitrary*: *B rule*: *rev-induct*)
case *Nil*
then show *?case* **by** *simp*
next

case (*snoc x xs*)
have *s*: *mk-all-list* (*xs*@[*x*]) *B* = *case-prod mk-all x* (*mk-all-list xs B*)
by (*simp add*: *mk-all-list-def*)
show *?case*
by (*simp only*: *s snoc.IH mk-all-fv-unchanged split*: *prod.splits*) *auto*
qed

abbreviation *forall-intro-vars t Hs* ≡ *mk-all-list*
(*diff-list* (*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*
by (*auto simp add*: *sort-ex-def class-ex-def subset-eq*)

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

lemma *class-les-irrefl*: ¬ *class-les cs c c*
using *wf* **by** (*simp add*: *class-les-def*)
lemma *class-les-trans*: *class-les cs x y* ⟹ *class-les cs y z* ⟹ *class-les cs x z*

using *wf* **by** (*auto simp add*: *class-les-def class-leq-def trans-def*)

lemma *class-leq-refl*[*iff*]: *class-ex cs c* $\Longrightarrow$ *class-leq cs c c*
using *wf* **by** (*simp add*: *class-leq-def class-ex-def refl-on-def*)
lemma *class-leq-trans*: *class-leq cs x y* $\Longrightarrow$ *class-leq cs y z* $\Longrightarrow$ *class-leq cs x z*
using *wf* **by** (*auto simp add*: *class-leq-def elim*: *transE*)
lemma *class-leq-antisym*: *class-leq cs c1 c2* $\Longrightarrow$ *class-leq cs c2 c1* $\Longrightarrow$ *c1=c2*
using *wf* **by** (*auto intro*: *antisymD simp*: *trans-def class-leq-def*)

lemma *sort-leq-refl*[*iff*]: *sort-ex cs s* $\Longrightarrow$ *sort-leq cs s s*
using *class-leq-refl* **by** (*auto simp add*: *sort-ex-class-ex sort-leq-def*)
lemma *sort-leq-trans*: *sort-leq cs x y* $\Longrightarrow$ *sort-leq cs y z* $\Longrightarrow$ *sort-leq cs x z*
by (*meson class-leq-trans sort-leq-def*)
lemma *sort-leq-ex*: *sort-leq cs s1 s2* $\Longrightarrow$ *sort-ex cs s2*
by (*auto simp add*: *sort-ex-def class-leq-def sort-leq-def intro*: *FieldI2*)

lemma *sort-leq-minimize*:
sort-leq cs s1 s2 $\Longrightarrow$ $\exists$ *s1′*. ($\forall$ *c1* $\in$ *s1′* . $\exists$ *c2* $\in$ *s2*. *class-leq cs c1 c2*) $\wedge$ *sort-leq cs s1′ s2*
by (*meson class-leq-refl sort-ex-class-ex sort-leq-ex sort-leq-refl*)

lemma *sort-ex cs s2* $\Longrightarrow$ *s1* $\subseteq$ *s2* $\Longrightarrow$ *sort-ex cs s1*
by (*meson sort-ex-def subset-trans*)

lemma *superset-imp-sort-leq*: *sort-ex cs s2* $\Longrightarrow$ *s1* $\supseteq$ *s2* $\Longrightarrow$ *sort-leq cs s1 s2*
by (*auto simp add*: *sort-ex-class-ex sort-leq-def sort-ex-def*)
lemma *full-sort-top*: *sort-ex cs s* $\Longrightarrow$ *sort-leq cs s full-sort*
by (*simp add*: *sort-leq-def*)

lemma *sort-les-trans*: *sort-les cs x y* $\Longrightarrow$ *sort-les cs y z* $\Longrightarrow$ *sort-les cs x z*
using *sort-les-def sort-leq-trans* **by** *blast*

lemma *sort-eqvI*: *sort-leq cs s1 s2* $\Longrightarrow$ *sort-leq cs s2 s1* $\Longrightarrow$ *sort-eqv cs s1 s2*
by (*simp add*: *sort-eqv-def*)
lemma *sort-eqv-refl*: *sort-ex cs s* $\Longrightarrow$ *sort-eqv cs s s*
using *sort-leq-refl* **by** (*auto simp add*: *sort-eqv-def*)
lemma *sort-eqv-trans*: *sort-eqv cs x y* $\Longrightarrow$ *sort-eqv cs y z* $\Longrightarrow$ *sort-eqv cs x z*
using *sort-eqv-def sort-leq-trans* **by** *blast*
lemma *sort-eqv-sym*: *sort-eqv cs x y* $\Longrightarrow$ *sort-eqv cs y x*
by (*auto simp add*: *sort-eqv-def*)

lemma *normalize-sort-empty*[*simp*]: *normalize-sort cs full-sort* = *full-sort*
by (*simp add*: *normalize-sort-def*)
lemma *normalize-sort-normalize-sort*[*simp*]:
normalize-sort cs (*normalize-sort cs s*) = *normalize-sort cs s*

by (*auto simp add*: *normalize-sort-def*)

lemma *sort-ex-norm-sort*: *sort-ex cs s* $\Longrightarrow$ *sort-ex cs* (*normalize-sort cs s*)
by (*simp add*: *normalize-sort-def sort-ex-class-ex*)

lemma *normalized-sort-subset*: *normalize-sort cs s* $\subseteq$ *s*
by (*auto simp add*: *normalize-sort-def*)

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−
have *class-ex cs c* **using** *assms*(*1*) **by** (*auto simp add*: *sort-ex-class-ex*)
from *this assms*(*2*) **obtain** *c'* **where** *class-les cs c' c c'* $\in$ *s*
using *class-les-irrefl* **by** (*auto simp add*: *normalize-sort-def*)
thus *?thesis*
using ‹*class-ex cs c*› *class-les-irrefl class-les-trans* **by** (*simp add*: *normalize-sort-def*) *blast*
qed

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*
using *assms normalize-sort-removed-elem-irrelevant'*
by (*simp add*: *normalize-sort-def*)

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*
using *assms* **proof** (*induction s rule*: *finite-induct*)
case *empty*
then show *?case* **by** *simp*
next
case (*insert c s*)
note *ICons* = *this*
then show *?case*
proof(*cases s*)
case *emptyI*
hence *normalize-sort cs* (*insert c s*) = {*c*}
using *insert class-les-irrefl* **by** (*auto simp add*: *normalize-sort-def sort-ex-class-ex*)
then show *?thesis* **by** *simp*
next
case (*insertI c' s'*)
hence *normalize-sort cs s* $\neq$ *full-sort*
using *ICons* **by** (*auto simp add*: *normalize-sort-def sort-ex-class-ex*)

```
    then show ?thesis
    proof (cases c ∈ (normalize-sort cs s))
      case True
      hence insert c s = s
        using normalized-sort-subset by fastforce
      then show ?thesis
     using ICons by (auto simp add: normalize-sort-def sort-ex-class-ex class-les-def)
    next
      case False
      then show ?thesis
        using normalize-sort-removed-elem-irrelevant
        using insert.prems(2) ICons(3) ‹normalize-sort cs s ≠ full-sort› by auto
    qed
  qed
qed

lemma choose-smaller-in-sort:
  assumes elem: c ∈ s and nelem: c ∉ (normalize-sort cs s) and sort-ex cs s
  obtains c' where c' ∈ s and class-les cs c' c
  using assms by (auto simp add: normalize-sort-def sort-ex-class-ex)

lemma normalize-ex-bound':
  assumes finite: finite s and elem: c ∈ s and nelem: c ∉ (normalize-sort cs s)
    and sort-ex cs s
  shows ∃ c' ∈ (normalize-sort cs s) . class-les cs c' c
using assms proof (induction s arbitrary: c)
  case empty
  then show ?case by simp
next
  case (insert ic s)
  then show ?case
  proof(cases ic=c)
    case True
    then show ?thesis
        by (smt choose-smaller-in-sort class-les-irrefl class-les-trans insert.IH in-
sert.prems(2)
      insert.prems(3) insert-iff insert-subset normalize-sort-removed-elem-irrelevant'
sort-ex-def)
  next
    case False
    hence c ∈ s using insert.prems by simp
    then show ?thesis
    proof(cases ic ∈ (normalize-sort cs (insert ic s)))
      case True
      then show ?thesis
      proof(cases class-les cs ic c)
        case True
        then show ?thesis
          using insert ‹c ∈ s› normalize-sort-removed-elem-irrelevant' sort-ex-def
```

```
        by (metis insert-subset)
      next
        case False

        obtain c'' where c'': c'' ∈ (normalize-sort cs s) class-les cs c'' c
          using insert ‹c ∈ s› normalize-sort-removed-elem-irrelevant' sort-ex-def
        by (metis False choose-smaller-in-sort class-les-trans insert-iff insert-subset)
        moreover have (c'', c) ∈ cs (c, c'') ∉ cs
          using c'' by (simp-all add: class-leq-def class-les-def)
        moreover hence ¬ class-les cs ic c''
          by (meson False class-leq-def class-les-def class-les-trans)

        ultimately show ?thesis
              by (auto simp add: normalize-sort-def sort-ex-class-ex class-ex-def
class-leq-def class-les-def)
      qed
    next
      case False
      then show ?thesis
        by (metis (full-types) insert.IH insert.prems(2) insert.prems(3) ‹c ∈ s›
            normalize-sort-removed-elem-irrelevant sort-ex-def insert-subset)
    qed
  qed
qed

corollary normalize-ex-bound:
  assumes finite: finite s and elem: c ∈ s and nelem: c ∉ (normalize-sort cs s)
    and sort-ex cs s
  obtains c' where c' ∈ (normalize-sort cs s) and class-les cs c' c
  using assms normalize-ex-bound' by auto

lemma sort-ex cs s ⟹ sort-leq cs s (normalize-sort cs s)
  by (auto simp add: normalize-sort-def sort-leq-def sort-ex-class-ex)
lemma sort-eqv-normalize-sort:
  assumes finite s
  assumes sort-ex cs s
  shows sort-eqv cs s (normalize-sort cs s)
proof (intro sort-eqvI)
  show sort-leq cs s (normalize-sort cs s)
   using assms(2) by (auto simp add: normalize-sort-def sort-leq-def sort-ex-class-ex)
next
  show sort-leq cs (normalize-sort cs s) s
  proof (unfold sort-leq-def; intro ballI)
    fix c2 assume c2 ∈ s
    show ∃ c1 ∈ normalize-sort cs s. class-leq cs c1 c2
    proof (cases c2 ∈ normalize-sort cs s)
      case True
      then show ?thesis using ‹c2 ∈ s› assms sort-ex-class-ex by fast
    next
```

```
    case False
    from this obtain c' where c' ∈ normalize-sort cs s and class-les cs c' c2
      using ‹c2 ∈ s› normalize-ex-bound assms by metis
    then show ?thesis using class-les-def by metis
  qed
 qed
qed

lemma normalize-sort-eq-imp-sort-eqv: sort-ex cs s1 ⟹ sort-ex cs s2 ⟹ finite
s1 ⟹ finite s2
 ⟹ normalize-sort cs s1 = normalize-sort cs s2
 ⟹ sort-eqv cs s1 s2
 by (metis sort-eqv-sym sort-eqv-trans wf-subclass-loc.sort-eqv-normalize-sort wf-subclass-loc-axioms)

lemma class-leq cs c1 c2 ⟷ class-les cs c1 c2 ∨ (c1=c2 ∧ class-ex cs c1)
 by (meson FieldI1 class-ex-def class-leq-antisym class-leq-def class-leq-refl class-les-def)

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 (rule ccontr)
 have sort-leq cs s1 s2 sort-leq cs s2 s1
  using assms(3) by (auto simp add: sort-eqv-def)

 assume normalize-sort cs s1 ≠ normalize-sort cs s2
 hence ¬ normalize-sort cs s1 ⊆ normalize-sort cs s2 ∨
  ¬ normalize-sort cs s2 ⊆ normalize-sort cs s1
  by simp
 from this consider ¬ normalize-sort cs s1 ⊆ normalize-sort cs s2
  | normalize-sort cs s1 ⊆ normalize-sort cs s2
    ¬ normalize-sort cs s2 ⊆ normalize-sort cs s1
  by blast
 thus False
 proof cases
  case 1
  from this obtain c where c: c ∈ normalize-sort cs s1 c ∉ normalize-sort cs s2
    by blast
  from this obtain c' where c': c' ∈ normalize-sort cs s2 class-les cs c' c
    by (smt ‹sort-leq cs s1 s2› ‹sort-leq cs s2 s1› class-les-def mem-Collect-eq
normalize-sort-def
      sort-leq-def wf-subclass-loc.class-leq-antisym wf-subclass-loc.class-leq-trans
wf-subclass-loc-axioms)
  then show ?thesis
  proof(cases c' ∈ normalize-sort cs s1)
    case True
    hence c ∉ normalize-sort cs s1
      using c c' by (auto simp add: normalize-sort-def)
    then show ?thesis using c(1) by simp
  next
```

```
      case False
      from False c' obtain c'' where c'': c'' ∈ normalize-sort cs s1 class-les cs c''
c'
       by (smt ‹sort-leq cs s1 s2› ‹sort-leq cs s2 s1› class-les-def mem-Collect-eq
normalize-sort-def
           sort-leq-def wf-subclass-loc.class-leq-antisym wf-subclass-loc.class-leq-trans
wf-subclass-loc-axioms)
      hence class-les cs c'' c
        using c'(2) class-les-trans by blast
      hence c ∉ normalize-sort cs s1
        using c c'' by (auto simp add: normalize-sort-def)
      then show ?thesis using c(1) by simp
    qed
  next

    case 2
    from this obtain c where c: c ∈ normalize-sort cs s2 c ∉ normalize-sort cs s1
      by blast
    from this obtain c' where c': c' ∈ normalize-sort cs s1 class-les cs c' c
       by (smt ‹sort-leq cs s1 s2› ‹sort-leq cs s2 s1› class-les-def mem-Collect-eq
normalize-sort-def
           sort-leq-def wf-subclass-loc.class-leq-antisym wf-subclass-loc.class-leq-trans
wf-subclass-loc-axioms)
    then show ?thesis
    proof(cases c' ∈ normalize-sort cs s2)
      case True
      hence c ∉ normalize-sort cs s2
        using c c' by (auto simp add: normalize-sort-def)
      then show ?thesis using c(1) by simp
    next
      case False
     from False c' obtain c'' where c'':c''∈ normalize-sort cs s2 class-les cs c'' c'
       by (smt ‹sort-leq cs s1 s2› ‹sort-leq cs s2 s1› class-les-def mem-Collect-eq
normalize-sort-def
           sort-leq-def wf-subclass-loc.class-leq-antisym wf-subclass-loc.class-leq-trans
wf-subclass-loc-axioms)
      hence class-les cs c'' c
        using c'(2) class-les-trans by blast
      hence c ∉ normalize-sort cs s2
        using c c'' by (auto simp add: normalize-sort-def)
      then show ?thesis using c(1) by simp
    qed
  qed
qed

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 ⟷ normalize-sort cs s1 = normalize-sort cs s2
```

using *assms normalize-sort-eq-imp-sort-eqv sort-eqv-imp-normalize-sort-eq* **by** *blast*

end

lemma *tcsigs-sorts-defined*: *wf-osig oss* $\Longrightarrow$
(∀ *ars* ∈ *ran* (*tcsigs oss*) . ∀ *ss* ∈ *ran ars* . ∀ *s* ∈ *set ss*. *sort-ex* (*subclass oss*) *s*)
by (*cases oss*) (*simp add*: *wf-sort-def all-normalized-and-ex-tcsigs-def*)

lemma *osig-subclass-loc*: *wf-osig oss* $\Longrightarrow$ *wf-subclass-loc* (*subclass oss*)
using *wf-subclass-loc.intro* **by** (*cases oss*) *simp*

lemma *wf-osig-imp-wf-subclass-loc*: *wf-osig oss* $\Longrightarrow$ *wf-subclass-loc* (*subclass oss*)
by (*cases oss*) (*simp add*: *wf-subclass-loc-def*)

lemma *has-sort-Tv-imp-sort-leq*: *has-sort oss* (*Tv idn S*) *S'* $\Longrightarrow$ *sort-leq* (*subclass oss*) *S S'*
by (*auto simp add*: *has-sort.simps*)

end

Constants for encoding class/sort constraints in term language

theory *SortConstants*
imports *Sorts*
begin

fun *dest-type* :: *term* ⇒ *typ option* **where**
dest-type (*Ct nc* (*Ty nt* [*ty*])) =
(*if nc* = *STR ''Pure.type''* ∧ *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* ⇒ *name* ⇒ *name option*

abbreviation *class-of-const c* ≡ (*unsuffix classN c*)

fun *dest-of-class* :: *term* ⇒ (*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*
by (*induction T*) (*auto split*: *option.splits intro*: *wf-type.intros simp add*: *list-all-iff*)
lemma *wf-type-imp-typ-ok-sig*: *wf-type* Σ *T* ⟹ *typ-ok-sig* Σ *T*
by (*induction* Σ *T rule*: *wf-type.induct*) (*simp-all split*: *option.splits add*: *list-all-iff*)

corollary *wf-type-iff-typ-ok-sig*[*iff*]: *wf-type* Σ *T* = *typ-ok-sig* Σ *T*
using *wf-type-imp-typ-ok-sig typ-ok-sig-imp-wf-type* **by** *blast*

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*
by (*induction t*) (*auto intro*: *wf-term.intros split*: *option.splits*)
lemma *wf-term-imp-term-ok'*: *wf-term* Σ *t* ⟹ *term-ok'* Σ *t*
by (*induction* Σ *t rule*: *wf-term.induct*) (*auto split*: *option.splits*)
corollary *wf-term-iff-term-ok'*[*iff*]: *wf-term* Σ *t* = *term-ok'* Σ *t*
using *term-ok'-imp-wf-term wf-term-imp-term-ok'* **by** *blast*

lemma *acyclic-empty*[*simp*]: *acyclic* {} **unfolding** *acyclic-def* **by** *simp*

lemma *wf-sig* (*Map.empty*, *Map.empty*, *empty-osig*)
by (*simp add*: *coregular-tcsigs-def complete-tcsigs-def consistent-length-tcsigs-def*

all-normalized-and-ex-tcsigs-def)
lemma
term-ok-imp-typ-ok-pre:
is-std-sig Σ ⟹ *wf-term* Σ *t* ⟹ *list-all* (*typ-ok-sig* Σ) *Ts*
⟹ *typ-of1 Ts t* = *Some ty* ⟹ *typ-ok-sig* Σ *ty*

proof (*induction Ts t arbitrary*: *ty rule*: *typ-of1.induct*)
case (*2 Ts i*)
then show *?case* **by** (*auto simp add*: *bind-eq-Some-conv list-all-length split*: *option.splits if-splits*)
next
case (*4 Ts T body*)
obtain *bodyT* **where** *bodyT*: *typ-of1* (*T*#*Ts*) *body* = *Some bodyT*
using *4.prems* **by** *fastforce*
hence *ty*: *ty* = *T* → *bodyT*
using *4* **by** *simp*
have *typ-ok-sig* Σ *bodyT*
using *4 bodyT* **by** *simp*
thus *?case*
using *ty 4* **by** (*cases* Σ) *auto*
next
case (*5 Ts f u T*)
from *this* **obtain** *U* **where** *typ-of1 Ts u* = *Some U*
using *typ-of1-split-App* **by** *blast*
moreover hence *typ-of1 Ts f* = *Some* (*U* → *T*)
using *5.prems*(*4*) **by** (*meson typ-of1-arg-typ*)
ultimately have *typ-ok-sig* Σ (*U* → *T*)
using *5.IH*(*2*) *5.prems*(*1*) *5.prems*(*2*) *5.prems*(*3*) *term-ok'.simps*(*4*) **by** *blast*
then show *?case*
by (*auto simp add*: *bind-eq-Some-conv split*: *option.splits if-splits*)
qed (*auto simp add*: *bind-eq-Some-conv split*: *option.splits if-splits*)

lemma *theory-full-exhaust*: (⋀*cto tao sorts axioms*.
Θ = ((*cto*, *tao*, *sorts*), *axioms*) ⟹ *P*)
⟹ *P*
apply (*cases* Θ) **subgoal for** Σ *axioms* **apply** (*cases* Σ) **by** *auto* **done**

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* ⟹ *subst-bv u t* = *t*
using *closed-subst-bv-no-change typ-of-imp-closed* **by** *auto*
corollary *term-ok-subst-bv-no-change*: *term-ok* Θ *t* ⟹ *subst-bv u t* = *t*
using *typ-of-subst-bv-no-change wt-term-def* **by** *auto*

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*
by (*auto simp add*: *typ-of-def eq-axs-def mk-all-list-def add-vars'-def bind-eq-Some-conv bind-fv-def*)

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*)
using *assms*
by (*all* ‹*cases* Θ *rule*: *theory-full-exhaust*›)
(*auto simp add*: *wt-term-def typ-of-def tinstT-def eq-axs-def bind-eq-Some-conv bind-fv-def sort-ex-def normalize-sort-def mk-all-list-def add-vars'-def wf-sort-def*)

lemma *wf-theory-imp-is-std-sig*: *wf-theory* Θ ⟹ *is-std-sig* (*sig* Θ)
by (*cases* Θ *rule*: *theory-full-exhaust*) *simp*
lemma *wf-theory-imp-wf-sig*: *wf-theory* Θ ⟹ *wf-sig* (*sig* Θ)
by (*cases* Θ *rule*: *theory-full-exhaust*) *simp*

lemma
term-ok-imp-typ-ok:
wf-theory thy ⟹ *term-ok thy t* ⟹ *typ-of t* = *Some ty* ⟹ *typ-ok thy ty*
apply (*cases thy*)
using *term-ok-imp-typ-ok-pre term-ok-def*
by (*metis list.pred-inject*(*1*) *wt-term-def wf-theory-imp-is-std-sig typ-of-def typ-ok-def wf-type-iff-typ-ok-sig*)

lemma *axioms-terms-ok*: *wf-theory thy* ⟹ *A*∈*axioms thy* ⟹ *term-ok thy A*
using *wt-term-def* **by** (*cases thy rule*: *theory-full-exhaust*) *simp*

lemma *axioms-typ-of-propT*: *wf-theory thy* ⟹ *A*∈*axioms thy* ⟹ *typ-of A* = *Some propT*
using *has-typ-iff-typ-of* **by** (*cases thy rule*: *theory-full-exhaust*) *simp*

lemma *propT-ok*[*simp*]: *wf-theory* Θ ⟹ *typ-ok* Θ *propT*
using *term-ok-imp-typ-ok wf-theory.elims*(*2*)
by (*metis sig.simps term-ok-eq-ax*(*4*) *typ-of-eq-ax*(*4*))

lemma *term-ok-mk-eqD*: *term-ok* Θ (*mk-eq s t*) ⟹ *term-ok* Θ *s* ∧ *term-ok* Θ *t*
using *term-ok'.simps*(*4*) *wt-term-def typ-of-def* **by** (*auto simp add*: *bind-eq-Some-conv*)
lemma *term-ok-app-eqD*: *term-ok* Θ (*s* $ *t*) ⟹ *term-ok* Θ *s* ∧ *term-ok* Θ *t*
using *term-ok'.simps*(*4*) *wt-term-def typ-of-def* **by** (*auto simp add*: *bind-eq-Some-conv*)

lemma *wf-type-Type-imp-mgd*:
wf-sig Σ ⟹ *wf-type* Σ (*Ty n Ts*) ⟹ *tcsigs* (*osig* Σ) *n* ≠ *None*
by (*cases* Σ) (*auto split*: *option.splits*)

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*))
using *assms typ-of-eta-expand* **by** (*auto simp add*: *wt-term-def*)

lemma *term-ok'-incr-bv*: *term-ok'* Σ *t* ⟹ *term-ok'* Σ (*incr-bv inc lev t*)
by (*induction inc lev t rule*: *incr-bv.induct*) *auto*

lemma *term-ok'-subst-bv2*: *term-ok'* Σ *s* ⟹ *term-ok'* Σ *u* ⟹ *term-ok'* Σ (*subst-bv2 s lev u*)
by (*induction s lev u rule*: *subst-bv2.induct*) (*auto simp add*: *term-ok'-incr-bv*)

lemma *term-ok'-subst-bv*: *term-ok'* Σ (*Abs T t*) ⟹ *term-ok'* Σ (*subst-bv* (*Fv x T*) *t*)
by (*simp add*: *substn-subst-0'* *term-ok'-subst-bv2*)
lemma *term-ok-subst-bv*: *term-ok* Θ (*Abs T t*) ⟹ *term-ok* Θ (*subst-bv* (*Fv x T*) *t*)
apply (*simp add*: *term-ok'-subst-bv wt-term-def*)
using *subst-bv-def typ-of1-subst-bv-gen'* *typ-of-Abs-body-typ'* *typ-of-def* **by** *fastforce*

lemma *term-ok-subst-bv2-0*: *term-ok* Θ (*Abs T t*) ⟹ *term-ok* Θ (*subst-bv2 t 0* (*Fv x T*))
apply (*clarsimp simp add*: *term-ok'-subst-bv2 wt-term-def*)
using *substn-subst-0'* *typ-of1-subst-bv-gen'* *typ-of-Abs-body-typ'* *typ-of-def*
wt-term-def term-ok-subst-bv **by** *auto*

lemma *has-sort-empty*[*simp*]:
assumes *wf-sig* Σ *wf-type* Σ *T*
shows *has-sort* (*osig* Σ) *T full-sort*
proof(*cases T*)
case (*Ty n Ts*)
obtain *cl tcs* **where** *cltcs*: *osig* Σ = (*cl*, *tcs*)
by *fastforce*
obtain *mgd* **where** *mgd*: *tcsigs* (*osig* Σ) *n* = *Some mgd*
using *wf-type-Type-imp-mgd assms Ty* **by** *blast*
show *?thesis*
using *mgd cltcs* **by** (*auto simp add*: *Ty intro*!: *has-sort-Ty*)
next
case (*Tv v S*)

then show *?thesis*
by (*cases osig* Σ) (*auto simp add*: *sort-leq-def split*: *prod.splits*)
qed

lemma *typ-Fv-of-full-sort*[*simp*]:
wf-theory Θ ⟹ *term-ok* Θ (*Fv v T*) ⟹ *has-sort* (*osig* (*sig* Θ)) *T full-sort*
by (*simp add*: *wt-term-def wf-theory-imp-wf-sig*)

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*
by (*induction T*) (*auto simp add*: *map-idI*)

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*
using *assms*
proof(*induction T*)
case (*Ty n Ts*)
then show *?case* **by** (*induction Ts*) *auto*
next
case (*Tv idn S*)
then show *?case* **using** *lookup-eq-order-irrelevant* **by** (*metis subst-typ.simps*(*2*))
qed

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(*induction T arbitrary*: *l*)
case (*Ty n Ts*)
then show *?case* **by** (*induction Ts*) *auto*
next
case (*Tv idn S*)
hence *d*: *distinct* (*map fst* (*map* (λ(*x*,*y*).((*x*,*y*), ϱ *x y*)) *l*))
by (*simp add*: *case-prod-beta map-idI*)
hence *el*: ((*idn*,*S*), ϱ *idn S*) ∈ *set* (*map* (λ*a*. *case a of* (*x*, *y*) ⇒ ((*x*, *y*), ϱ *x y*))

l)
  using *Tv* **by** *auto*
 show *?case* **using** *iffD1*[*OF lookup-present-eq-key*, *OF - el*] *Tv.prems d* **by** *auto*
qed

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(*rule someI2-ex*)
 show ∃ *a*. *distinct a* ∧ *tvsT T* ⊆ *set a*
  using *finite-tvsT finite-distinct-list*
  by (*metis order-refl*)
next
 fix *l* **assume** *l*: *distinct l* ∧ *tvsT T* ⊆ *set l*
 then show *tsubstT T ϱ* = *subst-typ* (*map* (λ*a*. *case a of* (*x*, *y*) ⇒ ((*x*, *y*), *ϱ x y*)) *l*) *T*
  using *subst-typ-simulates-tsubstT-gen′* **by** *blast*
qed

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
 apply (*rule someI2-ex*)
 using *finite-tvsT finite-distinct-list* **apply** *metis*
 using *subst-typ-simulates-tsubstT-gen′* **apply** *simp*
 done

lemma *tsubstT-simulates-subst-typ*: *subst-typ insts T*
 = *tsubstT T* (λ*idn S* . *the-default* (*Tv idn S*) (*lookup* (λ*x*. *x*=(*idn*, *S*)) *insts*))
 by (*induction T*) *auto*

lemma *subst-typ-comp*:
  subst-typ inst1 (*subst-typ inst2 T*) = *subst-typ* (*map* (*apsnd* (*subst-typ inst1*)) *inst2* @ *inst1*) *T*
proof (*induction inst2 T arbitrary*: *inst1 rule*: *subst-typ.induct*)
 case (*1 insts a Ts*)
 then show *?case*
  by *auto*
next
 case (*2 insts idn S*)
 then show *?case*
  by (*induction insts*) *auto*
qed

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

```
  case (Ty n Ts)
  then show ?case
    by auto
next
  case (Tv n S)
  then show ?case
  proof(induction insts)
    case Nil
    then show ?case
      by auto
  next
    case (Cons inst insts)
    then show ?case
      by simp (metis clearjunk.simps(2) lookup-AList-clearjunk)
  qed
qed

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
  by (induction t) (simp-all add:)

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
  using assms
proof(induction t)
  case (Fv idn T)
  then show ?case
    apply (simp add: Let-def subst-typ-irrelevant-order[OF Fv.prems(1–3)])
   using lookup-eq-order-irrelevant by (metis Fv.prems(4) Fv.prems(5) insts-assms)
next
  case (Abs T t)
  then show ?case using subst-typ-irrelevant-order[OF instT-assms] by simp
qed (simp-all add: subst-typ-irrelevant-order[OF instT-assms])
```

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−
let *?lty* = *map* (λ(*x*,*y*).((*x*,*y*), *ϱty x y*)) *lty*

have *p1ty*: *distinct* (*map fst ?lty*) **using** *lty-assms*
by (*simp add*: *case-prod-beta map-idI*)

let *?lt* = *map* (λ(*x*,*y*).((*x*,*y*), *ϱt x y*)) *lt*

have *p1t*: *distinct* (*map fst ?lt*) **using** *lt-assms*
by (*simp add*: *case-prod-beta map-idI*)

show *?thesis* **using** *assms*
proof(*induction t arbitrary*: *lty lt*)
case (*Fv idn T*)

let *?T* = *tsubstT T ϱty*
have *el*: ((*idn*, *?T*), *ϱt idn ?T*) ∈ *set* (*map* (λ(*x*,*y*).((*x*,*y*), *ϱt x y*)) *lt*)
using *Fv* **by** *auto*
have *d*: *distinct* (*map fst* (*map* (λ(*x*,*y*).((*x*,*y*), *ϱt x y*)) *lt*))
using *Fv* **by** (*simp add*: *case-prod-beta map-idI*)
show *?case* **using** *Fv.prems d*
by (*auto simp add*: *iffD1*[*OF lookup-present-eq-key*, *OF d el*]
subst-typ-simulates-tsubstT-gen′[*symmetric*] *Let-def*)
qed (*simp-all add*: *subst-typ-simulates-tsubstT-gen′*)
qed

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*), *ϱt x y*)) (*SOME lt* . *distinct lt* ∧ *fv* (*tsubst t ϱty*) = *set lt*)) *t*
apply (*rule someI2-ex*)
using *finite-fv finite-distinct-list* **apply** *metis*
apply (*rule someI2-ex*)
using *finite-tvs finite-distinct-list* **apply** *metis*
using *subst-type-term-simulates-subst-tsubst-gen′* **by** *simp*

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

lemma *subst-typ′-nil*[*simp*]: *subst-typ′* [] *A* = *A*
by (*induction A*) (*auto simp add*:)

lemma *subst-typ′-simulates-tsubst-gen′*: *distinct pairs* ⟹ *tvs t* ⊆ *set pairs*

$\Longrightarrow$ *tsubst t ϱ = subst-typ′ (map (λ(x,y).((x,y), ϱ x y)) pairs) t*
by (*induction t arbitrary*: *pairs ϱ*)
(*auto simp add*: *subst-typ-simulates-tsubstT-gen′*)

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(*rule someI2-ex*)
show ∃ *a. distinct a ∧ tvs t ⊆ set a*
using *finite-tvs finite-distinct-list*
by (*metis order-refl*)
next
fix *l* **assume** *l*: *distinct l ∧ tvs t ⊆ set l*

then show *tsubst t ϱ = subst-typ′ (map (λa. case a of (x, y) ⇒ ((x, y), ϱ x y)) l) t*
using *subst-typ′-simulates-tsubst-gen′* **by** *blast*
qed

lemma *tsubst-simulates-subst-typ′*: *subst-typ′ insts T*
= tsubst T (λidn S . the-default (Tv idn S) (lookup (λx. x=(idn, S)) insts))
by (*induction T*) (*auto simp add*: *tsubstT-simulates-subst-typ*)

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
by (*induction T*) (*auto simp add*: *lookup-eq-key-not-present*)

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
by (*induction t*) (*auto simp add*: *subst-type-add-degenerate-instance*)

lemma *subst-typ′-comp*:
subst-typ′ inst1 (subst-typ′ inst2 t) = subst-typ′ (map (apsnd (subst-typ inst1)) inst2 @ inst1) t
by (*induction t*) (*use subst-typ-comp* **in** *auto*)

lemma *subst-typ′-AList-clearjunk*: *subst-typ′ insts t = subst-typ′ (AList.clearjunk insts) t*
by (*induction t*) (*use subst-typ-AList-clearjunk* **in** *auto*)

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 (λx. 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*
by (*induction t*) *auto*

lemma *subst-type-term-without-type-insts-eq-subst-term*[*simp*]:
subst-type-term [] *insts t* = *subst-term insts t*
by (*induction insts t rule*: *subst-term.induct*) *simp-all*

lemma *subst-type-term-split-levels*:
subst-type-term instT insts t = *subst-term insts* (*subst-typ' instT t*)
by (*induction t*) (*auto simp add*: *Let-def*)

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*
using *assms* **proof** (*induction instT T rule*: *subst-typ.induct*)
case (*1 inst a Ts*)
then show *?case*
proof (*induction Ts arbitrary*: *inst*)
case *Nil*
then show *?case* **by** (*induction inst*) *auto*
next
case (*Cons T Ts*)
hence *subst-typ inst* (*Ty a Ts*) = *fold* (λ*single*. *subst-typ* [*single*]) *inst* (*Ty a Ts*)
by *simp*
moreover have *subst-typ inst T* = *fold* (λ*single*. *subst-typ* [*single*]) *inst T*
using *Cons 1* **by** *simp*
moreover have *fold* (λ*single*. *subst-typ* [*single*]) *inst* (*Ty a* (*T*#*Ts*))
= (*Ty a* (*map* (*fold* (λ*single*. *subst-typ* [*single*]) *inst*) (*T*#*Ts*)))
proof (*induction inst rule*: *rev-induct*)
case *Nil*
then show *?case* **by** *simp*
next
case (*snoc x xs*)
hence *fold* (λ*single*. *subst-typ* [*single*]) (*xs* @ [*x*]) (*Ty a* (*T* # *Ts*)) =
Ty a (*map* (*subst-typ* [*x*]) (*map* (*fold* (λ*single*. *subst-typ* [*single*]) *xs*) (*T* # *Ts*)))
by *simp*
then show *?case* **by** *simp*
qed
ultimately show *?case*
using *Cons.prems*(*1*) *Cons.prems*(*2*) *local.Cons*(*4*) **by** *auto*

```
  qed
next
  case (2 inst idn S)
  then show ?case
  proof (cases lookup (λx . x = (idn, S)) (inst))
    case None
     hence fst p ≠ (idn, S) if p∈set inst for p using that by (auto simp add:
lookup-None-iff)
    hence subst-typ [p] (Tv idn S) = Tv idn S if p∈set inst for p
      using that by (cases p) fastforce
    from this None show ?thesis by (induction inst) (auto split: if-splits)
  next
    case (Some a)

     have elem: ((idn, S), a) ∈ set inst using Some lookup-present-eq-key'' 2 by
fastforce
    from this obtain fs bs where split: inst = fs @ ((idn, S), a) # bs
      by (meson split-list)
    hence (idn, S) ∉ set (map fst fs) and (idn, S) ∉ set (map fst bs) using 2 by
simp-all

    hence fst p ≠ (idn, S) if p∈set fs for p
      using that by force
    hence id-subst-fs: subst-typ [p] (Tv idn S) = Tv idn S if p∈set fs for p
      using that by (cases p) fastforce
    hence fs-step: fold (λsingle. subst-typ [single]) fs (Tv idn S) = Tv idn S
      by (induction fs) (auto split: if-splits)

    have change-step: subst-typ [((idn, S), a)] (Tv idn S) = a by simp

    have bs-sub: set bs ⊆ set inst using split by auto
    hence x ∉ fst ‘ set bs
      if x∈ ⋃ (tvsT ‘ snd ‘ set bs) for x
      using 2 that split by (auto simp add: image-iff)

    have v ∉ fst ‘ set bs if v ∈ tvsT a for v
      using that 2 elem bs-sub by (fastforce simp add: image-iff)

    hence id-subst-bs: subst-typ [p] a = a if p ∈ set bs for p
    using that proof(cases p, induction a)
      case (Ty n Ts)
      then show ?case
        by (induction Ts) auto
    next
      case (Tv n S)
      then show ?case
        by force
    qed
    hence bs-step: fold (λsingle. subst-typ [single]) bs a = a
```

by (*induction bs*) *auto*

from *fs-step change-step bs-step split Some* **show** *?thesis* **by** *simp*
qed
qed

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−
have *subst-typ* (*x*#*xs*) *T* = *fold* (λ*single* . *subst-typ* [*single*]) (*x*#*xs*) *T*
using *assms subst-typ-stepwise* **by** *blast*
also have ... = *fold* (λ*single* . *subst-typ* [*single*]) *xs* (*subst-typ* [*x*] *T*)
by *simp*
also have ... = *subst-typ xs* (*subst-typ* [*x*] *T*)
using *assms subst-typ-stepwise* **by** *simp*
finally show *?thesis* **.**
qed

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−
have *subst-typ* (*xs* @ [*x*]) *T* = *fold* (λ*single* . *subst-typ* [*single*]) (*xs*@[*x*]) *T*
using *assms subst-typ-stepwise* **by** *blast*
also have ... = *subst-typ* [*x*] (*fold* (λ*single* . *subst-typ* [*single*]) *xs T*)
by *simp*
also have ... = *subst-typ* [*x*] (*subst-typ xs T*)
using *assms subst-typ-stepwise* **by** *simp*
finally show *?thesis* **.**
qed

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*

using *assms* **proof** (*induction instT arbitrary*: *t rule*: *rev-induct*)
case *Nil*
then show *?case* **by** *simp*
next
case (*snoc x xs*)
then show *?case*
apply (*induction t*)
using *subst-typ-split-last* **apply** *simp-all*
apply (*metis map-types.simps*)+

```
  done
qed


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 (λsingle acc . subst-term [single] acc) insts t
using assms proof (induction insts arbitrary: t rule: rev-induct)
  case Nil
  then show ?case by simp
next
  case (snoc x xs)
  then show ?case
  proof (induction t)
    case (Fv idn T)

    define insts where insts-def: insts = xs @ [x]
    have insts-thm1: distinct (map fst insts) using insts-def snoc by simp
    have insts-thm2: x ∉ fst ‘ set insts if x ∈ ⋃ (fv ‘ snd ‘ set insts) for x
      using insts-def snoc that by blast
    from Fv show ?case

    proof (cases lookup (λx . x = (idn, T)) insts)
      case None
       hence fst p ≠ (idn, T) if p∈set insts for p using that by (auto simp add:
lookup-None-iff)
      hence subst-term [p] (Fv idn T) = Fv idn T if p∈set insts for p
        using that by (cases p) fastforce
      from this None show ?thesis
        unfolding insts-def[symmetric]
        by (induction insts) (auto split: if-splits)
    next
      case (Some a)

        have elem: ((idn, T), a) ∈ set insts using Some lookup-present-eq-key''
insts-thm1 by fastforce
      from this obtain fs bs where split: insts = fs @ ((idn, T), a) # bs
        by (meson split-list)
       hence (idn, T) ∉ set (map fst fs) and (idn, T) ∉ set (map fst bs) using
insts-thm1 by simp-all

      hence fst p ~= (idn, T) if p∈set fs for p
        using that by force
      hence id-subst-fs: subst-term [p] (Fv idn T) = Fv idn T if p∈set fs for p
        using that by (cases p) fastforce
      hence fs-step: fold (λsingle. subst-term [single]) fs (Fv idn T) = Fv idn T
        by (induction fs) (auto split: if-splits)
```

have *change-step*: *subst-term* [((*idn*, *T*), *a*)] (*Fv idn T*) = *a* **by** *simp*

have *bs-sub*: *set bs* ⊆ *set insts* **using** *split* **by** *auto*
hence *x* ∉ *fst* ‘ *set bs*
if *x*∈ ⋃ (*fv* ‘ *snd* ‘ *set bs*) **for** *x*
using *insts-thm2 that split* **by** (*auto simp add*: *image-iff*)

have *v* ∉ *fst* ‘ *set bs* **if** *v* ∈ *fv a* **for** *v*
using *that insts-thm2 elem bs-sub* **by** (*fastforce simp add*: *image-iff*)

hence *id-subst-bs*: *subst-term* [*p*] *a* = *a* **if** *p*∈*set bs* **for** *p*
using *that* **by** (*cases p*, *induction a*) *force*+
hence *bs-step*: *fold* (λ*single*. *subst-term* [*single*]) *bs a* = *a*
by (*induction bs*) *auto*

from *fs-step change-step bs-step split Some* **show** *?thesis* **by** (*simp add*: *insts-def*)
qed
qed (*simp*, *metis subst-term.simps*)+
qed

corollary *subst-term-split-last*:
assumes *distinct* (*map fst* (*xs* @ [*x*]))
assumes ⋀*y* . *y* ∈ (⋃ *t* ∈ *snd* ‘ (*set* (*xs* @ [*x*])) . *fv t*) ⟹ *y* ∉ *fst* ‘ (*set* (*xs* @ [*x*]))
shows *subst-term* (*xs* @ [*x*]) *t* = *subst-term* [*x*] (*subst-term xs t*)
proof−
have *subst-term* (*xs* @ [*x*]) *t* = *fold* (λ*single* . *subst-term* [*single*]) (*xs*@[*x*]) *t*
using *assms subst-term-stepwise* **by** *blast*
also have ... = *subst-term* [*x*] (*fold* (λ*single* . *subst-term* [*single*]) *xs t*)
by *simp*
also have ... = *subst-term* [*x*] (*subst-term xs t*)
using *assms subst-term-stepwise* **by** *simp*
finally show *?thesis* **.**
qed

corollary *subst-type-term-stepwise*:
assumes *distinct* (*map fst instT*)
assumes ⋀*x* . *x* ∈ (⋃ *T* ∈ *snd* ‘ (*set instT*) . *tvsT T*) ⟹ *x* ∉ *fst* ‘ (*set instT*)
assumes *distinct* (*map fst insts*)
assumes ⋀*x* . *x* ∈ (⋃ *t* ∈ *snd* ‘ (*set insts*) . *fv t*) ⟹ *x* ∉ *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*)
using *assms subst-typ*′*-stepwise subst-term-stepwise subst-type-term-split-levels* **by** *auto*

lemma *distinct-fst-imp-distinct*: *distinct* (*map fst l*) ⟹ *distinct l* **by** (*induction*

l) *auto*
lemma *distinct-kv-list*: *distinct l* ⟹ *distinct* (*map* (λ*x*. (*x*, *f x*)) *l*) **by** (*induction l*) *auto*

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*
using *assms* **proof** (*induction t arbitrary*: *l*)
case (*Fv idn T*)
then show *?case*
proof (*cases* (*idn*, *T*) ∈ *set l*)
case *True*
hence ((*idn*, *T*), *ϱ idn T*) ∈ *set* (*map* (λ*x*.(*x*, *case-prod ϱ x*)) *l*) **by** *auto*
moreover have *distinct* (*map fst* (*map* (λ*x*.(*x*, *case-prod ϱ x*)) *l*))
using *Fv*(*1*) **by** (*induction l*) *auto*
ultimately have (*lookup* (λ*x*. *x* = (*idn*, *T*)) (*map* (λ*x*. (*x*, *case x of* (*x*, *xa*) ⇒ *ϱ x xa*)) *l*))
= *Some* (*ϱ idn T*) **using** *lookup-present-eq-key* **by** *fast*
then show *?thesis* **by** *simp*
next
case *False*
then show *?thesis* **using** *Fv* **by** *simp*
qed
qed *auto*

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*)
using *assms* **proof** (*induction t*)
case (*Fv idn T*)
then show *?case*
proof (*cases lookup* (λ*x*. *x* = (*idn*, *T*)) *l*)
case *None*
hence (*idn*, *T*) ∉ *set* (*map fst l*)
by (*metis* (*full-types*) *lookup-None-iff*)

hence (*fold* (λ((*idn*, *T*), *t*) *f x y*. *if x*=*idn* ∧*y*=*T then t else f x y*) *l Fv*) *idn T* = *Fv idn T*
by (*induction l rule*: *rev-induct*) (*auto split*: *if-splits prod.splits*)

then show *?thesis* **by** (*simp add*: *None*)
next
case (*Some a*)

have *elem*: ((*idn*, *T*), *a*) ∈ *set l*
using *Some lookup-present-eq-key*′′ *Fv* **by** *fastforce*
from *this* **obtain** *fs bs* **where** *split*: *l* = *fs* @ ((*idn*, *T*), *a*) # *bs*

```
    by (meson split-list)
  hence (idn, T) ∉ set (map fst fs) and not-in-bs: (idn, T) ∉ set (map fst bs)
using Fv by simp-all

  hence fst p ~= (idn, T) if p∈set fs for p
    using that by force
  hence fs-step: (fold (λ((idn, T), t) f x y. if x=idn ∧y=T then t else f x y) fs
Fv) idn T = Fv idn T
    by (induction fs rule: rev-induct) (fastforce split: if-splits prod.splits)+

  have bs-sub: set bs ⊆ set l using split by auto

  have fst p ~= (idn, T) if p∈set bs for p
    using that not-in-bs by force
  hence bs-step: (fold (λ((idn, T), t) f x y. if x=idn ∧y=T then t else f x y) bs
f) idn T = f idn T
    for f
    by (induction bs rule: rev-induct) (fastforce split: if-splits prod.splits)+

  from fs-step bs-step split Some show ?thesis by simp
 qed
qed auto

lemma subst-typ-combine-single:
 assumes fresh-idn ∉ fst ` tvsT τ
 shows subst-typ [((fresh-idn, S), τ2)] (subst-typ [((idn, S), Tv fresh-idn S)] τ)
   = subst-typ [((idn, S), τ2)] τ
 using assms by (induction τ) auto

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)
 using assms proof (induction insts τ arbitrary: fresh-idns rule: subst-typ.induct)
 case (1 inst a Ts)
 then show ?case by fastforce
next
 case (2 inst idn S)
 show ?case
 proof (cases lookup (λx. x = (idn, S)) inst)
   case None
   hence ((idn, S)) ∉ fst ` set inst
```

```
    by (metis (mono-tags, lifting) list.set-map lookup-None-iff)
  hence 1: (lookup (λx. x = (idn, S))
    (zip (map fst inst) (map2 Tv fresh-idns (map (snd ∘ fst) inst)))) = None
    using 2 by (simp add: lookup-eq-key-not-present)

  have (idn, S) ∉ set (zip fresh-idns (map (snd ∘ fst) inst))
    using 2 set-zip-leftD by fastforce
  hence (lookup (λx. x = (idn, S))
    (zip (zip fresh-idns (map (snd ∘ fst) inst)) (map snd inst))) = None
    using 2 by (simp add: lookup-eq-key-not-present)

  then show ?thesis using None 1 by simp
next
  case (Some ty)
  from this obtain idx where idx: inst ! idx = ((idn, S), ty) idx < length inst
  proof (induction inst)
    case Nil
    then show ?case
      by simp
  next
    case (Cons a as) thm Cons.IH
    have (⋀idx. as ! idx = ((idn, S), ty) ⟹ idx < length as ⟹ thesis)
      by (metis Cons.prems(1) in-set-conv-nth list.set-intros(2))
    then show ?case
   by (meson Cons.prems(1) Cons.prems(2) in-set-conv-nth lookup-present-eq-key')
  qed

   from this obtain fresh-idn where fresh-idn: fresh-idns ! idx = fresh-idn by
simp

  from 2(1) idx fresh-idn have ren:
    (zip (map fst inst) (map2 Tv fresh-idns (map (snd ∘ fst) inst))) ! idx
    = ((idn, S), Tv fresh-idn S)
    by auto
  from this idx(2) have ((idn, S), Tv fresh-idn S) ∈ set
    (zip (map fst inst) (map2 Tv fresh-idns (map (snd ∘ fst) inst)))
  by (metis (no-types, opaque-lifting) 2.prems(1) length-map map-fst-zip map-map
map-snd-zip nth-mem)
  from this have 1: (lookup (λx. x = (idn, S))
    (zip (map fst inst) (map2 Tv fresh-idns (map (snd ∘ fst) inst)))) = Some (Tv
fresh-idn S)
    by (simp add: 2.prems(1) 2.prems(3) lookup-present-eq-key'')

  from 2(1) idx fresh-idn 1 have ((fresh-idn, S), ty)
    ∈ set (zip (zip fresh-idns (map (snd ∘ fst) inst)) (map snd inst))
    using in-set-conv-nth by fastforce
  hence 2: (lookup (λx. x = (fresh-idn, S))
    (zip (zip fresh-idns (map (snd ∘ fst) inst)) (map snd inst))) = Some ty
    by (simp add: 2.prems(1) 2.prems(2) distinct-zipI1 lookup-present-eq-key'')
```

```
    then show ?thesis using Some 1 2 by simp
  qed
qed

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 (λsingle acc . subst-typ [single] acc) (zip (zip fresh-idns (map snd (map
fst insts))) (map snd insts))
        (fold (λsingle acc . subst-typ [single] acc) (zip (map fst insts) (map2 Tv
fresh-idns (map snd (map fst insts)))) τ)
proof−
  have s1: fst ‘ set (zip (map fst insts) (map2 Tv fresh-idns (map snd (map fst
insts))))
    = fst ‘ set insts
  proof−
     have fst ‘ set (zip (map fst insts) (map2 Tv fresh-idns (map snd (map fst
insts))))
      = set (map fst (zip (map fst insts) (map2 Tv fresh-idns (map snd (map fst
insts)))))
     by auto
    also have ... = set (map fst insts) using map-fst-zip assms(1) by auto
    also have ... = fst ‘ set insts by simp
    finally show ?thesis .
  qed

  have snd ‘ set (zip (map fst insts) (map2 Tv fresh-idns (map snd (map fst insts))))
     = set (map2 Tv fresh-idns (map snd (map fst insts))) using map-snd-zip
assms(1)
    by (metis (no-types, lifting) image-set length-map)
  hence (⋃ (tvsT ‘ snd ‘ set (zip (map fst insts) (map2 Tv fresh-idns (map snd
(map fst insts))))))
    = (⋃ (tvsT ‘ set (map2 Tv fresh-idns (map snd (map fst insts)))))
    by simp
  from assms(1) this have s2:
    (⋃ (tvsT ‘ snd ‘ set (zip (map fst insts) (map2 Tv fresh-idns (map snd (map
fst insts))))))
    = (set (zip fresh-idns (map snd (map fst insts))))
    using assms(1) by (induction fresh-idns insts rule: list-induct2) auto
  hence s3: ⋃ (tvsT ‘ snd ‘ set (zip (map fst insts)
                (map2 Tv fresh-idns (map (snd ∘ fst) insts))))
    = set (zip fresh-idns (map snd (map fst insts))) by simp
  have idn ∉ fst ‘ fst ‘ set insts if idn ∈ set fresh-idns for idn
    using that assms by auto
```

hence *I*: (*idn*, *S*) ∉ *fst* ‘ *set insts* **if** *idn* ∈ *set fresh-idns* **for** *idn S*
using *that assms* **by** (*metis fst-conv image-eqI*)

have *u1*: (*subst-typ* (*zip* (*map fst insts*) (*map2 Tv fresh-idns* (*map snd* (*map fst insts*)))) τ)
= *fold* (λ*single acc* . *subst-typ* [*single*] *acc*) (*zip* (*map fst insts*) (*map2 Tv fresh-idns* (*map snd* (*map fst insts*)))) τ
apply (*rule subst-typ-stepwise*)
using *assms* **apply** *simp*
apply (*simp only*: *s1 s2*)
using *assms I* **by** (*metis prod.collapse set-zip-leftD*)

moreover have *u2*: *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)
= *fold* (λ*single acc* . *subst-typ* [*single*] *acc*) (*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)
apply (*rule subst-typ-stepwise*)
using *assms* **apply** (*simp add*: *distinct-zipI1*)
using *assms*
by (*smt UnCI imageE image-eqI length-map map-snd-zip prod.collapse set-map set-zip-leftD*)
ultimately have *unfold*: *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)
= *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)
by *simp*
show *?thesis* **using** *assms subst-typ-combine unfold* **by** *auto*
qed

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*)
using *assms* **proof** (*induction t arbitrary*: *fresh-idns insts*)

```
  case (Abs T t)
  moreover have tvs t ⊆ tvs (Abs T t)  by simp
  ultimately have 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)
    by blast
  moreover have 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)
    using subst-typ-combine Abs.prems by fastforce
  ultimately show ?case by simp
next
  case (App t1 t2)
  moreover have tvs t1 ⊆ tvs (t1 $ t2) tvs t2 ⊆ tvs (t1 $ t2) by auto
  ultimately have subst-typ' insts t1 =
    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))))
t1)
  and subst-typ' insts t2 =
    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))))
t2)
    by blast+
  then show ?case by simp
qed (use subst-typ-combine in auto)


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)
using assms proof (induction t arbitrary: fresh-idns insts)
  case (Fv idn ty)

  then show ?case
  proof (cases lookup (λx. x = (idn, ty)) insts)
    case None
    hence ((idn, ty)) ∉ fst ` set insts
      by (metis (mono-tags, lifting) list.set-map lookup-None-iff)
    hence 1: (lookup (λx. x = (idn, ty))
      (zip (map fst insts) (map2 Fv fresh-idns (map (snd ∘ fst) insts)))) = None
```

using *Fv* **by** (*simp add*: *lookup-eq-key-not-present*)

have (*idn*, *ty*) ∉ *set* (*zip fresh-idns* (*map* (*snd* ∘ *fst*) *insts*))
using *Fv set-zip-leftD* **by** *fastforce*
hence (*lookup* (λ*x*. *x* = (*idn*, *ty*))
(*zip* (*zip fresh-idns* (*map* (*snd* ∘ *fst*) *insts*)) (*map snd insts*))) = *None*
using *Fv* **by** (*simp add*: *lookup-eq-key-not-present*)

then show *?thesis* **using** *None 1* **by** *simp*
next
case (*Some u*)
from *this* **obtain** *idx* **where** *idx*: *insts* ! *idx* = ((*idn*, *ty*), *u*) *idx* < *length insts*
proof (*induction insts*)
case *Nil*
then show *?case*
by *simp*
next
case (*Cons a as*)
have (⋀*idx*. *as* ! *idx* = ((*idn*, *ty*), *u*) ⟹ *idx* < *length as* ⟹ *thesis*)
by (*metis Cons.prems*(*1*) *in-set-conv-nth insert-iff list.set*(*2*))
then show *?case*
by (*meson Cons.prems*(*1*) *Cons.prems*(*2*) *in-set-conv-nth lookup-present-eq-key′*)
qed

from *this* **obtain** *fresh-idn* **where** *fresh-idn*: *fresh-idns* ! *idx* = *fresh-idn* **by** *simp*

from *Fv*(*1*) *idx fresh-idn* **have** *ren*:
(*zip* (*map fst insts*) (*map2 Fv fresh-idns* (*map* (*snd* ∘ *fst*) *insts*))) ! *idx*
= ((*idn*, *ty*), *Fv fresh-idn ty*)
by *auto*
from *this idx*(*2*) **have** ((*idn*, *ty*), *Fv fresh-idn ty*) ∈ *set*
(*zip* (*map fst insts*) (*map2 Fv fresh-idns* (*map* (*snd* ∘ *fst*) *insts*)))
by (*metis* (*no-types*, *opaque-lifting*) *Fv.prems*(*1*) *length-map map-fst-zip map-map map-snd-zip nth-mem*)
from *this* **have** *1*: (*lookup* (λ*x*. *x* = (*idn*, *ty*))
(*zip* (*map fst insts*) (*map2 Fv fresh-idns* (*map* (*snd* ∘ *fst*) *insts*)))) = *Some* (*Fv fresh-idn ty*)
by (*simp add*: *Fv.prems*(*1*) *Fv.prems*(*3*) *lookup-present-eq-key′′*)

from *Fv*(*1*) *idx fresh-idn 1* **have** ((*fresh-idn*, *ty*), *u*)
∈ *set* (*zip* (*zip fresh-idns* (*map* (*snd* ∘ *fst*) *insts*)) (*map snd insts*))
using *in-set-conv-nth* **by** *fastforce*
hence *2*: (*lookup* (λ*x*. *x* = (*fresh-idn*, *ty*))
(*zip* (*zip fresh-idns* (*map* (*snd* ∘ *fst*) *insts*)) (*map snd insts*))) = *Some u*
by (*simp add*: *Fv.prems*(*1*) *Fv.prems*(*2*) *distinct-zipI1 lookup-present-eq-key′′*)
then show *?thesis* **using** *Some 1 2* **by** *simp*
qed

next
case (*App t1 t2*)
moreover have *fv t1* ⊆ *fv* (*t1* $ *t2*) *fv t2* ⊆ *fv* (*t1* $ *t2*) **by** *simp-all*
ultimately have *subst-term insts t1* =
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*)))) *t1*)
and *subst-term insts t2* =
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*)))) *t2*)
by *blast*+
then show *?case* **by** *simp*
qed *auto*

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−
have *s1*: *fst* ‘ *set* (*zip* (*map fst insts*) (*map2 Fv fresh-idns* (*map snd* (*map fst insts*))))
= *fst* ‘ *set insts*
proof−
have *fst* ‘ *set* (*zip* (*map fst insts*) (*map2 Fv fresh-idns* (*map snd* (*map fst insts*))))
= *set* (*map fst* (*zip* (*map fst insts*) (*map2 Fv fresh-idns* (*map snd* (*map fst insts*)))))
by *auto*
also have ... = *set* (*map fst insts*) **using** *map-fst-zip assms*(*1*) **by** *auto*
also have ... = *fst* ‘ *set insts* **by** *simp*
finally show *?thesis* **.**
qed

have *snd* ‘ *set* (*zip* (*map fst insts*) (*map2 Fv fresh-idns* (*map snd* (*map fst insts*))))
= *set* (*map2 Fv fresh-idns* (*map snd* (*map fst insts*))) **using** *map-snd-zip assms*(*1*)
by (*metis* (*no-types*, *lifting*) *image-set length-map*)
hence (⋃ (*fv* ‘ *snd* ‘ *set* (*zip* (*map fst insts*) (*map2 Fv fresh-idns* (*map snd* (*map fst insts*)))))) = (⋃ (*fv* ‘ *set* (*map2 Fv fresh-idns* (*map snd* (*map fst insts*)))))
by *simp*

from *assms*(*1*) *this* **have** *s2*:
(⋃ (*fv* ‘ *snd* ‘ *set* (*zip* (*map fst insts*) (*map2 Fv fresh-idns* (*map snd* (*map fst insts*))))))
= (*set* (*zip fresh-idns* (*map snd* (*map fst insts*))))
using *assms*(*1*) **by** (*induction fresh-idns insts rule*: *list-induct2*) *auto*
hence *s3*: ⋃ (*fv* ‘ *snd* ‘ *set* (*zip* (*map fst insts*)
(*map2 Fv fresh-idns* (*map* (*snd* ∘ *fst*) *insts*))))
= *set* (*zip fresh-idns* (*map snd* (*map fst insts*))) **by** *simp*
have *idn* ∉ *fst* ‘ *fst* ‘ *set insts* **if** *idn* ∈ *set fresh-idns* **for** *idn*
using *that assms* **by** *auto*
hence *I*: (*idn*, *T*) ∉ *fst* ‘ *set insts* **if** *idn* ∈ *set fresh-idns* **for** *idn T*
using *that assms* **by** (*metis fst-conv image-eqI*)

have *u1*: (*subst-term* (*zip* (*map fst insts*) (*map2 Fv fresh-idns* (*map snd* (*map fst insts*)))) *t*)
= *fold* (λ*single acc* . *subst-term* [*single*] *acc*) (*zip* (*map fst insts*) (*map2 Fv fresh-idns* (*map snd* (*map fst insts*)))) *t*
apply (*rule subst-term-stepwise*)
using *assms* **apply** *simp*
apply (*simp only*: *s1 s2*)
using *assms I* **by** (*metis prod.collapse set-zip-leftD*)

moreover have *u2*: *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*)
= *fold* (λ*single acc* . *subst-term* [*single*] *acc*) (*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*)
apply (*rule subst-term-stepwise*)
using *assms* **apply** (*simp add*: *distinct-zipI1*)
using *assms*
by (*smt UnCI imageE image-eqI length-map map-snd-zip prod.collapse set-map set-zip-leftD*)
ultimately have *unfold*: *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*)
= *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*)
by *simp*
show *?thesis* **using** *assms subst-term-combine unfold* **by** *auto*
qed

lemma *subst-term-not-loose-bvar*:

assumes ¬ *loose-bvar t n is-closed b*
shows ¬ *loose-bvar* (*subst-term* [((*idn*,*T*),*b*)] *t*) *n*
using *assms* **by** (*induction t arbitrary*: *n idn T b*) (*auto simp add*: *is-open-def loose-bvar-leq*)

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*
using *assms* **by** (*induction t arbitrary*: *n idn T b*) (*auto simp add*: *is-open-def incr-boundvars-def*)

corollary
assumes *is-closed t is-closed b*
shows *subst-bv b* (*bind-fv* (*idn*, *T*) *t*) = (*subst-term* [((*idn*, *T*),*b*)] *t*)
using *assms bind-fv2-subst-bv1-eq-subst-term*
by (*simp add*: *bind-fv-def subst-bv-def is-open-def*)

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*
using *bind-fv2-subst-bv1-eq-subst-term assms typ-of-imp-closed* **by** *metis*

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*
using *instantiate-var-same-typ bind-fv-def subst-bv-def is-open-def assms* **by** *auto*

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*
using *assms instantiate-var-same-typ′* **by** *simp*

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*
using *assms* **proof** (*induction insts arbitrary*: *B lev*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons x xs*)

from *this* **obtain** *idn ty t* **where** *x*: *x* = ((*idn*, *ty*), *t*) **by** (*metis prod.collapse*)

hence *typ-a*: *typ-of t* = *Some ty* **using** *Cons.prems* **by** *simp*
have *typs*: *list-all* (λ((*idx*, *ty*), *t*) . *typ-of t* = *Some ty*) *xs* **using** *Cons.prems* **by** *simp*
have *not-loose*: ¬ *loose-bvar* (*subst-term* [((*idn*, *ty*), *t*)] *B*) *lev*
using *Cons.prems subst-term-not-loose-bvar typ-a typ-of-imp-closed* **by** *simp*

note *single* = *instantiate-var-same-typ*[*OF typ-a Cons.prems*(*2*), *of idn*]

have *fold* (λ((*idx*, *ty*), *t*) *B* . *subst-bv1* (*bind-fv2* (*idx*, *ty*) *lev B*) *lev t*) (*x* # *xs*) *B*
= *fold* (λ((*idx*, *ty*), *t*) *B*. *subst-bv1* (*bind-fv2* (*idx*, *ty*) *lev B*) *lev t*) *xs*
(*subst-bv1* (*bind-fv2* (*idn*, *ty*) *lev B*) *lev t*)
by (*simp add*: *x*)
also have . . . = *fold* (λ((*idx*, *ty*), *t*) *B*. *subst-bv1* (*bind-fv2* (*idx*, *ty*) *lev B*) *lev t*) *xs*
(*subst-term* [((*idn*, *ty*), *t*)] *B*)
using *single* **by** *simp*
also have . . . = *fold* (λ*single*. *subst-term* [*single*]) *xs* (*subst-term* [((*idn*, *ty*), *t*)] *B*)
using *Cons.IH*[**where** *B* = *subst-term* [((*idn*, *ty*), *t*)] *B*, *OF typs not-loose*] *Cons.prems* **by** *blast*
also have . . . = *fold* (λ*single*. *subst-term* [*single*]) (*x* # *xs*) *B*
by (*simp add*: *x*)
finally show *?case* **.**
qed

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*
using *instantiate-vars-same-typ subst-term-stepwise*[*symmetric*] *assms* **by** *simp*

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(*cases S=empty*)
  case *True*
  then show *?thesis* **by** *simp*
next
  case *False*
  hence *length* (*fresh-name S*) > (*Max* (*image length S*)) **by** *auto*
   hence ∀ *s*∈*S*. *length* (*fresh-name S*) > *length s* **using** *assms* **by** (*simp add*: *le-imp-less-Suc*)
  thus *fresh-name S ∉ S* **by** *blast*
qed

context
  includes *String.literal.lifting*
begin
lift-definition *fresh-name′* :: *String.literal set* ⇒ *String.literal* **is** *fresh-name*
  by (*auto split*: *if-splits*)

lemma [*code*]: *fresh-name′ S* = *String.implode* (*fresh-name* (*String.explode* ‘ *S*))
  by (*metis String.implode-explode-eq fresh-name′.rep-eq*)

lemma *fresh-name′-fresh*:
  assumes *finite S*
  shows *fresh-name′ S ∉ S*
  by (*metis assms finite-imageI fresh-name′.rep-eq fresh-name-fresh rev-image-eqI*)
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*
  using *assms fresh-name′-fresh*
  by (*metis fst-conv variant-name.simps*)

lemma *variant-name-adds*:
  shows *snd* (*variant-name s S*) = *insert* (*fst* (*variant-name s S*)) *S*
  by (*metis fst-conv snd-conv variant-name.simps*)

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*
  apply (*cases s*)
  using *assms fresh-name′-fresh*
 apply (*metis finite-imageI fstI name.simps*(*1*) *rev-image-eqI variant-variable.simps*(*1*))
  using *assms fresh-name′-fresh*
 by (*metis* (*no-types*, *opaque-lifting*) *finite-imageI fst-conv image-iff name.simps*(*2*)
surj-pair variant-variable.simps(*2*))

lemma *variant-variable-adds*:
  shows *snd* (*variant-variable s S*) = *insert* (*fst* (*variant-variable s S*)) *S*
  by (*metis* (*no-types*, *lifting*) *fst-conv snd-conv variant-variable.elims*)

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*
  using *assms* **proof** (*induction n arbitrary*: *s S*)
  case *0*
  then show *?case* **by** *simp*
next
  case (*Suc n*)
  obtain *s′ S′* **where** *s′S′*: *variant-variable s S* = (*s′*, *S′*)
    by *fastforce*
  hence *s′* ∉ *S*
    by (*metis Suc.prems fst-conv variant-variable-fresh*)
  moreover have *I*: ∀ *s*∈*set* (*fst* (*variant-variables n s′ S′*)). *s* ∉ *S′*
    by (*metis Suc.IH Suc.prems s′S′ finite.insertI snd-conv variant-variable-adds*)
  moreover have *S* ⊆ *S′*
    by (*metis insert-iff s′S′ snd-conv subsetI variant-variable-adds*)

ultimately show *?case*
by (*auto simp add*: *Let-def s'S' split*: *prod.splits*)
qed

lemma *variant-names-distinct*:
assumes *finite S*
shows *distinct* (*fst* (*variant-variables n s S*))
using *assms* **proof** (*induction n arbitrary*: *s S*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc n*)
obtain *s' S'* **where** *s'S'*: *variant-variable s S* = (*s'*, *S'*)
by *fastforce*
hence *s'* ∉ *S*
by (*metis Suc.prems fst-conv variant-variable-fresh*)
moreover have *I*: *distinct* (*fst* (*variant-variables n s' S'*))
by (*metis Suc.IH Suc.prems s'S' finite.insertI snd-conv variant-variable-adds*)
moreover have *S* ⊆ *S'*
by (*metis insert-iff s'S' snd-conv subsetI variant-variable-adds*)
ultimately show *?case*
apply (*simp add*: *Let-def s'S' split*: *prod.splits*)
by (*metis Suc.prems finite.insertI fst-conv insertI1 s'S' snd-conv variant-names-fresh variant-variable-adds*)
qed

corollary *variant-names-amount*:
assumes *finite S*
shows *length* (*fst* (*variant-variables n s S*)) = *n*
using *assms* **by** (*induction n arbitrary*: *s S*) (*simp-all add*: *case-prod-beta variant-variable-adds*)

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*)
by (*induction l*) *auto*

lemma *distinct-fresh-rename-ns*: *finite G* ⟹ *distinct* (*fresh-rename-ns n B insts G*)
by (*metis* (*no-types*, *lifting*) *List.finite-set add-vars'-fv finite-UN finite-Un finite-imageI variant-names-distinct*)

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*)
by (*metis* (*no-types*, *lifting*) *List.finite-set add-vars′-fv finite-UN finite-Un finite-imageI variant-names-fresh*)

lemma *length-fresh-rename-ns*: *finite G* ⟹ *length* (*fresh-rename-ns n B insts G*) = *n*
by (*metis* (*no-types*, *lifting*) *List.finite-set add-vars′-fv finite-UN finite-Un finite-imageI variant-names-amount*)

lemma *distinct-fresh-rename-idns*: *finite G* ⟹ *distinct* (*fresh-rename-idns n B insts G*)
using *distinct-fresh-rename-ns* **by** (*metis*)

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*)
using *distinct-fresh-rename-ns map-Pair-zip-replicate-conv map-Pair-zip-replicate-conv*
by (*smt fresh-fresh-rename-ns fst-conv imageE image-eqI list.set-map*)

lemma *length-fresh-rename-idns*: *finite G* ⟹ *length* (*fresh-rename-idns n B insts G*) = *n*
by (*metis length-fresh-rename-ns*)

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*!]:
  $s \to_\beta{}^* s' \Longrightarrow Abs\ T\ s \to_\beta{}^* Abs\ T\ s'$
 by (*induct set*: *rtranclp*) (*blast intro*: *rtranclp.rtrancl-into-rtrancl*)+

lemma *rtrancl-beta-AppL*:
  $s \to_\beta{}^* s' \Longrightarrow s\ \$\ t \to_\beta{}^* s'\ \$\ t$
 by (*induct set*: *rtranclp*) (*blast intro*: *rtranclp.rtrancl-into-rtrancl*)+

lemma *rtrancl-beta-AppR*:
  $t \to_\beta{}^* t' \Longrightarrow s\ \$\ t \to_\beta{}^* s\ \$\ t'$
 by (*induct set*: *rtranclp*) (*blast intro*: *rtranclp.rtrancl-into-rtrancl*)+

lemma *rtrancl-beta-App* [*intro*]:
  $s \to_\beta{}^* s' \Longrightarrow t \to_\beta{}^* t' \Longrightarrow s\ \$\ t \to_\beta{}^* s'\ \$\ t'$
 by (*blast intro*!: *rtrancl-beta-AppL rtrancl-beta-AppR intro*: *rtranclp-trans*)

theorem *subst-bv2-preserves-beta* [*simp*]:
  $r \to_\beta s \Longrightarrow subst\text{-}bv2\ r\ k\ u \to_\beta subst\text{-}bv2\ s\ k\ u$
 by (*induct arbitrary*: *k u set*: *beta*) (*simp-all add*: *subst-bv2-subst-bv2*[*symmetric*])

theorem *subst-bv2-preserves-beta'*: $r \to_\beta{}^* s \Longrightarrow subst\text{-}bv2\ r\ i\ t \to_\beta{}^* subst\text{-}bv2\ s\ i\ t$
 apply (*induct set*: *rtranclp*)
  apply (*rule rtranclp.rtrancl-refl*)
 apply (*erule rtranclp.rtrancl-into-rtrancl*)
 apply (*erule subst-bv2-preserves-beta*)
 done

theorem *lift-preserves-beta* [*simp*]:
  $r \to_\beta s \Longrightarrow lift\ r\ i \to_\beta lift\ s\ i$
proof (*induction arbitrary*: *i set*: *beta*)
 case (*beta T s t*)
 then show *?case*
  using *lift-subst* **by** *force*
qed *auto*
theorem *lift-preserves-beta'*: $r \to_\beta{}^* s \Longrightarrow lift\ r\ i \to_\beta{}^* lift\ s\ i$
 apply (*induct set*: *rtranclp*)
  apply (*rule rtranclp.rtrancl-refl*)
 apply (*erule rtranclp.rtrancl-into-rtrancl*)
 apply (*erule lift-preserves-beta*)
 done

theorem *subst-bv2-preserves-beta2* [*simp*]: $r \to_\beta s \Longrightarrow subst\text{-}bv2\ t\ i\ r \to_\beta{}^* subst\text{-}bv2\ t\ i\ s$
 apply (*induct t arbitrary*: *r s i*)
  apply (*solves* ‹*simp add*: *r-into-rtranclp*›)+
 using *lift-preserves-beta* **by** (*auto simp add*: *rtrancl-beta-App*)

theorem *subst-bv2-preserves-beta2'*: $r \to_\beta^* s \Longrightarrow$ *subst-bv2 t i r* $\to_\beta^*$ *subst-bv2 t i s*
apply (*induct set*: *rtranclp*)
apply (*auto elim*: *rtranclp-trans subst-bv2-preserves-beta2*)
done

lemma *beta-preserves-typ-of1*: *typ-of1 Ts r* = *Some T* $\Longrightarrow$ $r \to_\beta s \Longrightarrow$ *typ-of1 Ts s* = *Some T*
proof (*induction Ts r arbitrary*: *s T rule*: *typ-of1.induct*)
case (*4 Ts T body*)
then show *?case*
by (*smt beta-cases*(*3*) *typ-of1.simps*(*4*) *typ-of-Abs-body-typ'*)
next
case (*5 Ts f u*)
from *this* **obtain** *argT* **where** *argT*: *typ-of1 Ts u* = *Some argT* **and** *typ-of1 Ts f* = *Some* (*argT* $\to$ *T*)
by (*meson typ-of1-split-App-obtains*)

from *5* **show** *?case* **apply** −
apply (*ind-cases f* $ *u* $\to_\beta$ *s* **for** *f u s*)
using ‹*typ-of1 Ts f* = *Some* (*argT* $\to$ *T*)› *argT typ-of1-subst-bv-gen' typ-of-Abs-body-typ'* **by** (*fastforce simp add*: *substn-subst-n*)+
qed (*use beta.cases* **in** ‹*blast*+›)

lemma *beta-preserves-typ-of*: *typ-of r* = *Some T* $\Longrightarrow$ $r \to_\beta s \Longrightarrow$ *typ-of s* = *Some T*
by (*metis beta-preserves-typ-of1 typ-of-def*)

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 (*induction rule*: *rtranclp.induct*)
case (*rtrancl-refl a*)
then show *?case*
by *simp*
next
case (*rtrancl-into-rtrancl a b c*)
then show *?case*
using *beta-preserves-typ-of1* **by** *blast*
qed

lemma *beta-reducible-imp-beta-step*: *beta-reducible t* $\Longrightarrow \exists t'.\ t \to_\beta t'$
proof (*induction t*)
case (*App t1 t2*)
then show *?case* **using** *App* **by** (*cases t1*) *auto*
qed *auto*

lemma *beta-step-imp-beta-reducible*: $t \to_\beta t' \Longrightarrow$ *beta-reducible t*
proof (*induction t t' rule*: *beta.induct*)

```
  case (beta T s t)
  then show ?case by simp
next
case (appL s t u)
  then show ?case by (cases s) auto
next
  case (appR s t u)
  then show ?case using beta-reducible.elims by blast
next
  case (abs s t T)
  then show ?case by simp
qed

lemma beta-norm-imp-beta-reds: assumes beta-norm t = Some t' shows t →β*
t'
  using assms proof (induction arbitrary: t t' rule: beta-norm.fixp-induct)
  case 1
  then show ?case
    by (smt Option.is-none-def ccpo.admissibleI chain-fun flat-lub-def flat-ord-def
fun-lub-def
        insertCI is-none-code(2) mem-Collect-eq option.lub-upper subsetI)
next
  case 2
  then show ?case
    by simp
next
  case (3 comp)
  then show ?case
  proof(cases t)
  next
    case (App f u)
    note fu = App
    then show ?thesis
    proof (cases comp f)
      case None
      show ?thesis
      proof(cases f)
        case (Abs B b)
        then show ?thesis
      by (metis (mono-tags, lifting) 3.IH 3.prems Core.subst-bv-def Core.term.simps(29)

                    Core.term.simps(30) beta fu rtranclp.rtrancl-into-rtrancl rtran-
clp.rtrancl-refl rtranclp-trans)
      qed (use 3 None in ‹simp-all add: fu split: term.splits option.splits if-splits›)
    next
      case (Some fo)
      then show ?thesis
      proof(cases fo)
        case (Ct n T)
```

```
  then show ?thesis
  proof(cases f)
    case (Abs B b)
    then show ?thesis
  by (metis (no-types, lifting) 3.IH 3.prems Core.subst-bv-def Core.term.simps(29)
        Core.term.simps(30) beta converse-rtranclp-into-rtranclp fu)
 qed (use 3 Some in ‹auto simp add: fu split: term.splits option.splits if-split›)
next
  case (Fv n T)
  then show ?thesis
  proof(cases f)
    case (Abs B b)
    then show ?thesis
  by (metis (no-types, lifting) 3.IH 3.prems Core.subst-bv-def Core.term.simps(29)
        Core.term.simps(30) beta converse-rtranclp-into-rtranclp fu)
 qed (use 3 Some in ‹auto simp add: fu split: term.splits option.splits if-split›)
next
case (Bv n)
  then show ?thesis
  proof(cases f)
    case (Abs B b)
    then show ?thesis
  by (metis (no-types, lifting) 3.IH 3.prems Core.subst-bv-def Core.term.simps(29)
        Core.term.simps(30) beta converse-rtranclp-into-rtranclp fu)
 qed (use 3 Some in ‹auto simp add: fu split: term.splits option.splits if-split›)
next
  case (Abs T t)
  then show ?thesis
  proof(cases f)
    case (Ct n C)
    show ?thesis
     by (metis 3.IH Abs Core.term.simps(11) Ct Some beta-reducible.simps(7)

        beta-step-imp-beta-reducible converse-rtranclpE)
  next
    case (Fv n C)
    then show ?thesis
  by (metis 3.IH Abs Fv Some beta-reducible.simps(1,4,8) beta-step-imp-beta-reducible

        converse-rtranclpE)
  next
    case (Bv n)
    then show ?thesis
  by (metis 3.IH Abs Some beta-cases(1) converse-rtranclpE term.distinct(15))
  next
    case (Abs B b)
    then show ?thesis
  by (metis (no-types, lifting) 3.IH 3.prems Core.subst-bv-def Core.term.simps(29)
        Core.term.simps(30) beta converse-rtranclp-into-rtranclp fu)
```

next
case (*App a b*)
then show *?thesis*
using *3* **apply** (*simp add*: *fu Some split*: *term.splits option.splits if-splits*; *fast?*)
by (*metis Core.subst-bv-def beta converse-rtranclp-into-rtranclp rtrancl-beta-AppL rtranclp-trans*)
qed
next
case *AppO*: (*App f u*)
then show *?thesis*
proof(*cases f*)
case (*Ct n C*)
show *?thesis*
using *3 Some* **apply** (*simp add*: *Ct AppO fu split*: *term.splits option.splits if-split*; *fast?*)
by (*metis Core.subst-bv-def beta converse-rtranclp-into-rtranclp*)
next
case (*Fv n C*)
then show *?thesis*
using *3 Some* **apply** (*simp add*: *Fv AppO fu split*: *term.splits option.splits if-split*; *fast?*)
by (*metis Core.subst-bv-def beta converse-rtranclp-into-rtranclp*)
next
case (*Bv n*)
then show *?thesis*
using *3 Some* **apply** (*simp add*: *Bv AppO fu split*: *term.splits option.splits if-split*; *fast?*)
by (*metis Core.subst-bv-def beta converse-rtranclp-into-rtranclp*)
next
case (*Abs B b*)
then show *?thesis*
using *3 Some* **apply** (*simp add*: *Abs AppO fu split*: *term.splits option.splits if-split*; *fast?*)
by (*metis Core.subst-bv-def beta converse-rtranclp-into-rtranclp*)
next
case (*App a b*)
then show *?thesis*
using *3 Some* **apply** (*simp add*: *App AppO fu split*: *term.splits option.splits if-split*; *fast?*)
by (*metis Core.subst-bv-def beta converse-rtranclp-into-rtranclp*)
qed
qed
qed
qed *auto*
qed

corollary *beta-norm t* = *Some t'* ⟹ *typ-of1 Ts t* = *Some T* ⟹ *typ-of1 Ts t'* = *Some T*

using *beta-norm-imp-beta-reds beta-star-preserves-typ-of1* **by** *blast*

lemma *beta-imp-beta-norm*: **assumes** $t \rightarrow_\beta t' \neg$ *beta-reducible* t' **shows** *beta-norm* $t =$ *Some* t'
using *assms* **proof** (*induction rule*: *beta.induct*)
case (*beta* T s t)
then show *?case* **using** *not-beta-reducible-imp-beta-norm-unchanged* **by** (*auto simp add*: *subst-bv-def substn-subst-n*)
next
case (*appL* s t u)
hence *t*: $\neg$ *beta-reducible* t **by** (*fastforce elim*: *beta-reducible.elims*)
hence *IH*: *beta-norm* $s =$ *Some* t **using** *appL.IH* **by** *simp*
from *appL* **have** *u*: $\neg$ *beta-reducible* u
using *beta-reducible.elims* **by** *blast*
show *?case*
apply (*cases s*; *cases t*)
using *not-beta-reducible-imp-beta-norm-unchanged IH t u appL.prems* **by** *auto*
next
case (*appR* s t u)
hence *t*: $\neg$ *beta-reducible* t
using *beta-reducible.elims* **by** *blast*
hence *IH*: *beta-norm* $s =$ *Some* t **using** *appR.IH* **by** *simp*
from *appR* **have** *u*: $\neg$ *beta-reducible* u
using *beta-reducible.elims* **by** *blast*
show *?case*
apply (*cases s*; *cases u*)
using *not-beta-reducible-imp-beta-norm-unchanged IH t u appR.prems* **by** *auto*
next
case (*abs* s t T)
then show *?case* **by** *auto*
qed

lemma *beta-subst-bv1*: $s \rightarrow_\beta t \Longrightarrow$ *subst-bv1* s *lev* $x \rightarrow_\beta$ *subst-bv1* t *lev* x
proof (*induction s t arbitrary*: *lev rule*: *beta.induct*)
case (*beta* T s t)
then show *?case*
using *beta.beta subst-bv2-preserves-beta substn-subst-n* **by** *presburger*
qed (*auto simp add*: *subst-bv-def*)

lemma *beta-subst-bv*: $s \rightarrow_\beta t \Longrightarrow$ *subst-bv* x $s \rightarrow_\beta$ *subst-bv* x t
by (*simp add*: *substn-subst-0'*)

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 (*induction subst-bv1 s* (*length* ($T \# Ts$)) x *subst-bv1 t* (*length* ($T \# Ts$)) x

arbitrary: *s t T T Ts ty rule*: *beta.induct*)
case (*beta T s t*)
then show *?case*
by (*metis beta.simps length-Cons loose-bvar-Suc no-loose-bvar-imp-no-subst-bv1 typ-of1-imp-no-loose-bvar*)
next
case (*appL s t u*)
then show *?case*
by (*metis beta.appL length-Cons loose-bvar-Suc no-loose-bvar-imp-no-subst-bv1 typ-of1-imp-no-loose-bvar*)
next
case (*appR s t u*)
then show *?case*
by (*metis beta.simps length-Cons loose-bvar-Suc no-loose-bvar-imp-no-subst-bv1 typ-of1-imp-no-loose-bvar*)
next
case (*abs s t bT sa ta T Ts rT*)
obtain *s'* **where** *Abs bT s' = sa*
using *abs.hyps*(*3*) *abs.prems loose-bvar-Suc no-loose-bvar-imp-no-subst-bv1 typ-of1-imp-no-loose-bvar*
by (*metis length-Cons*)
moreover obtain *t'* **where** *Abs bT t' = ta*
using *abs.hyps*(*4*) *abs.prems loose-bvar-Suc no-loose-bvar-imp-no-subst-bv1 typ-of1-imp-no-loose-bvar*
by (*metis length-Cons*)
ultimately have *s' →β t'*
by (*metis abs.hyps*(*1*) *abs.hyps*(*3*) *abs.hyps*(*4*) *abs.prems*(*1*) *abs.prems*(*2*) *length-Cons*

loose-bvar-Suc no-loose-bvar-imp-no-subst-bv1 term.inject(*4*) *typ-of1-imp-no-loose-bvar*)
then show *?case*
using ‹*Abs bT s' = sa*› ‹*Abs bT t' = ta*› **by** *blast*
qed

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* (λ*t*. *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*
by (*induction t lev* []::*term list rule*: *subst-bvs1.induct*)*auto*

lemma *subst-bvs1'-eq* [*simp*]: *args* ≠ [] ⟹ *subst-bvs1'* (*Bv k*) *k args* = *args* ! *0*
by *simp*
lemma *subst-bvs1'-eq'* [*simp*]: *i* < *length args* ⟹ *subst-bvs1'* (*Bv* (*k*+*i*)) *k args* =

args ! *i*
by *auto*

lemma *subst-bvs1′-gt* [*simp*]:
i + *length args* < *j* ⟹ *subst-bvs1′* (*Bv j*) *i args* = *Bv* (*j* − *length args*)
by *auto*

lemma *subst-bv2-lt* [*simp*]: *j* < *i* ⟹ *subst-bvs1′* (*Bv j*) *i u* = *Bv j*
by *simp*

lemma *subst-bvs1′-App*[*simp*]: *subst-bvs1′* (*s*$*t*) *k args*
= *subst-bvs1′ s k args* $ *subst-bvs1′ t k args*
by *simp*

lemma *incr-bv-incr-bv*:
i < *k* + *1* ⟹ *incr-bv inc2* (*k*+*inc1*) (*incr-bv inc1 i t*) = *incr-bv inc1 i* (*incr-bv inc2 k t*)
proof (*induction t arbitrary*: *i k*)
case (*Abs T t*)
then show *?case*
by (*metis Suc-eq-plus1 add-Suc add-mono1 incr-bv.simps*(*2*))
qed *auto*

lemma *subst-bvs1-subst-bvs1′*: *subst-bvs1 t n s* = *subst-bvs1′ t n* (*map* (*incr-bv n 0*) *s*)
proof (*induction t arbitrary*: *n*)
case (*Abs T t*)
then show *?case*
by (*simp add*: *incr-boundvars-def incr-bv-combine*)
(*metis One-nat-def comp-apply incr-bv-combine plus-1-eq-Suc*)
qed (*auto simp add*: *incr-boundvars-def incr-bv-combine*)

theorem *subst-bvs1-subst-bvs1′-0*: *subst-bvs1 t 0 s* = *subst-bvs1′ t 0 s*
proof−
have *subst-bvs1 t 0 s* = *subst-bvs1′ t 0* (*map* (*incr-bv 0 0*) *s*)
using *subst-bvs1-subst-bvs1′* **by** *blast*
moreover have *map* (*incr-bv 0 0*) *s* = *s*
by (*induction s*) *auto*
ultimately show *?thesis*
by *simp*
qed

corollary *subst-bvs-subst-bvs1′*: *subst-bvs s t* = *subst-bvs1′ t 0 s*
using *subst-bvs-def subst-bvs1-subst-bvs1′-0* **by** *simp*

lemma *no-loose-bvar-subst-bvs1′-unchanged*: ¬ *loose-bvar t lev* ⟹ *subst-bvs1′ t lev args* = *t*
by (*induction t lev args rule*: *subst-bvs1′.induct*) *auto*

lemma *subst-bvs1'-step*: $\forall x \in set\ (a\#args)\ .\ is\text{-}closed\ x \Longrightarrow$
subst-bvs1' t lev (a#args) = subst-bvs1' (subst-bv2 t lev a) lev args
proof (*induction t lev args rule*: *subst-bvs1'.induct*)
case (*1 i lev args*)
then show *?case*
using *no-loose-bvar-subst-bvs1'-unchanged*
by (*simp add*: *is-open-def*)
(*metis Suc-diff-Suc le-add1 le-add-same-cancel1 less-antisym loose-bvar-leq not-less-eq*)
qed (*auto simp add*: *is-open-def*)

lemma *not-loose-bvar-incr-bv*: $\neg$ *loose-bvar a lev* $\Longrightarrow \neg$ *loose-bvar* (*incr-bv inc lev a*) (*lev+inc*)
by (*induction a lev rule*: *loose-bvar.induct*) *auto*

lemma *not-loose-bvar-incr-bv-less*:
$i < j \Longrightarrow \neg$ *loose-bvar* (*incr-bv inc i a*) (*lev+inc*) $\Longrightarrow \neg$ *loose-bvar* (*incr-bv inc j a*) (*lev+inc*)
proof (*induction inc i a arbitrary*: *lev j rule*: *incr-bv.induct*)
case (*2 inc n T body*)
then show *?case*
by (*metis Suc-eq-plus1 add-Suc add-mono1 incr-bv.simps*(*2*) *loose-bvar.simps*(*3*))
qed (*auto split*: *if-splits*)

lemma *subst-bvs1'-step-work*: $\forall x \in set\ args\ .\ is\text{-}closed\ x \Longrightarrow \neg$ *loose-bvar* (*subst-bv2 t lev a*) *lev* $\Longrightarrow$
subst-bvs1' t lev (a#args) = subst-bvs1' (subst-bv2 t lev a) lev args
proof (*induction t lev args arbitrary*: *a rule*: *subst-bvs1'.induct*)
case (*1 i*)
then show *?case* **using** *no-loose-bvar-subst-bvs1'-unchanged*
by (*auto simp add*: *is-open-def*)
next
case (*2 T body lev args*)
then show *?case* **using** *no-loose-bvar-subst-bvs1'-unchanged*
by (*auto simp add*: *is-open-def*)
next
case (*3 f t lev u*)
then show *?case* **using** *no-loose-bvar-subst-bvs1'-unchanged*
by (*auto simp add*: *is-open-def*)
next
case (*4-1 v va uu uv*)
then show *?case* **using** *no-loose-bvar-subst-bvs1'-unchanged*
by (*auto simp add*: *is-open-def*)
next
case (*4-2 v va uu uv*)
then show *?case* **using** *no-loose-bvar-subst-bvs1'-unchanged*
by (*auto simp add*: *is-open-def*)
qed

lemma *is-closed-subst-bv2-unchanged*: *is-closed t* $\Longrightarrow$ *subst-bv2 t n u* = *t*
by (*metis is-open-def lift-def loose-bvar-Suc no-loose-bvar-no-incr subst-bv2-lift zero-induct*)

lemma *subst-bvs1′-step-extend-lower-level*: $\forall$ *x* $\in$ *set* (*a*#*args*) . *is-closed x* $\Longrightarrow$
subst-bv2 (*subst-bvs1′ t* (*Suc lev*) *args*) *lev a*
= *subst-bvs1′ t lev* (*a*#*args*)
proof (*induction t lev a*#*args arbitrary*: *a args rule*: *subst-bvs1′.induct*)
case (*1 i lev*)
have *subst-bv2* (*subst-bvs1′* (*Bv i*) (*Suc lev*) *args*) *lev a* =
subst-bvs1′ (*Bv i*) *lev* (*a* # *args*)
if *i* < *Suc lev*
using *that* **by** *auto*
moreover have *subst-bv2* (*subst-bvs1′* (*Bv i*) (*Suc lev*) *args*) *lev a* =
subst-bvs1′ (*Bv i*) *lev* (*a* # *args*)
if *i* − *Suc lev* < *length args* ¬ *i* < *Suc lev*
proof−
have *subst-bv2* (*subst-bvs1′* (*Bv i*) (*Suc lev*) *args*) *lev a* = *subst-bv2* (*args* ! (*i* − *Suc lev*)) *lev a*
using *that* **by** *simp*
also have . . . = *args* ! (*i* − *Suc lev*)
using *1 that*(*1*) **by** (*auto simp add*: *is-closed-subst-bv2-unchanged*)
also have *subst-bvs1′* (*Bv i*) *lev* (*a* # *args*) = *args* ! (*i* − *Suc lev*)
using *that* **by** *auto*
finally show *?thesis*
by *simp*
qed
moreover have *subst-bv2*(*subst-bvs1′* (*Bv i*) (*Suc lev*) *args*) *lev a* =
subst-bvs1′ (*Bv i*) *lev* (*a* # *args*)
if *i* ≥ *Suc lev i* − *lev* ≥ *length args* ¬ *i* < *Suc lev*
using *that 1* **by** (*auto simp add*: *is-closed-subst-bv2-unchanged*)
ultimately show *?case* **by** (*auto simp add*: *is-open-def split*: *if-splits*)
qed (*auto simp add*: *is-open-def*)

corollary *subst-bvs-extend-lower-level*:
$\forall$ *x* $\in$ *set* (*a*#*args*) . *is-closed x* $\Longrightarrow$
subst-bv a (*subst-bvs1′ t 1 args*) = *subst-bvs* (*a*#*args*) *t*
using *subst-bvs1′-step-extend-lower-level*
by (*simp add*: *subst-bvs-subst-bvs1′ substn-subst-0′*)

lemma *subst-bvs1′-preserves-beta*:
$\forall$ *x* $\in$ *set u* . *is-closed x* $\Longrightarrow$ *r* $\rightarrow_\beta$ *s* $\Longrightarrow$ *subst-bvs1′ r k u* $\rightarrow_\beta$ *subst-bvs1′ s k u*
proof (*induction u arbitrary*: *r s*)
case *Nil*
then show *?case* **by** *auto*
next
case (*Cons a u*)

hence *subst-bv2 r k a* $\rightarrow_\beta$ *subst-bv2 s k a*
by *simp*
hence *subst-bvs1′* (*subst-bv2 r k a*) *k u* $\rightarrow_\beta$ *subst-bvs1′* (*subst-bv2 s k a*) *k u*
using *Cons* **by** *simp*
then show *?case*
by (*simp add*: *subst-bvs1′-step*[*symmetric*] *Cons.prems*(*1*))
qed

lemma *subst-bvs1′-fold*: $\forall$ *x* $\in$ *set args* . *is-closed x* $\Longrightarrow$
subst-bvs1′ t lev args = *fold* (λ*arg t* . *subst-bv2 t lev arg*) *args t*
by (*induction args arbitrary*: *t*) (*simp-all add*: *subst-bvs1′-step*)

lemma *subst-bvs1′-Abs*[*simp*]: $\forall$ *x* $\in$ *set args* . *is-closed x* $\Longrightarrow$
subst-bvs1′ (*Abs T t*) *lev args* = *Abs T* (*subst-bvs1′ t* (*Suc lev*) *args*)
by (*simp add*: *is-open-def map-idI*)

lemma *subst-bvs-Abs*[*simp*]: $\forall$ *x* $\in$ *set args* . *is-closed x* $\Longrightarrow$
subst-bvs args (*Abs T t*) = *Abs T* (*subst-bvs1′ t 1 args*)
using *subst-bvs1′-Abs subst-bvs-subst-bvs1′* **by** *auto*

lemma *subst-bvs1′-incr-bv* [*simp*]:
subst-bvs1′ (*incr-bv* (*length ss*) *k t*) *k ss* = *t*
proof (*induct t arbitrary*: *k ss*)
case (*Abs T t*)
then show *?case*
by *simp* (*metis length-map*)
qed *auto*

lemma *lift-subst-bvs1′* [*simp*]:
j < *i* + *1* $\Longrightarrow$ *lift* (*subst-bvs1′ t j ss*) *i*
= *subst-bvs1′* (*lift t* (*i* + *length ss*)) *j* (*map* (λ*s* . *lift s i*) *ss*)
proof (*induct t arbitrary*: *i j ss*)
case (*Abs T t*)
hence *I*: *lift* (*subst-bvs1′ t* (*Suc j*) (*map* (λ*t*. *lift t 0*) *ss*)) (*Suc i*) =
subst-bvs1′ (*lift t* (*Suc i* + *length* (*map* (λ*t*. *lift t 0*) *ss*))) (*Suc j*) (*map* (λ*a*. *lift a* (*Suc i*)) (*map* (λ*t*. *lift t 0*) *ss*))
by *auto*

have *lift* (*subst-bvs1′* (*Abs T t*) *j ss*) *i*
= *Abs T* (*lift* (*subst-bvs1′ t* (*Suc j*) (*map* (λ*t*. *lift t 0*) *ss*)) (*Suc i*))
by *simp*
also have ... = *Abs T*
(*subst-bvs1′* (*lift t* (*Suc i* + *length* (*map* (*incr-bv 1 0*) *ss*))) (*Suc j*)
(*map* (*incr-bv 1* (*Suc i*)) (*map* (*incr-bv 1 0*) *ss*)))
using *I* **by** *auto*
also have ... = *Abs T*
(*subst-bvs1′* (*lift t* (*Suc i* + *length* (*map* (*incr-bv 1 0*) *ss*))) (*Suc j*)
(*map* (λ*t*. *lift t 0*) (*map* (λ*t*. *lift t i*) *ss*)))
proof–

have *map* (λ*t* . *lift t* (*Suc i*)) (*map* (λ*t*. *lift t 0*) *ss*) = *map* (λ*t*. *lift t 0*) (*map* (λ*t*. *lift t i*) *ss*)
using *lift-lift* **by** *auto*
thus *?thesis* **unfolding** *lift-def*
by *argo*
qed
also have ... = *subst-bvs1′* (*Abs T* (*lift t* (*Suc i* + *length* (*map* (*incr-bv 1 0*) *ss*)))) *j*
(*map* (λ*t*. *lift t i*) *ss*)
by *auto*
finally show *?case*
by *simp*
qed (*auto simp add*: *diff-Suc lift-lift split*: *nat.split*)

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 (*induct t arbitrary*: *i j ss*)
case (*Abs T t*)
then show *?case* **using** *lift-lift*
by *simp* (*smt comp-apply map-eq-conv zero-less-Suc*)
qed *auto*

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(*induction t arbitrary*: *i j u vs*)
case (*Abs T t*)
then show *?case*
by *simp* (*smt One-nat-def Suc-eq-plus1 Suc-less-eq comp-apply lift-lift lift-def lift-subst-bvs1′-lt map-eq-conv map-map zero-less-Suc*)
qed (*use subst-bv2-lift* **in** *auto*)

lemma *fv-subst-bv2-upper-bound*: *fv* (*subst-bv2 t lev u*) ⊆ *fv t* ∪ *fv u*
by (*induction t lev u rule*: *subst-bv2.induct*) *auto*
lemma *beta-fv*: *s* →$_\beta$ *t* ⟹ *fv t* ⊆ *fv s*
by (*induction rule*: *beta.induct*) (*use fv-subst-bv2-upper-bound* **in** *auto*)

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*
by (*induction t k us arbitrary*: *lev rule*: *subst-bvs1′.induct*)
(*use is-open-def loose-bvar-iff-exist-loose-bvar1* **in** ‹*auto simp add*: *is-open-def*›)

lemma *is-closed-subst-bvs1′-closeds*: ¬ *is-dependent t* ⟹ ∀ *x*∈*set us* . *is-closed x*
⟹ ¬ *is-dependent* (*subst-bvs1′ t* (*Suc k*) *us*)
by (*simp add*: *is-dependent-def loose-bvar1-subst-bvs1′-closeds*)

end

Facts about beta normalization involving theories

theory *BetaNormProof*
imports *BetaNorm Theory*
begin

lemma *beta-preserves-term-ok'*: *term-ok'* Σ *r* $\Longrightarrow$ *r* $\rightarrow_\beta$ *s* $\Longrightarrow$ *term-ok'* Σ *s*
proof (*induction r arbitrary*: *s*)
case (*Ct n T*)
then show *?case*
apply (*simp add*: *tinstT-def split*: *option.splits*)

using *beta-reducible.simps*(*7*) *beta-step-imp-beta-reducible* **by** *blast*
next
case (*Fv n T*)
then show *?case*
by *auto*
next
case (*Bv n*)
then show *?case*
by *auto*
next
case (*Abs R r*)
then show *?case*
by *auto*
next
case (*App f u*)
then show *?case*
apply −
apply (*ind-cases f* $ *u* $\rightarrow_\beta$ *s* **for** *f u s*)
using *term-ok'-subst-bv2 term-ok'.simps*(*4*) *term-ok'.simps*(*5*) **apply** *blast*
using *term-ok'.simps*(*4*) **apply** *blast*
using *term-ok'.simps*(*4*) **apply** *blast*
done
qed

lemma *beta-preserves-term-ok*: *term-ok* Θ *r* $\Longrightarrow$ *r* $\rightarrow_\beta$ *s* $\Longrightarrow$ *term-ok* Θ *s*
proof −
assume *a1*: *term-ok* Θ *r*
assume *a2*: *r* $\rightarrow_\beta$ *s*
then have *None* $\neq$ *typ-of1* [] *s*
using *a1 beta-preserves-typ-of1*
by (*metis has-typ1-imp-typ-of1 has-typ-def option.distinct*(*1*) *term-ok-def wt-term-def*)
then show *?thesis*
using *a2 a1 beta-preserves-term-ok' has-typ-iff-typ-of wt-term-def typ-of-def*
by (*meson beta-preserves-typ-of term-ok-def wf-term-iff-term-ok'*)
qed

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

by (*induction rule*: *rtranclp.induct*) (*auto simp add*: *beta-preserves-term-ok'*)

corollary *beta-star-preserves-term-ok*: $r \to_\beta^* s \Longrightarrow$ *term-ok thy r* $\Longrightarrow$ *term-ok thy s*
using *beta-star-preserves-term-ok' beta-star-preserves-typ-of1 wt-term-def typ-of-def*
by *auto*

corollary *term-ok-beta-norm*: *term-ok thy t* $\Longrightarrow$ *beta-norm t* = *Some t'* $\Longrightarrow$ *term-ok thy t'*
using *beta-norm-imp-beta-reds beta-star-preserves-term-ok* **by** *blast*

end

9 Eta Normalization

theory *EtaNorm*
imports *Term BetaNorm*
begin

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

abbreviation
eta-reds :: *term* $\Rightarrow$ *term* $\Rightarrow$ *bool* (**infixl** $\to_\eta^*$ *50*) **where**
s $\to_\eta^*$ *t* $\equiv$ *eta*** *s t*

abbreviation
eta-red0 :: *term* $\Rightarrow$ *term* $\Rightarrow$ *bool* (**infixl** $\to_\eta^=$ *50*) **where**
s $\to_\eta^=$ *t* $\equiv$ *eta*$^{==}$ *s t*

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

lemma *subst-bv2-not-free* [*simp*]: $\neg$ *loose-bvar1 s i* $\Longrightarrow$ *subst-bv2 s i t* = *subst-bv2 s i u*
by (*induction s arbitrary*: *i t u*) (*simp-all add*:)

lemma *free-lift* [*simp*]:
loose-bvar1 (*lift t k*) *i* = (*i* < *k* $\wedge$ *loose-bvar1 t i* $\vee$ *k* < *i* $\wedge$ *loose-bvar1 t* (*i* − *1*))
by (*induct t arbitrary*: *i k*) (*auto cong*: *conj-cong*)

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*))
 apply (*induct s arbitrary*: *i k t*)
 using *free-lift* **apply** (*simp-all add*: *diff-Suc split*: *nat.split*)
 by *blast*

lemma *free-eta*: *s* →$_\eta$ *t* ⟹ *loose-bvar1 t i* = *loose-bvar1 s i*
 apply (*induct arbitrary*: *i set*: *eta*)
 apply (*simp-all cong*: *conj-cong*)
 using *is-dependent-def loose-bvar1-decr'''' loose-bvar1-decr''''* **by** *blast*

lemma *not-free-eta*:
  s →$_\eta$ *t* ⟹ ¬ *loose-bvar1 s i* ⟹ ¬ *loose-bvar1 t i*
 by (*simp add*: *free-eta*)

lemma *no-loose-bvar1-subst-bv2-decr*: ¬ *loose-bvar1 t i* ⟹ *subst-bv2 t i x* = *decr i t*
 by (*induction t i x rule*: *subst-bv2.induct*) *auto*

lemma *eta-subst-bv2* [*simp*]:
  s →$_\eta$ *t* ⟹ *subst-bv2 s i u* →$_\eta$ *subst-bv2 t i u*
proof (*induction s t arbitrary*: *u i rule*: *eta.induct*)
 case (*eta s T*)
 hence *1*: ¬ *loose-bvar1 s 0*
  using *is-dependent-def* **by** *simp*
 have *decr 0 s* = *subst-bv2 s 0 dummy* **for** *dummy*
  using *no-loose-bvar1-subst-bv2-decr*[*symmetric*, *OF 1*, *of dummy*] **.**
 from *this* **obtain** *dummy* **where** *dummy*: *decr 0 s* = *subst-bv2 s 0 dummy*
  by *simp*

 show *?case*
  using *1* **apply** (*simp add*: *dummy subst-bv2-subst-bv2* [*symmetric*])
  using *free-lift is-dependent-def no-loose-bvar1-subst-bv2-decr* **by** *auto*
qed *auto*

theorem *lift-subst-bv2-dummy*: ¬ *loose-bvar s i* ⟹ *lift* (*decr i s*) *i* = *s*
 by (*induct s arbitrary*: *i*) *simp-all*

lemma *decr-is-closed*[*simp*]: *is-closed t* ⟹ *decr lev t* = *t*
 by (*metis is-open-def lift-subst-bv2-dummy lift-def loose-bvar-Suc loose-bvar-incr-bvar no-loose-bvar-no-incr zero-induct*)

lemma *eta-reducible-imp-eta-step*: *eta-reducible t* ⟹ ∃ *t'*. *t* →$_\eta$ *t'*
 by (*induction t rule*: *eta-reducible.induct*) *auto*

lemma *eta-step-imp-eta-reducible*: *t* →$_\eta$ *t'* ⟹ *eta-reducible t*
proof (*induction t t' rule*: *eta.induct*)

```
  case (abs s t T)
  show ?case
  proof(cases s)
    case (App u v)
    then show ?thesis by (cases v; use abs eta-reducible-Abs in metis)
  qed (use abs in auto)
qed auto

lemma eta-reds-appR: s →η* t ⟹ u $ s →η* u $ t
  by (induction s t rule: rtranclp.induct) (auto simp add: rtranclp.rtrancl-into-rtrancl)
lemma eta-reds-appL: s →η* t ⟹ s $ u →η* t $ u
  by (induction s t rule: rtranclp.induct) (auto simp add: rtranclp.rtrancl-into-rtrancl)
lemma eta-reds-abs: s →η* t ⟹ Abs T s →η* Abs T t
  by (induction s t rule: rtranclp.induct) (auto simp add: rtranclp.rtrancl-into-rtrancl)

lemma eta-norm-imp-eta-reds: assumes eta-norm t = t' shows t →η* t'
using assms proof (induction t arbitrary: t' rule: eta-norm.induct)
  case (1 T body)
  then show ?case
  proof (cases eta-norm body)
    case (App f u)
    then show ?thesis
        using 1 apply (clarsimp simp add: is-dependent-def eta-reds-abs split:
term.splits nat.splits if-splits)
      by (metis eta.eta eta-reds-abs eta-reducible.simps(11) is-dependent-def
        not-eta-reducible-eta-norm not-eta-reducible-imp-eta-norm-no-change rtran-
clp.simps)
   qed (auto simp add: is-dependent-def eta-reds-abs split: term.splits nat.splits
if-splits)
next
  case (2 f u)
  hence f →η* eta-norm f u →η* eta-norm u
    by simp-all
  then show ?case using 2
    by (metis eta-norm.simps(2) eta-reds-appL eta-reds-appR rtranclp-trans)
qed auto

lemma rtrancl-eta-App:
    s →η* s' ⟹ t →η* t' ⟹ s $ t →η* s' $ t'
  by (blast intro!: eta-reds-appR eta-reds-appL intro: rtranclp-trans)

lemma eta-preserves-typ-of1: t →η t' ⟹ typ-of1 Ts t = Some τ ⟹ typ-of1 Ts
t' = Some τ
proof (induction Ts t arbitrary: τ t' rule: typ-of1.induct)
  case (1 uu uv T)
  then show ?case
    using eta-step-imp-eta-reducible by fastforce
next
  case (2 Ts i)
```

```
  then show ?case
    using eta-step-imp-eta-reducible by fastforce
next
  case (3 uw ux T)
  then show ?case
    using eta-step-imp-eta-reducible by fastforce
next
  case (4 Ts T body)
  then show ?case
  proof(cases body)
    case (Abs B b)
    then show ?thesis using 4
     by (metis eta-cases(1) term.distinct(19) typ-of1.simps(4) typ-of-Abs-body-typ')
  next
    case (App u v)
    note oApp = App
    then show ?thesis
    proof(cases is-dependent u)
      case True
      then show ?thesis
        by (metis 4.IH 4.prems(1) 4.prems(2) App eta-cases(1) term.inject(5)
            typ-of1.simps(4) typ-of-Abs-body-typ')
    next
      case False
      then show ?thesis
      proof(cases v)
        case (Ct n T)
        then show ?thesis
          using 4 oApp False typ-of-Abs-body-typ'
          by (metis eta-cases(1) term.distinct(3) term.inject(5) typ-of1.simps(4))
      next
        case (Fv n T)
        then show ?thesis
          using 4 oApp False typ-of-Abs-body-typ'
          by (metis eta-cases(1) term.distinct(9) term.inject(5) typ-of1.simps(4))
      next
        case (Bv n)
        then show ?thesis
        proof(cases n)
          case 0 thm 4
          show ?thesis
          proof(cases rule: eta-cases(1)[OF 4.prems(1)])
            case (1 s)
            thm 4(3)
            obtain rty where typ-of1 (T#Ts) (s $ Bv 0) = Some (rty)
              using typ-of-Abs-body-typ'[OF 4(3)] 1(3) 1(1) by blast
            moreover have τ = T → rty
             by (metis 1(1) 4.prems(2) calculation option.inject typ-of-Abs-body-typ')
            ultimately have typ-of1 (T#Ts) s = Some τ
```

```
            using typ-of1-arg-typ
            by (metis length-Cons nth-Cons-0 typ-of1.simps(2) zero-less-Suc)
          hence typ-of1 Ts (decr 0 s) = Some τ
            by (metis 1(3) append-Cons append-Nil is-dependent-def list.size(3)
typ-of1-decr)
          then show ?thesis
            using 1 oApp False typ-of-Abs-body-typ' Bv 0 by auto
        next
          case (2 t)
          then show ?thesis
            using oApp False typ-of-Abs-body-typ' Bv 0
            by (metis 4.IH 4.prems(2) typ-of1.simps(4))
        qed
      next
        case (Suc nat)
        then show ?thesis
          using 4 oApp False typ-of-Abs-body-typ' Bv
          apply −
          apply (rule eta-cases(1)[of T body t'])
          apply blast
          apply blast
          apply (metis 4.IH 4.prems(2) typ-of1.simps(4))
          done
      qed
    next
      case (Abs T t)
      then show ?thesis
        using 4 oApp False typ-of-Abs-body-typ'

        apply −
        apply (erule eta.cases(1))
       by (metis term.distinct(15) term.distinct(19) term.inject(4) term.inject(5)
           typ-of1.simps(4))+
    next
      case (App f u)
      then show ?thesis
        using 4 oApp False typ-of-Abs-body-typ'
        by (metis eta-cases(1) term.distinct(17) term.inject(5) typ-of1.simps(4))
    qed
  qed
 qed (use 4 in auto)
next
 case (5 Ts f u)
 then show ?case
  by (smt bind.bind-lunit eta-cases(2) typ-of1.simps(5) typ-of1-split-App-obtains)
qed

lemma eta-preserves-typ-of: t →η t' ⟹ typ-of t = Some τ ⟹ typ-of t' = Some
τ
```

using *eta-preserves-typ-of1 typ-of-def* **by** *simp*

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 (*induction rule*: *rtranclp.induct*)
case (*rtrancl-refl a*)
then show *?case*
by *simp*
next
case (*rtrancl-into-rtrancl a b c*)
then show *?case*
using *eta-preserves-typ-of1* **by** *blast*
qed

lemma *eta-star-preserves-typ-of*: $r \rightarrow_\eta^* s \Longrightarrow$ *typ-of r* = *Some T* $\Longrightarrow$ *typ-of s* = *Some T*
using *eta-star-preserves-typ-of1 typ-of-def* **by** *simp*

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*)
by (*induction k t arbitrary*: *us rule*: *decr.induct*) (*auto simp add*: *is-open-def*)

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*)
by (*simp add*: *is-dependent-def subst-bvs1'-decr subst-bvs-subst-bvs1'*)

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*)
by (*induction i t rule*: *decr.induct*) *auto*

lemma *eta-preserves-term-ok'*: *term-ok'* Σ *r* $\Longrightarrow$ *r* $\rightarrow_\eta$ *s* $\Longrightarrow$ *term-ok'* Σ *s*
proof (*induction r arbitrary*: *s*)
case (*Ct n T*)
then show *?case*
apply (*simp add*: *tinstT-def split*: *option.splits*)

using *eta-reducible.simps*(*12*) *eta-step-imp-eta-reducible* **by** *blast*
next
case (*Fv n T*)
then show *?case*
using *eta.cases*

by *blast*
next
case (*Bv n*)
then show *?case*
by *auto*
next
case (*Abs R r*)
then show *?case*
using *eta.cases*
by (*fastforce simp add*: *term-ok′-decr*)
next
case (*App f u*)
then show *?case*
apply −
apply (*erule eta-cases*(*2*))
using *term-ok′.simps*(*4*) **by** *blast*+
qed

lemma *eta-preserves-term-ok*: *term-ok* Θ *r* ⟹ *r* →$_\eta$ *s* ⟹ *term-ok* Θ *s*
proof −
assume *a1*: *term-ok* Θ *r*
assume *a2*: *r* →$_\eta$ *s*
then have *None* ≠ *typ-of1* [] *s*
using *a1 eta-preserves-typ-of1 option.collapse wt-term-def typ-of-def*
by *auto*
then show *?thesis*
using *a2 a1 eta-preserves-term-ok′ wt-term-def typ-of-def wf-term-iff-term-ok′ term-ok-def*
by (*meson eta-preserves-typ-of has-typ-iff-typ-of*)
qed

lemma *eta-star-preserves-term-ok′*: *r* →$_\eta$* *s* ⟹ *term-ok′* Σ *r* ⟹ *term-ok′* Σ *s*
by (*induction rule*: *rtranclp.induct*) (*auto simp add*: *eta-preserves-term-ok′*)

corollary *eta-star-preserves-term-ok*: *r* →$_\eta$* *s* ⟹ *term-ok thy r* ⟹ *term-ok thy s*
using *eta-star-preserves-term-ok′ eta-star-preserves-typ-of1 wt-term-def typ-of-def*
by *auto*

corollary *term-ok-eta-norm*: *term-ok thy t* ⟹ *eta-norm t* = *t′* ⟹ *term-ok thy t′*
using *eta-norm-imp-eta-reds eta-star-preserves-term-ok* **by** *blast*

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* ⟹ *wf-inst* Θ (λ*idn S* .*the-default* (*Tv idn S*) (*lookup* (λ*x*. *x*=(*idn*, *S*)) *insts*))
 by (*induction insts*) (*auto split*: *if-splits prod.splits*)

lemma *term-ok′-eta-norm*: *term-ok′* Σ *t* ⟹ *term-ok′* Σ (*eta-norm t*)
 by (*induction t rule*: *eta-norm.induct*)
  (*auto split*: *term.splits nat.splits simp add*: *term-ok′-decr is-dependent-def*)
corollary *term-ok-eta-norm*: *term-ok thy t* ⟹ *term-ok thy* (*eta-norm t*)
 using *wt-term-def typ-of-eta-norm term-ok′-eta-norm* **by** *auto*

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

lemma *beta-eta-norm t* = *Some t′* ⟹ ¬ *eta-reducible t′*
 using *not-eta-reducible-eta-norm* **by** *auto*
lemma *term-ok-beta-eta-norm*: *term-ok thy t* ⟹ *beta-eta-norm t* = *Some t′* ⟹ *term-ok thy t′*
 using *term-ok-eta-norm term-ok-beta-norm* **by** *blast*
lemma *typ-of-beta-eta-norm*:
 typ-of t = *Some T* ⟹ *beta-eta-norm t* = *Some t′* ⟹ *typ-of t′* = *Some T*
 using *beta-norm-imp-beta-reds beta-star-preserves-typ-of1 typ-of1-eta-norm typ-of-def*
by *fastforce*

lemma *inst-ok-nil*[*simp*]: *inst-ok* Θ [] **by** *simp*

lemma *axiom-subst-typ′*:
 assumes *wf-theory* Θ *A*∈*axioms* Θ *inst-ok* Θ *insts*
 shows Θ, Γ ⊢ *subst-typ′ insts A*
proof−
 have *wf-inst* Θ (λ*idn S* . *the-default* (*Tv idn S*) (*lookup* (λ*x*. *x*=(*idn*, *S*)) *insts*))
  using *inst-ok-imp-wf-inst assms*(*3*) **by** *blast*
 moreover have *subst-typ′ insts A*
  = *tsubst A* (λ*idn S* . *the-default* (*Tv idn S*) (*lookup* (λ*x*. *x*=(*idn*, *S*)) *insts*))
  by (*simp add*: *tsubst-simulates-subst-typ′*)
 ultimately show *?thesis*
  using *assms axiom* **by** *simp*
qed

corollary *axiom′*: *wf-theory* Θ ⟹ *A* ∈ *axioms* Θ ⟹ Θ, Γ ⊢ *A*
 apply (*subst subst-typ′-nil*[*symmetric*])

using *axiom-subst-typ′ inst-ok-nil* **by** *metis*

lemma *has-sort-Tv-refl*: *wf-osig oss* $\Longrightarrow$ *sort-ex* (*subclass oss*) *S* $\Longrightarrow$ *has-sort oss* (*Tv v S*) *S*
by (*cases oss*) (*simp add*: *osig-subclass-loc wf-subclass-loc.intro has-sort-Tv wf-subclass-loc.sort-leq-refl*)

lemma *has-sort-Tv-refl′*:
wf-theory Θ $\Longrightarrow$ *typ-ok* Θ (*Tv v S*) $\Longrightarrow$ *has-sort* (*osig* (*sig* Θ)) (*Tv v S*) *S*
using *has-sort-Tv-refl*
by (*metis wf-sig.simps osig.elims wf-theory-imp-wf-sig typ-ok-def*
wf-type-imp-typ-ok-sig typ-ok-sig.simps(*2*) *wf-sort-def*)

lemma *wf-inst-imp-inst-ok*:
wf-theory Θ $\Longrightarrow$ *distinct l* $\Longrightarrow$ $\forall$ (*v*, *S*) $\in$ *set l* . *typ-ok* Θ (*Tv v S*) $\Longrightarrow$ *wf-inst* Θ ϱ
$\Longrightarrow$ *inst-ok* Θ (*map* (λ(*v*, *S*) . ((*v*, *S*), ϱ *v S*)) *l*)
proof (*induction l*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons a l*)
have *I*: *inst-ok* Θ (*map* (λ(*v*,*S*) . ((*v*, *S*), ϱ *v S*)) *l*)
using *Cons* **by** *fastforce*

have *a* $\notin$ *set l*
using *Cons.prems*(*2*) **by** *auto*
hence (*a*, *case-prod* ϱ *a*) $\notin$ *set* (*map* (λ(*v*,*S*) . ((*v*, *S*), ϱ *v S*)) *l*)
by (*simp add*: *image-iff prod.case-eq-if*)
moreover have *distinct* (*map* (λ(*v*,*S*) . ((*v*, *S*), ϱ *v S*)) *l*)
using *I distinct-kv-list distinct-map* **by** *fast*
ultimately have *distinct* (*map* (λ(*v*,*S*) . ((*v*, *S*), ϱ *v S*)) (*a*#*l*))
by (*auto split*: *prod.splits*)

moreover have *wf-type* (*sig* Θ) (*case-prod* ϱ *a*)
using *Cons.prems*(*3*−*4*) **by** *auto* (*metis typ-ok-Tv wf-type-imp-typ-ok-sig*)
moreover hence *typ-ok* Θ (*case-prod* ϱ *a*)
by *simp*
moreover hence *has-sort* (*osig* (*sig* Θ)) (*case-prod* ϱ *a*) (*snd a*)
using *Cons.prems* **by** (*metis* (*full-types*) *has-sort-Tv-refl′ prod.case-eq-if wf-inst-def*)

ultimately show *?case*
using *I* **by** (*auto simp del*: *typ-ok-def split*: *prod.splits*)
qed

lemma *typs-of-fv-subset-Types*: *snd* ‘ *fv t* ⊆ *Types t*
by (*induction t*) *auto*
lemma *osig-tvsT-subset-SortsT*: *snd* ‘ *tvsT T* ⊆ *SortsT T*
by (*induction T*) *auto*

lemma *osig-tvs-subset-Sorts*: *snd ‘ tvs t* $\subseteq$ *Sorts t*
by (*induction t*) (*use osig-tvsT-subset-SortsT* **in** ‹*auto simp add*: *image-subset-iff*›)

lemma *term-ok-Types-imp-typ-ok-pre*:
is-std-sig $\Sigma \Longrightarrow$ *term-ok'* Σ *t* $\Longrightarrow$ $\tau \in$ *Types t* $\Longrightarrow$ *typ-ok-sig* Σ τ
by (*induction t arbitrary*: τ) (*auto split*: *option.splits*)

lemma *term-ok-Types-typ-ok*: *wf-theory* $\Theta \Longrightarrow$ *term-ok* Θ *t* $\Longrightarrow$ $\tau \in$ *Types t* $\Longrightarrow$ *typ-ok* Θ τ
by (*cases* Θ *rule*: *theory-full-exhaust*) (*fastforce simp add*: *wt-term-def intro*: *term-ok-Types-imp-typ-ok-pre*)

lemma *term-ok-fv-imp-typ-ok-pre*:
is-std-sig $\Sigma \Longrightarrow$ *term-ok'* Σ *t* $\Longrightarrow$ $(x,\tau) \in$ *fv t* $\Longrightarrow$ *typ-ok-sig* Σ τ
using *typs-of-fv-subset-Types term-ok-Types-imp-typ-ok-pre*
by (*metis image-subset-iff snd-conv*)

lemma *term-ok-vars-typ-ok*: *wf-theory* $\Theta \Longrightarrow$ *term-ok* Θ *t* $\Longrightarrow$ $(x, \tau) \in$ *fv t* $\Longrightarrow$ *typ-ok* Θ τ
using *term-ok-Types-typ-ok typs-of-fv-subset-Types* **by** (*metis image-subset-iff snd-conv*)

lemma *typ-ok-TFreesT-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 (*induction T*)
case (*Ty n Ts*)
then show *?case* **by** (*induction Ts*) (*fastforce dest*: *split-list split*: *option.split-asm*)+
qed (*auto simp add*: *wf-sort-def*)

lemma *term-ok-TFrees-imp-sort-ok-pre*:
is-std-sig $\Sigma \Longrightarrow$ *term-ok'* Σ *t* $\Longrightarrow$ $(x, S) \in$ *tvs t* $\Longrightarrow$ *wf-sort* (*subclass* (*osig* Σ)) *S*
proof (*induction t arbitrary*: *S*)
case (*Ct n T*)
then show *?case*
apply (*clarsimp split*: *option.splits*)
by (*use typ-ok-TFreesT-imp-sort-ok-pre wf-sort-def* **in** *auto*)
next
case (*Fv n T*)
then show *?case*
apply (*clarsimp split*: *option.splits*)
by (*use typ-ok-TFreesT-imp-sort-ok-pre wf-sort-def* **in** *auto*)
next
case (*Bv n*)
then show *?case*
by (*clarsimp split*: *option.splits*)
next
case (*Abs T t*)
then show *?case*

apply *simp*
using *typ-ok-TFreesT-imp-sort-ok-pre wf-sort-def*
by *meson*
next
case (*App t1 t2*)
then show *?case*
by *auto*
qed

lemma *typ-ok-tvsT-imp-sort-ok-pre*:
is-std-sig Σ ⟹ *typ-ok-sig* Σ *T* ⟹ (*x*,*S*) ∈ *tvsT T* ⟹ *wf-sort* (*subclass* (*osig* Σ)) *S*
proof (*induction T*)
case (*Ty n Ts*)
then show *?case* **by** (*induction Ts*) (*fastforce dest*: *split-list split*: *option.split-asm*)+
qed (*auto simp add*: *wf-sort-def*)

lemma *term-ok-tvars-sort-ok*:
assumes *wf-theory* Θ *term-ok* Θ *t* (*x*, *S*) ∈ *tvs t*
shows *wf-sort* (*subclass* (*osig* (*sig* Θ))) *S*
proof−
have *term-ok'* (*sig* Θ) *t*
using *assms*(*2*) **by** (*simp add*: *wt-term-def*)
moreover have *is-std-sig* (*sig* Θ)
using *assms* **by** (*cases* Θ *rule*: *theory-full-exhaust*) *simp*
ultimately show *?thesis*
using *assms*(*3*) *term-ok-TFrees-imp-sort-ok-pre* **by** *simp*
qed

lemma *term-ok'-bind-fv2*:
assumes *term-ok'* Σ *t*
shows *term-ok'* Σ (*bind-fv2* (*v*,*T*) *lev t*)
using *assms* **by** (*induction* (*v*,*T*) *lev t rule*: *bind-fv2.induct*) *auto*

lemma *term-ok'-bind-fv*:
assumes *term-ok'* Σ *t*
shows *term-ok'* Σ (*bind-fv* (*v*,τ) *t*)
using *term-ok'-bind-fv2 bind-fv-def assms* **by** *metis*

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*))
using *term-ok'-bind-fv assms* **by** *simp*

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*)
using *assms term-ok'-bind-fv*

by (*cases* Θ *rule*: *wf-theory.cases*) (*auto simp add*: *typ-of-def tinstT-def*)

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*)
using *typ-of-mk-all term-ok′-mk-all assms* **by** (*auto simp add*: *wt-term-def*)

lemma *term-ok′-incr-boundvars*:
term-ok′ (*sig* Θ) *t* ⟹ *term-ok′* (*sig* Θ) (*incr-boundvars lev t*)
using *term-ok′-incr-bv incr-boundvars-def* **by** *simp*

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*)
using *assms* **by** (*induction f lev u rule*: *subst-bv1.induct*) (*use term-ok′-incr-boundvars* **in** *auto*)

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*)
using *assms term-ok′-subst-bv1 subst-bv-def* **by** *simp*

lemma *term-ok′-betapply*:
assumes *term-ok′* (*sig* Θ) *f term-ok′* (*sig* Θ) *u*
shows *term-ok′* (*sig* Θ) (*f* ⋅ *u*)
proof(*cases f*)
case (*Abs T t*)
then show *?thesis*
using *assms term-ok′-subst-bv1* **by** (*simp add*: *subst-bv-def*)
qed (*use assms* **in** *auto*)

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*)
using *assms term-ok′-betapply wt-term-def typ-of-betaply assms* **by** *auto*

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*)
using *assms* **proof** (*induction insts ty rule*: *subst-typ.induct*)
case (*1 inst a Ts*)
have *typ-ok-sig* Σ (*subst-typ inst ty*) **if** *ty* ∈ *set Ts* **for** *ty*
using *that 1* **by** (*auto simp add*: *list-all-iff split*: *option.splits*)

hence ∀ *ty* ∈ *set* (*map* (*subst-typ inst*) *Ts*) . *typ-ok-sig* Σ *ty*
by *simp*

hence *list-all* (*typ-ok-sig* Σ) (*map* (*subst-typ inst*) *Ts*)
using *list-all-iff* **by** *blast*
moreover have *length* (*map* (*subst-typ inst*) *Ts*) = *length Ts* **by** *simp*
ultimately show *?case* **using** *1.prems* **by** (*auto split*: *option.splits*)
next
case (*2 inst idn S*)
then show *?case*
proof(*cases lookup* (λ*x*. *x* = (*idn*, *S*)) *inst* ≠ *None*)
case *True*
from *this 2* **obtain** *res* **where** *res*: *lookup* (λ*x*. *x* = (*idn*, *S*)) *inst* = *Some res*
by *auto*
have *res* ∈ *set* (*map snd inst*) **using** *2 res* **by** (*induction inst*) (*auto split*: *if-splits*)
hence *typ-ok-sig* Σ *res* **using** *2*(*4*) *res*
by (*induction inst*) (*auto split*: *if-splits simp add*: *rev-image-eqI*)
then show *?thesis* **using** *res* **by** *simp*
next
case *False*
hence *rewr*: *subst-typ inst* (*Tv idn S*) = *Tv idn S* **by** *auto*
then show *?thesis* **using** *2.prems*(*2*) **by** *simp*
qed
qed

corollary *subst-typ-tinstT*: *tinstT* (*subst-typ insts ty*) *ty*
unfolding *tinstT-def* **using** *tsubstT-simulates-subst-typ* **by** *fastforce*

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*
unfolding *tinstT-def* **proof** (*induction ty arbitrary*: *ty1 ty2*)
case (*Tv idx s*)
then show *?case* **by** (*cases ϱ1 idx s*) *auto*
qed *auto*

corollary *tinstT-trans*: *tinstT ty1 ty* ⟹ *tinstT ty2 ty1* ⟹ *tinstT ty2 ty*
unfolding *tinstT-def* **using** *tsubstT-trans* **by** *blast*

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*)
using *assms* **by** (*induction t*)
(*use typ-ok-sig-subst-typ subst-typ-tinstT tinstT-trans* **in** ‹*auto split*: *option.splits*›)

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

by (*induction t*) *auto*

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 (*induction Ts t arbitrary*: *ty rule*: *typ-of1.induct*)
case (*2 Ts i*)
then show *?case* **by** (*auto split*: *if-splits*)
next
case (*4 Ts T body*)
then show *?case* **by** (*auto simp add*: *bind-eq-Some-conv*)
next
case (*5 Ts f u*)
from *5.prems* **obtain** *u-ty* **where** *u-ty*: *typ-of1 Ts u* = *Some u-ty* **by** (*auto simp add*: *bind-eq-Some-conv*)
from *this 5.prems* **have** *f-ty*: *typ-of1 Ts f* = *Some* (*u-ty* → *ty*)
by (*auto simp add*: *bind-eq-Some-conv typ-of1-arg-typ*[*OF 5.prems*(*1*)]
split: *if-splits typ.splits option.splits*)

from *u-ty 5.IH*(*1*) **have** *typ-of1* (*map* (λ*T*. *tsubstT T* ϱ) *Ts*) (*tsubst u* ϱ) = *Some* (*tsubstT u-ty* ϱ)
by *simp*
moreover from *u-ty f-ty 5.IH*(*2*) **have** *typ-of1* (*map* (λ*T*. *tsubstT T* ϱ) *Ts*) (*tsubst f* ϱ)
= *Some* (*tsubstT* (*u-ty* → *ty*) ϱ)
by *simp*
ultimately show *?case* **by** *simp*
qed *auto*

corollary *typ-of1-tsubst-weak*:
assumes *typ-of1 Ts t* = *Some ty*
assumes *typ-of1* (*map* (λ*T* . *tsubstT T* ϱ) *Ts*) (*tsubst t* ϱ) = *Some ty*′
shows *tsubstT ty* ϱ = *ty*′
using *assms typ-of1-tsubst* **by** *auto*

lemma *tsubstT-no-change*[*simp*]: *tsubstT T Tv* = *T*
by (*induction T*) (*auto simp add*: *map-idI*)

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*
using *assms* **by** (*cases* Θ *rule*: *theory-full-exhaust*)
(*fastforce simp add*: *wt-term-def typ-of-def bind-eq-Some-conv tinstT-def*)

lemma *typ-of-eta-expand*: *typ-of f* = *Some* (τ → τ′) ⟹ *typ-of* (*Abs* τ (*f* $ *Bv 0*)) = *Some* (τ → τ′)
using *typ-of1-weaken* **by** (*fastforce simp add*: *bind-eq-Some-conv typ-of-def*)

lemma *term-okI*: *term-ok′* (*sig* Θ) *t* ⟹ *typ-of t* ≠ *None* ⟹ *term-ok* Θ *t*
by (*simp add*: *wt-term-def*)
lemma *term-okD1*: *term-ok* Θ *t* ⟹ *term-ok′* (*sig* Θ) *t*
by (*simp add*: *wt-term-def*)
lemma *term-okD2*: *term-ok* Θ *t* ⟹ *typ-of t* ≠ *None*
by (*simp add*: *wt-term-def*)

lemma *term-ok-imp-typ-ok′*: **assumes** *wf-theory* Θ *term-ok* Θ *t* **shows** *typ-ok* Θ (*the* (*typ-of t*))
proof−
obtain *ty* **where** *ty*: *typ-of t* = *Some ty*
by (*meson assms option.exhaust term-okD2*)
hence *typ-ok* Θ *ty*
using *term-ok-imp-typ-ok assms* **by** *blast*
thus *?thesis* **using** *ty* **by** *simp*
qed

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 (*rule term-okI*)
have *typ-ok* Θ (*the* (*typ-of t*))
using *assms*(*1*) *assms*(*3*) *term-ok-imp-typ-ok′* **by** *blast*
hence *typ-ok-sig* (*sig* Θ) (*the* (*typ-of t*))
by *simp*
then show *term-ok′* (*sig* Θ) (*mk-eq s t*)
using *assms* **apply** −
apply (*drule term-okD1*)+
apply (*cases* Θ *rule*: *theory-full-exhaust*)
by (*auto split*: *option.splits simp add*: *tinstT-def*)
next
show *typ-of* (*mk-eq s t*) ≠ *None*
using *assms typ-of-def* **by** (*auto dest*: *term-okD2 simp add*: *wt-term-def*)
qed

lemma *typ-of1-decr′*: ¬ *loose-bvar1 t 0* ⟹ *typ-of1* (*T*#*Ts*) *t* = *Some* τ ⟹ *typ-of1 Ts* (*decr 0 t*) = *Some* τ
proof (*induction Ts t arbitrary*: *T* τ *rule*: *typ-of1.induct*)
case (*4 Ts B body*)
then show *?case*
using *typ-of1-decr-gen*
apply (*simp add*: *bind-eq-Some-conv split*: *if-splits option.splits*)
by (*metis append-Cons append-Nil length-Cons list.size*(*3*) *typ-of1-decr-gen*)
next
case (*5 Ts f u*)
then show *?case* **apply** (*simp add*: *bind-eq-Some-conv split*: *if-splits option.splits*)
by (*smt no-loose-bvar1-subst-bv2-decr subst-bv-def substn-subst-0′ typ-of1.simps*(*3*) *typ-of1-subst-bv-gen′*)
qed (*auto simp add*: *bind-eq-Some-conv split*: *if-splits option.splits*)

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'*)
using *typ-of1-decr'*
by (*smt length-Cons nth-Cons-0 typ-of1.simps*(*2*) *typ-of1-arg-typ typ-of-Abs-body-typ' zero-less-Suc*)

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'*)
using *typ-of-def is-dependent-def typ-of1-eta-red-step-pre* **by** *simp*

lemma *distinct-add-vars'*: *distinct acc* ⟹ *distinct* (*add-vars' t acc*)
unfolding *add-vars'-def*
by (*induction t arbitrary*: *acc*) *auto*

lemma *distinct-add-tvarsT'*: *distinct acc* ⟹ *distinct* (*add-tvarsT' T acc*)
proof (*induction T arbitrary*: *acc*)
case (*Ty n Ts*)
then show *?case*
by (*induction Ts rule*: *rev-induct*) (*auto simp add*: *add-tvarsT'-def*)
qed (*simp add*: *add-tvarsT'-def*)

lemma *distinct-add-tvars'*: *distinct acc* ⟹ *distinct* (*add-tvars' t acc*)
by (*induction t arbitrary*: *acc*) (*simp-all add*: *add-tvars'-def fold-types-def distinct-add-tvarsT'*)

lemma *proved-terms-well-formed-pre*: Θ, Γ ⊢ *p* ⟹ *typ-of p* = *Some propT* ∧ *term-ok* Θ *p*
proof (*induction* Γ *p rule*: *proves.induct*)
case (*axiom A ϱ*)

from *axiom* **have** *ty*: *typ-of1* [] *A* = *Some propT*
by (*cases* Θ *rule*: *theory-full-exhaust*) (*simp add*: *wt-term-def typ-of-def*)
let *?l* = *add-tvars' A* []
let *?l'* = *map* (λ(*v*, *S*) . ((*v*, *S*), *ϱ v S*)) *?l*
have *dist*: *distinct ?l*
using *distinct-add-tvars'* **by** *simp*
moreover have ∀(*v*, *S*) ∈ *set ?l* . *typ-ok* Θ (*Tv v S*)
proof−
have *typ-ok* Θ (*Tv v T*) **if** (*v*, *T*) ∈ *tvs A* **for** *v T*
using *axiom.hyps*(*1*) *axiom.hyps*(*2*) *axioms-terms-ok*
term-ok-tvars-sort-ok that typ-ok-def typ-ok-Tv
by (*meson wf-sort-def*)
moreover have *set ?l* = *tvs A*
by *auto*

```
    ultimately show ?thesis
      by auto
  qed
  moreover hence ∀(v, S) ∈ set ?l . has-sort (osig (sig Θ)) (Tv v S) S
    using axiom.hyps(1) has-sort-Tv-refl' by blast

  ultimately have inst-ok Θ ?l'
    apply − apply (rule wf-inst-imp-inst-ok)
    using axiom.hyps(1) axiom.hyps(3) by blast+

  have simp: tsubst A ϱ = subst-typ' ?l' A
    using dist subst-typ'-simulates-tsubst-gen' by auto

  have typ-of1 [] (tsubst A ϱ) = Some propT
    using tsubst-simulates-subst-typ' axioms-typ-of-propT typ-of1-tsubst ty by fast-
force
  hence 1: typ-of1 [] (subst-typ' ?l' A) = Some propT
    using simp by simp

  from axiom have term-ok' (sig Θ) A
    by (cases Θ rule: theory-full-exhaust) (simp add: wt-term-def)
  hence 2: term-ok' (sig Θ) (subst-typ' ?l' A)
    using axiom term-ok'-subst-typ' apply (cases Θ rule: theory-full-exhaust)
    apply (simp add: list-all-iff wt-term-def typ-of-def)
   by (metis (no-types, lifting) ‹inst-ok Θ (map (λ(v, S). ((v, S), ϱ v S)) (add-tvars'
A []))›
      axiom.hyps(1) list.pred-mono-strong sig.simps term-ok'-subst-typ' wf-theory.simps
        typ-ok-def wf-type-imp-typ-ok-sig)
  from 1 2 show ?case using simp by (simp add: wt-term-def typ-of-def)
next
  case (assume A)
  then show ?case by (simp add: wt-term-def)
next

  case (forall-intro Γ B x τ)
  hence term-ok' (sig Θ) B and typ-of B = Some propT
    by (simp-all add: wt-term-def)
  show ?case using typ-of-mk-all forall-intro
      term-ok-mk-all[OF ‹wf-theory Θ› ‹term-ok' (sig Θ) B›
        ‹typ-of B = Some propT› -, of - x] ‹wf-type (sig Θ) τ›
    by auto
next
  case (forall-elim Γ τ B a)
  thus ?case using term-ok'-subst-bv1
    by (auto simp add: typ-of-def term-ok'-subst-bv tinstT-def
      wt-term-def bind-eq-Some-conv subst-bv-def typ-of1-subst-bv-gen'
      split: if-splits option.splits)
next
  case (implies-intro Γ B A)
```

then show *?case*
by (*cases* Θ *rule*: *wf-theory.cases*) (*auto simp add*: *typ-of-def wt-term-def tinstT-def*)
next
case (*implies-elim* Γ_1 *A B* Γ_2)

then show *?case*
by (*auto simp add*: *bind-eq-Some-conv typ-of-def wt-term-def tinstT-def*
split: *option.splits if-splits*)
next
case (*of-class c iT T*)

then show *?case*
by (*cases* Θ *rule*: *theory-full-exhaust*)
(*auto simp add*: *bind-eq-Some-conv typ-of-def wt-term-def*
tinstT-def mk-of-class-def mk-type-def)
next
case (*β-conversion T t x*)
hence *1*: *typ-of* (*mk-eq* (*Abs T t* $ *x*) (*subst-bv x t*)) = *Some propT*
by (*auto simp add*: *typ-of-def wt-term-def subst-bv-def bind-eq-Some-conv*
typ-of1-subst-bv-gen')
moreover have *term-ok* Θ (*mk-eq* (*Abs T t* $ *x*) (*subst-bv x t*))
proof−
have *typ-of* (*mk-eq* (*Abs T t* $ *x*) (*subst-bv x t*)) ≠ *None*
using *1* **by** *simp*

moreover have *term-ok'* (*sig* Θ) (*mk-eq* (*Abs T t* $ *x*) (*subst-bv x t*))
proof−
have *term-ok'* (*sig* Θ) (*Abs T t* $ *x*)
using *β-conversion.hyps*(*2*) *β-conversion.hyps*(*3*) *term-ok'.simps*(*4*) *wt-term-def term-ok-def* **by** *blast*
moreover hence *term-ok'* (*sig* Θ) (*subst-bv x t*)
using *subst-bv-def term-ok'-subst-bv1* **by** *auto*
moreover have *const-type* (*sig* Θ) *STR ''Pure.eq''*
= *Some* ((*Tv* (*Var* (*STR '''a''*, *0*)) *full-sort*) → ((*Tv* (*Var* (*STR '''a''*, *0*)) *full-sort*) → *propT*))
using *β-conversion.hyps*(*1*) **by** (*cases* Θ) *fastforce*
moreover obtain *t'* **where** *typ-of* (*Abs T t* $ *x*) = *Some t'*
by (*smt 1 typ-of1-split-App typ-of-def*)
moreover hence *typ-of* (*subst-bv x t*) = *Some t'*
by (*smt list.simps*(*1*) *subst-bv-def typ.simps*(*1*) *typ-of1-split-App typ-of1-subst-bv-gen' typ-of-Abs-body-typ' typ-of-def*)
moreover have *typ-ok-sig* (*sig* Θ) *t'*
using *β-conversion.hyps*(*1*) *calculation*(*2*) *calculation*(*5*) *wt-term-def term-ok-imp-typ-ok typ-ok-def* **by** *auto*
moreover hence *typ-ok-sig* (*sig* Θ) (*t'* → *propT*)
using ‹*wf-theory* Θ› **by** (*cases* Θ *rule*: *theory-full-exhaust*) *auto*
moreover have *tinstT* (*T* → (*T* → *propT*)) ((*Tv* (*Var* (*STR '''a''*, *0*)) *full-sort*) → ((*Tv* (*Var* (*STR '''a''*, *0*)) *full-sort*) → *propT*))

unfolding *tinstT-def* **by** *auto*
moreover have *tinstT* $(t' \to (t' \to propT))$ $((Tv\ (Var\ (STR\ '''a'',\ 0))\ full\text{-}sort) \to ((Tv\ (Var\ (STR\ '''a'',\ 0))\ full\text{-}sort) \to propT))$
unfolding *tinstT-def* **by** *auto*
ultimately show *?thesis* **using** ‹*wf-theory* Θ› **by** (*cases* Θ *rule*: *theory-full-exhaust*) *auto*
qed
ultimately show *?thesis* **using** *wt-term-def* **by** *simp*
qed
ultimately show *?case* **by** *simp*
next
case (*eta* $t\ \tau\ \tau'$)
hence *tyeta*: *typ-of* $(Abs\ \tau\ (t\ \$\ Bv\ 0)) = Some\ (\tau \to \tau')$
using *typ-of-eta-expand* **by** *auto*
moreover have $\neg$ *is-dependent t*
proof−
have *is-closed t*
using *eta.hyps*(*3*) *typ-of-imp-closed* **by** *blast*
thus *?thesis*
using *is-dependent-def is-open-def loose-bvar1-imp-loose-bvar* **by** *blast*
qed
ultimately have *ty-decr*: *typ-of* $(decr\ 0\ t) = Some\ (\tau \to \tau')$
using *typ-of1-eta-red-step* **by** *blast*

hence *1*: *typ-of* $(mk\text{-}eq\ (Abs\ \tau\ (t\ \$\ Bv\ 0))\ (decr\ 0\ t)) = Some\ propT$
using *eta tyeta* **by** (*auto simp add*: *typ-of-def*)

have *typ-ok* $\Theta\ (\tau \to \tau')$
using *eta term-ok-imp-typ-ok* **by** (*simp add*: *wt-term-def del*: *typ-ok-def*)
hence *tyok*: *typ-ok* $\Theta\ \tau$ *typ-ok* $\Theta\ \tau'$
unfolding *typ-ok-def* **by** (*auto split*: *option.splits*)
hence *term-ok* $\Theta\ (Abs\ \tau\ (t\ \$\ Bv\ 0))$
using *eta*(*2*) *tyeta* **by** (*simp add*: *wt-term-def*)
moreover have *term-ok* $\Theta\ (decr\ 0\ t)$
using *eta term-ok'-decr tyeta ty-decr wt-term-def typ-ok-def tyok*
by (*cases* Θ *rule*: *theory-full-exhaust*) (*auto split*: *option.splits simp add*: *tinstT-def*)
ultimately have *term-ok* $\Theta\ (mk\text{-}eq\ (Abs\ \tau\ (t\ \$\ Bv\ 0))\ (decr\ 0\ t))$
using *eta.hyps ty-decr tyeta tyok 1 term-ok-mk-eqI*
by *metis*
then show *?case* **using** *1*
using *eta.hyps*(*2*) *eta.hyps*(*3*) *has-typ-imp-closed term-ok-subst-bv-no-change closed-subst-bv-no-change* **by** *auto*
qed

corollary *proved-terms-well-formed*:
assumes $\Theta, \Gamma \vdash p$
shows *typ-of p* = *Some propT term-ok* Θ *p*
using *assms proved-terms-well-formed-pre* **by** *auto*

lemma *forall-intros*:
wf-theory $\Theta \Longrightarrow \Theta,\Gamma \vdash B \Longrightarrow \forall (x, \tau) \in$ *set frees* . $(x,\tau) \notin FV\ \Gamma \wedge$ *typ-ok* $\Theta\ \tau$
$\Longrightarrow \Theta,\Gamma \vdash$ *mk-all-list frees B*
by (*induction frees arbitrary*: *B*)
(*auto intro*: *proves.forall-intro simp add*: *mk-all-list-def simp del*: *FV-def split*: *prod.splits*)

lemma *term-ok-var*[*simp*]: *term-ok* Θ (*Fv idn* τ) = *typ-ok* $\Theta\ \tau$
by (*simp add*: *wt-term-def typ-of-def*)
lemma *typ-of-var*[*simp*]: *typ-of* (*Fv idn* τ) = *Some* τ
by (*simp add*: *typ-of-def*)

lemma *is-closed-Fv*[*simp*]: *is-closed* (*Fv idn* τ) **by** (*simp add*: *is-open-def*)

corollary *proved-terms-closed*: Θ, $\Gamma \vdash B \Longrightarrow$ *is-closed B*
by (*simp add*: *proved-terms-well-formed*(*1*) *typ-of-imp-closed*)

lemma *not-loose-bvar-bind-fv2*:
$\neg$ *loose-bvar t lev* $\Longrightarrow \neg$ *loose-bvar* (*bind-fv2 v lev t*) (*Suc lev*)
by (*induction t arbitrary*: *lev*) *auto*
lemma *not-loose-bvar-bind-fv2-*:
$\neg$ *loose-bvar* (*bind-fv2 v lev t*) *lev* $\Longrightarrow \neg$ *loose-bvar t lev*
by (*induction t arbitrary*: *lev*) (*auto split*: *if-splits*)

lemma *fold-add-vars′-FV-pre*: *set* (*fold add-vars′ Hs acc*) = *set acc* $\cup$ *FV* (*set Hs*)
by (*induction Hs arbitrary*: *acc*) (*auto simp add*: *add-vars′-fv-pre*)
corollary *fold-add-vars′-FV*[*simp*]: *set* (*fold* (*add-vars′*) *Hs* []) = *FV* (*set Hs*)
using *fold-add-vars′-FV-pre* **by** *simp*

lemma *forall-intro-vars*:
assumes *wf-theory* Θ Θ, *set Hs* $\vdash$ *B*
shows Θ, *set Hs* $\vdash$ *forall-intro-vars B Hs*
apply (*rule forall-intros*)
using *assms* **apply** *simp-all* **apply** *clarsimp*
using *add-vars′-fv proved-terms-well-formed-pre term-ok-vars-typ-ok*
by (*metis term-ok-vars-typ-ok typ-ok-def wf-type-imp-typ-ok-sig*)

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*
using *assms* **proof** (*induction vs rule*: *rev-induct*)
case *Nil*
then show *?case* **by** *simp*
next

case (*snoc v vs*)
hence *I*: *term-ok* Θ (*mk-all-list vs B*) *typ-of* (*mk-all-list vs B*) = *Some propT*
by *simp-all*
obtain *idn ty* **where** *v*: *v*=(*idn*,*ty*) **by** *fastforce*
hence *s*: (*mk-all-list* (*vs* @ [*v*]) *B*) = *mk-all idn ty* (*mk-all-list* (*vs*) *B*)
by (*simp add*: *mk-all-list-def*)
have *typ-ok* Θ *ty* **using** *v snoc.prems* **by** *simp*
then show *?case* **using** *I s term-ok-mk-all snoc.prems*(*1*) *wt-term-def typ-of-mk-all*
by *auto*
qed

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*) ∧ *typ-of* (*forall-intro-vars B Hs*) = *Some propT*
proof−
have *1*: ∀(*idn*,*ty*)∈*set* (*add-vars*′ *B* []) . *typ-ok* Θ *ty*
using *add-vars*′*-fv assms*(*1*) *assms*(*2*) *term-ok-vars-typ-ok* **by** *blast*
thus *?thesis* **using** *assms mk-all-list*′*-preserves-term-ok-typ-of* **by** *simp*
qed

lemma *bind-fv-remove-var-from-fv*: *fv* (*bind-fv* (*idn*, τ) *t*) = *fv t* − {(*idn*, τ)}
using *bind-fv2-Fv-fv bind-fv-def* **by** *simp*

lemma *forall-intro-vars-remove-fv*[*simp*]: *fv* (*forall-intro-vars t* []) = {}
using *mk-all-list-fv-unchanged add-vars*′*-fv* **by** *simp*

lemma *term-ok-mk-all-list*:
assumes *wf-theory* Θ
assumes *term-ok* Θ *B*
assumes *typ-of B* = *Some propT*
assumes ∀(*idn*, τ) ∈ *set l* . *typ-ok* Θ τ
shows *term-ok* Θ (*mk-all-list l B*) ∧ *typ-of* (*mk-all-list l B*) = *Some propT*
using *assms* **proof**(*induction l rule*: *rev-induct*)
case *Nil*
then show *?case* **by** *simp*
next
case (*snoc v vs*)
obtain *idn* τ **where** *v*: *v* = (*idn*, τ) **by** *fastforce*
hence *simp*: *mk-all-list* (*vs*@[*v*]) *B* = *mk-all idn* τ (*mk-all-list vs B*)
by (*auto simp add*: *mk-all-list-def*)
have *I*: *term-ok* Θ (*mk-all-list vs B*) *typ-of* (*mk-all-list vs B*) = *Some propT*
using *snoc* **by** *auto*
have *term-ok* Θ (*mk-all idn* τ (*mk-all-list vs B*))
using *term-ok-mk-all snoc.prems I v* **by** (*auto simp add*: *wt-term-def*)
moreover have *typ-of* (*mk-all idn* τ (*mk-all-list vs B*)) = *Some propT*
using *I*(*2*) *v typ-of-mk-all* **by** *simp*
ultimately show *?case* **by** (*simp add*: *simp*)

qed

lemma *tvs-bind-fv2*: *tvs* (*bind-fv2* (*v*, *T*) *lev t*) ∪ *tvsT T* = *tvs t* ∪ *tvsT T*
by (*induction* (*v*, *T*) *lev t rule*: *bind-fv2.induct*) *auto*
lemma *tvs-bind-fv*: *tvs* (*bind-fv* (*v*,*T*) *t*) ∪ *tvsT T* = *tvs t* ∪ *tvsT T*
using *tvs-bind-fv2 bind-fv-def* **by** *simp*

lemma *tvs-mk-all′*: *tvs* (*mk-all idn ty B*) = *tvs B* ∪ *tvsT ty*
using *tvs-bind-fv typ-of-def is-variable.simps*(*2*) **by** *fastforce*

lemma *tvs-mk-all-list*:
tvs (*mk-all-list vs B*) = *tvs B* ∪ *tvsT-Set* (*snd* ‘ *set vs*)
proof(*induction vs rule*: *rev-induct*)
case *Nil*
then show *?case* **by** *simp*
next
case (*snoc v vs*)
obtain *idn τ* **where** *v*: *v* = (*idn*, *τ*) **by** *fastforce*
show *?case* **using** *snoc v tvs-mk-all′* **by** (*auto simp add*: *mk-all-list-def*)
qed

lemma *tvs-occs*: *occs v t* ⟹ *tvs v* ⊆ *tvs t*
by (*induction t*) *auto*

lemma *tvs-forall-intro-vars*: *tvs* (*forall-intro-vars B Hs*) = *tvs B*
proof−
have ∀(*idn*, *ty*)∈*fv B* . *occs* (*Fv idn ty*) *B*
using *fv-occs* **by** *blast*
hence ∀(*idn*, *ty*)∈*fv B* . *tvs* (*Fv idn ty*) ⊆ *tvs B*
using *tvs-occs* **by** *blast*
hence ∀(*idn*, *ty*)∈*fv B* . *tvsT ty* ⊆ *tvs B*
by *simp*
hence *tvsT-Set* (*snd* ‘ *fv B*) ⊆ *tvs B*
by *fastforce*
hence *tvsT-Set* (*snd* ‘ *set* (*add-vars′ B* [])) ⊆ *tvs B*
by (*simp add*: *add-vars′-fv*)
thus *?thesis* **using** *tvs-mk-all-list* **by** *auto*
qed

lemma *strip-all-single-var B* = *Some τ* ⟹ *strip-all-single-body B* ≠ *B*
using *strip-all-vars-step* **by** *fastforce*

lemma *strip-all-body-unchanged-iff-strip-all-single-body-unchanged*:
strip-all-body B = *B* ⟷ *strip-all-single-body B* = *B*
by (*metis not-Cons-self2 not-None-eq not-is-all-imp-strip-all-body-unchanged*
strip-all-body-single-simp′ strip-all-single-var-is-all strip-all-vars-step)

lemma *strip-all-body-unchanged-imp-strip-all-vars-no*:

assumes *strip-all-body B = B*
shows *strip-all-vars B = []*
by (*smt assms not-Cons-self2 strip-all-body-single-simp' strip-all-single-body.simps(1) strip-all-vars.elims*)

lemma *strip-all-body-unchanged-imp-strip-all-single-body-unchanged*:
strip-all-body B = B ⟹ strip-all-single-body B = B
by (*smt* (*z3*) *not-Cons-self2 strip-all-body-single-simp' strip-all-single-body.simps(1) strip-all-vars.simps(1)*)

lemma *strip-all-single-body-unchanged-imp-strip-all-body-unchanged*:
strip-all-single-body B = B ⟹ strip-all-body B = B
by (*auto elim*!: *strip-all-single-body.elims*)

lemma *strip-all-single-var-np-imp-strip-all-body-single-unchanged*:
strip-all-single-var B = None ⟹ strip-all-single-body B = B
by (*auto elim*!: *strip-all-single-var.elims*)

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
by (*auto elim*!: *strip-all-single-var.elims split*: *if-splits*)

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−
have *1*: *Ct STR ''Pure.all'' ((τ → propT) → propT) $ Abs τ (strip-all-single-body B) = B*
using *assms(2) strip-all-single-form* **by** *blast*
hence *Θ, Γ ⊢ Abs τ (strip-all-single-body B) · t*
using *assms forall-elim*
proof −
have *has-typ t τ*
by (*meson ‹typ-of t = Some τ› has-typ-iff-typ-of*)
then show *?thesis*
by (*metis 1 assms(1) assms(4) betapply.simps(1) forall-elim term-ok-def wt-term-def*)
qed
thus *?thesis* **by** *simp*
qed

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 −
have *ok*: *term-ok Θ B*
using *assms(1) proved-terms-well-formed(2)* **by** *auto*

thm *strip-all-single-form*
wt-term-def term-ok-var typ-of-var typ-ok-def proves-strip-all-single
strip-all-single-form
have *s*: $B = Ct\ STR\ ''Pure.all''\ ((\tau \to propT) \to propT)\ \$\ Abs\ \tau\ (strip\text{-}all\text{-}single\text{-}body\ B)$
using *assms*(2) *strip-all-single-form*[*symmetric*] **by** *simp*
have $\tau \in Types\ B$
by (*subst s*, *simp*)
hence *typ-ok* $\Theta\ \tau$
by (*metis ok s term-ok'.simps*(4) *term-ok'.simps*(5) *term-okD1 typ-ok-def typ-ok-sig-imp-wf-type*)
hence *term-ok* $\Theta\ (Fv\ x\ \tau)$
using *term-ok-var* **by** *blast*
then show *?thesis*
using *assms proves-strip-all-single*[**where** $\tau{=}\tau$] **by** *auto*
qed

lemma *strip-all-vars-no-strip-all-body-unchanged*[*simp*]:
$strip\text{-}all\text{-}vars\ B = [] \Longrightarrow strip\text{-}all\text{-}body\ B = B$
by (*auto elim*!: *strip-all-vars.elims*)

lemma *strip-all-vars* $B = (\tau s @ [\tau]) \Longrightarrow strip\text{-}all\text{-}body\ B$
$= strip\text{-}all\text{-}single\text{-}body\ (Ct\ STR\ ''Pure.all''\ ((\tau \to propT) \to propT)\ \$\ Abs\ \tau\ (strip\text{-}all\text{-}body\ B))$
by *simp*

lemma *strip-all-vars-incr-bv*: $strip\text{-}all\text{-}vars\ (incr\text{-}bv\ inc\ lev\ t) = strip\text{-}all\text{-}vars\ t$
by (*induction t arbitrary*: *lev rule*: *strip-all-vars.induct*) *auto*
lemma *strip-all-vars-incr-boundvars*: $strip\text{-}all\text{-}vars\ (incr\text{-}boundvars\ inc\ t) = strip\text{-}all\text{-}vars\ t$
using *incr-boundvars-def strip-all-vars-incr-bv* **by** *simp*

lemma *strip-all-vars-subst-bv1-Fv*:
$strip\text{-}all\text{-}vars\ (subst\text{-}bv1\ B\ lev\ (Fv\ x\ \tau)) = strip\text{-}all\text{-}vars\ B$
by (*induction B arbitrary*: *lev rule*: *strip-all-vars.induct*) (*auto simp add*: *incr-boundvars-def*)
lemma *strip-all-vars-subst-bv-Fv*:
$strip\text{-}all\text{-}vars\ (subst\text{-}bv\ (Fv\ x\ \tau)\ B) = strip\text{-}all\text{-}vars\ B$
by (*simp add*: *strip-all-vars-subst-bv1-Fv subst-bv-def*)

lemma *strip-all-single-var* $B = Some\ \tau$
$\Longrightarrow strip\text{-}all\text{-}vars\ (subst\text{-}bv\ (Fv\ x\ \tau)\ (strip\text{-}all\text{-}single\text{-}body\ B)) = tl\ (strip\text{-}all\text{-}vars\ B)$
by (*metis list.sel*(3) *strip-all-vars-step strip-all-vars-subst-bv-Fv*)

corollary *proves-strip-all-vars-Fv*:
assumes $length\ xs = length\ (strip\text{-}all\text{-}vars\ B)\ \Theta, \Gamma \vdash B$
shows $\Theta, \Gamma \vdash fold\ (\lambda(x,\tau).\ subst\text{-}bv\ (Fv\ x\ \tau)\ o\ strip\text{-}all\text{-}single\text{-}body)$
$(zip\ xs\ (strip\text{-}all\text{-}vars\ B))\ B$

using *assms* **proof** (*induction xs strip-all-vars B arbitrary*: *B rule*: *list-induct2*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons x xs τ τs*)
have *st*: *strip-all-single-var B* = *Some τ*
by (*metis Cons.hyps*(*3*) *is-all-iff-strip-all-vars-not-empty list.distinct*(*1*) *list.inject*

option.exhaust strip-all-single-var-is-all strip-all-vars-step)
moreover have *term-ok* Θ (*Fv x τ*)
proof−
obtain *B′* **where** *Ct STR ''Pure.all''* ((*τ* → *propT*) → *propT*) $ *Abs τ B′* = *B*
using *st strip-all-single-form* **by** *blast*
moreover have *term-ok* Θ *B*
using *Cons.prems proved-terms-well-formed*(*2*) **by** *auto*
ultimately have *typ-ok* Θ *τ*
using *term-ok′.simps*(*5*) *term-ok′.simps*(*4*) *term-ok-def wt-term-def typ-ok-def*
by *blast*
thus *?thesis* **unfolding** *term-ok-def wt-term-def typ-ok-def* **by** *simp*
qed
ultimately have *1*: Θ,Γ ⊢ *subst-bv* (*Fv x τ*) (*strip-all-single-body B*)
using *proves-strip-all-single*
by (*simp add*: *Cons.prems proves-strip-all-single-Fv*)
have Θ,Γ ⊢ *fold* (λ(*x*, *τ*). *subst-bv* (*Fv x τ*) ∘ *strip-all-single-body*)
(*zip xs* (*strip-all-vars* (*subst-bv* (*Fv x τ*) (*strip-all-single-body B*))))
(*subst-bv* (*Fv x τ*) (*strip-all-single-body B*))
apply (*rule Cons.hyps*)
apply (*metis Cons.hyps*(*3*) *list.inject st strip-all-vars-step strip-all-vars-subst-bv-Fv*)
using *1* **by** *simp*
moreover have *strip-all-vars B* = *τ* # *τs*
using *Cons.hyps*(*3*) **by** *auto*
ultimately show *?case*
using *st strip-all-vars-step strip-all-vars-subst-bv-Fv* **by** *fastforce*
qed

lemma *trivial-pre-depr*: *term-ok* Θ *c* ⟹ *typ-of c* = *Some propT* ⟹ Θ, {*c*} ⊢ *c*
by (*rule assume*) (*simp-all add*: *wt-term-def*)

lemma *trivial-pre*:
assumes *wf-theory* Θ *term-ok* Θ *c typ-of c* = *Some propT*
shows Θ, {} ⊢ *c* ⟼ *c*
proof−
have *s*: {} = {*c*} − {*c*} **by** *simp*
show *?thesis*
apply (*subst s*)
apply (*rule implies-intro*)
using *assms* **by** (*auto simp add*: *wt-term-def intro*: *assume*)
qed

lemma *inst-var*:
assumes *wf-theory*: *wf-theory* Θ
assumes *B*: Θ, $\Gamma \vdash B$
assumes *a-ok*: *term-ok* Θ *a*
assumes *typ-a*: *typ-of a* = *Some* τ
assumes *free*: $(x,\tau) \notin FV\ \Gamma$
shows Θ, $\Gamma \vdash$ *subst-term* $[((x,\ \tau),\ a)]\ B$
proof−
have *s1*: *mk-all x* τ *B* = *Ct STR ''Pure.all''* $((\tau \to propT) \to propT)$ $
Abs τ (*bind-fv* $(x,\ \tau)$ *B*)
by (*simp add*: *typ-of-def*)
have *closed-B*: *is-closed B* **using** *B proved-terms-well-formed-pre*
using *typ-of-imp-closed* **by** *blast*
have *typ-ok* Θ τ **using** *wt-term-def typ-ok-def term-ok-imp-typ-ok*
using *a-ok wf-theory typ-a* **by** *blast*
hence *p1*: Θ, $\Gamma \vdash$ *mk-all x* τ *B*
using *forall-intro*[*OF wf-theory B*] *B typ-a wt-term-def wf-theory*
term-ok-imp-typ-ok free **by** *simp*
have Θ, $\Gamma \vdash$ *subst-bv a* (*bind-fv* $(x,\ \tau)$ *B*)
using *forall-elim*[*of - - τ*] *p1 typ-a a-ok proves-strip-all-single*
by (*meson has-typ-iff-typ-of term-ok-def wt-term-def*)
have Θ, $\Gamma \vdash$ *subst-bv a* ((*bind-fv* $(x,\ \tau)$ *B*))
using *forall-elim*[*of - - τ*] *p1 typ-a a-ok proves-strip-all-single*
by (*meson has-typ-iff-typ-of term-ok-def wt-term-def*)
thus Θ, $\Gamma \vdash$ *subst-term* $[((x,\ \tau),\ a)]\ B$
using *instantiate-var-same-type'' assms closed-B* **by** *simp*
qed

lemma *subst-term-single-no-change*[*simp*]:
assumes *nvar*: $(x,\tau) \notin fv\ B$
shows *subst-term* $[((x,\tau),\ t)]\ B = B$
using *assms* **by** (*induction B*) *auto*

lemma *fv-subst-term-single*:
assumes *var*: $(x,\tau) \in fv\ B$
assumes $\bigwedge p\ .\ p \in fv\ t \Longrightarrow p \sim= (x,\tau)$
shows *fv* (*subst-term* $[((x,\tau),\ t)]\ B$) = *fv B* − $\{(x,\tau)\}$ ∪ *fv t*
using *assms* **proof** (*induction B*)
case (*App B1 B2*)
then show *?case*
by (*cases* $(x,\tau) \in fv\ B1$; *cases* $(x,\tau) \in fv\ B2$) *auto*
qed *simp-all*

lemma *inst-vars-pre*:
assumes *wf-theory*: *wf-theory* Θ
assumes *B*: Θ, $\Gamma \vdash B$

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*
using *assms* **proof**(*induction insts arbitrary*: *B*)
case *Nil*
then show *?case* **using** *B* **by** *simp*
next
case (*Cons x xs*)

from *this* **obtain** *idn ty t* **where** *x*: *x* = ((*idn*, *ty*), *t*) **by** (*metis prod.collapse*)

have Θ, Γ ⊢ *fold* (λ*single*. *subst-term* [*single*]) (*x* # *xs*) *B*
⟷ Θ, Γ ⊢ *fold* (λ*single*. *subst-term* [*single*]) *xs* (*subst-term* [*x*] *B*)
by *simp*
moreover have Θ, Γ ⊢ *fold* (λ*single*. *subst-term* [*single*]) *xs* (*subst-term* [*x*] *B*)
proof–
have *single*: Θ, Γ ⊢ (*subst-term* [*x*] *B*) **using** *inst-var Cons* **by** (*simp add*: *x*)
show *?thesis* **using** *Cons single* **by** *simp*
qed
ultimately show *?case* **by** *simp*
qed

lemma *subterm-term-ok′*:
is-std-sig Σ ⟹ *term-ok′* Σ *t* ⟹ *is-closed st* ⟹ *occs st t* ⟹ *term-ok′* Σ *st*
proof (*induction t arbitrary*: *st*)
case (*Abs T t*)
then show *?case* **by** (*auto simp add*: *is-open-def*)
next
case (*App t1 t2*)
then show *?case* **using** *term-ok′-occs* **by** *blast*
qed *auto*

lemma *infinite-fv-UNIV*: *infinite* (*UNIV* :: (*indexname* × *typ*) *set*)
by (*simp add*: *finite-prod*)

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*
using *assms proves.implies-intro* **apply** (*simp add*: *wt-term-def*)
by (*metis Diff-empty Diff-insert0*)

lemma *implies-intro′-pre2*:
assumes *wf-theory* Θ Θ, $\Gamma \vdash B$ *term-ok* Θ *A typ-of A* = *Some propT A* $\in \Gamma$
shows Θ, $\Gamma \vdash A \longmapsto B$
proof−
have *1*: Θ, $\Gamma - \{A\} \vdash A \longmapsto B$
using *assms proves.implies-intro* **by** (*simp add*: *wt-term-def*)
have Θ, $\Gamma - \{A\} - \{A\} \vdash A \longmapsto (A \longmapsto B)$
using *assms proves.implies-intro*
by (*simp add*: *1 implies-intro′-pre*)
moreover have Θ, $\{A\} \vdash A$
using *proves.assume assms*
by (*simp add*: *trivial-pre-depr*)
moreover have $\Gamma = (\Gamma - \{A\} - \{A\}) \cup \{A\}$
using *assms* **by** *auto*
ultimately show *?thesis* **using** *proves.implies-elim* **by** *metis*
qed

lemma *subst-term-preserves-typ-of1*[*simp*]:
typ-of1 Ts (*subst-term* [(*x*, τ), *Fv y* τ)] *t*) = *typ-of1 Ts t*
by (*induction Ts t rule*: *typ-of1.induct*) (*fastforce*)+

lemma *subst-term-preserves-typ-of*[*simp*]:
typ-of (*subst-term* [(*x*, τ), *Fv y* τ)] *t*) = *typ-of t*
using *typ-of-def* **by** *simp*

lemma *subst-term-preserves-term-ok′*[*simp*]:
term-ok′ Σ (*subst-term* [(*x*, τ), *Fv y* τ)] *t*) $\longleftrightarrow$ *term-ok′* Σ *t*
by (*induction t*) *auto*

lemma *subst-term-preserves-term-ok*[*simp*]:
term-ok Θ (*subst-term* [(*x*, τ), *Fv y* τ)] *A*) $\longleftrightarrow$ *term-ok* Θ *A*
by (*simp add*: *wt-term-def*)

lemma *not-in-FV-in-fv-not-in*: $(x,\tau) \notin FV\ \Gamma \Longrightarrow (x,\tau) \in fv\ t \Longrightarrow t \notin \Gamma$
by *auto*

lemma *subst-term-fv*: *fv* (*subst-term* [(*x*, τ), *Fv y* τ)] *t*)
= (*if* $(x,\tau) \in$ *fv t then insert* (y,τ) *else id*) (*fv t* − {(x,τ)})
by (*induction t*) *auto*

lemma *rename-free*:
assumes *wf-theory*: *wf-theory* Θ
assumes *B*: Θ, $\Gamma \vdash B$
assumes *free*: $(x,\tau) \notin FV\ \Gamma$
shows Θ, $\Gamma \vdash$ *subst-term* [(*x*, τ), *Fv y* τ)] *B*
by (*metis B free inst-var proved-terms-well-formed*(*2*) *subst-term-single-no-change*
term-ok-vars-typ-ok term-ok-var wf-theory typ-of-var)

lemma *tvs-subst-term-single*[*simp*]: *tvs* (*subst-term* [((*x*, *τ*), *Fv y τ*)] *A*) = *tvs A*
by (*induction A*) *auto*

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 (*induction* Γ *B arbitrary*: *A rule*: *proves.induct*)
case (*axiom A insts* Γ *A′*)
then show *?case* **using** *proves.axiom axiom* **by** *metis*
next
case (*assume A* Γ *A′*)
then show *?case* **using** *proves.intros* **by** *blast*
next
case (*forall-intro* Γ *B x τ*)

have ∃ *y* . *y*∉*fst* ‘ (*fv A* ∪ *fv B* ∪ *FV* Γ)
proof−
have *finite* (*FV* Γ)
using *finite-fv forall-intro.prems* **by** *auto*
hence *finite* (*fv A* ∪ *fv B* ∪ *FV* Γ) **by** *simp*
hence *finite* (*fst* ‘ (*fv A* ∪ *fv B* ∪ *FV* Γ)) **by** *simp*

thus *?thesis* **using** *variant-variable-fresh* **by** *blast*
qed
from *this* **obtain** *y* **where** *y*∉*fst* ‘ (*fv A* ∪ *fv B* ∪ *FV* Γ) **by** *auto*

have *not-in-ren*: *subst-term* [((*x*, *τ*), *Fv y τ*)] *A* ∉ Γ
proof(*cases* (*x*, *τ*) ∈ *fv A*)
case *True*
show *?thesis*
apply (*rule not-in-FV-in-fv-not-in*[*of y τ*])
apply (*metis* (*full-types*) *Un-iff* ‹*y* ∉ *fst* ‘ (*fv A* ∪ *fv B* ∪ *FV* Γ)› *fst-conv image-eqI*)
using *True subst-term-fv* **by** *auto*
next
case *False*
hence *subst-term* [((*x*, *τ*), *Fv y τ*)] *A* = *A*
by *simp*
then show *?thesis*
by (*simp add*: *forall-intro.prems*(*3*))
qed
have *term-ok-ren*: *term-ok* Θ (*subst-term* [((*x*, *τ*), *Fv y τ*)] *A*)
using *forall-intro.prems*(*1*) *subst-term-preserves-term-ok* **by** *blast*
have *typ-of-ren*: *typ-of* (*subst-term* [((*x*, *τ*), *Fv y τ*)] *A*) = *Some propT*
using *forall-intro.prems* **by** *auto*

hence Θ, *insert* (*subst-term* [((*x*, τ), *Fv y* τ)] *A*) Γ ⊢ *B*
using *forall-intro.IH forall-intro.prems*(*3*) *forall-intro.prems*(*4*)
not-in-ren term-ok-ren typ-of-ren **by** *blast*
have Θ, *insert* (*subst-term* [((*x*, τ), *Fv y* τ)] *A*) Γ ⊢ *mk-all x* τ *B*
apply (*rule proves.forall-intro*)
apply (*simp add*: *forall-intro.hyps*(*1*))
using ‹Θ, *insert* (*subst-term* [((*x*, τ), *Fv y* τ)] *A*) Γ ⊢ *B*› **apply** *blast*
subgoal using *subst-term-fv* ‹(*x*, τ) ∉ *FV* Γ› **apply** *simp*
by (*metis Un-iff* ‹*y* ∉ *fst* ‘ (*fv A* ∪ *fv B* ∪ *FV* Γ)› *fst-conv image-eqI*)
using *forall-intro.hyps*(*4*) **by** *blast*
hence Θ, Γ ⊢ *subst-term* [((*x*, τ), *Fv y* τ)] *A* ⟼ *mk-all x* τ *B*
using *forall-intro.hyps*(*1*) *forall-intro.hyps*(*2*) *forall-intro.hyps*(*4*)
forall-intro.prems(*1*) *forall-intro.prems*(*3*)
implies-intro′-pre local.forall-intro not-in-ren proves.forall-intro
subst-term-preserves-typ-of term-ok-ren **by** *auto*
hence Θ, Γ ⊢ *subst-term* [((*y*, τ), *Fv x* τ)]
(*subst-term* [((*x*, τ), *Fv y* τ)] *A* ⟼ *mk-all x* τ *B*)
by (*smt Un-iff* ‹*y* ∉ *fst* ‘ (*fv A* ∪ *fv B* ∪ *FV* Γ)› *forall-intro.hyps*(*1*)
fst-conv image-eqI rename-free)
hence Θ, Γ ⊢ *A* ⟼ *mk-all x* τ *B*
using *forall-intro proves.forall-intro implies-intro′-pre* **by** *auto*
moreover have Θ, {*A*} ⊢ *A*
using *forall-intro.prems*(*1*) *local.forall-intro*(*7*) *trivial-pre-depr* **by** *blast*
ultimately show *?case*
using *implies-elim* **by** *fastforce*
next
case (*forall-elim* Γ τ *B a*)
then show *?case* **using** *proves.forall-elim* **by** *blast*
next
case (*implies-intro* Γ *B N*)
then show *?case*
proof (*cases A*=*N*)
case *True*

hence Θ,Γ − {*N*} ⊢ *N* ⟼ *B*

using *implies-intro.hyps*(*1*) *implies-intro.hyps*(*2*) *implies-intro.hyps*(*3*)
implies-intro.hyps(*4*) *proves.implies-intro* **by** *blast*
hence Θ,Γ − {*N*} ⊢ *A* ⟼ *N* ⟼ *B*
using *True implies-intro′-pre implies-intro.hyps*(*1*) *implies-intro.hyps*(*3*)
implies-intro.hyps(*4*) *implies-intro.prems*(*1*) **by** *blast*
hence Θ,*insert N* Γ ⊢ *B*
using *True implies-elim implies-intro insert-absorb* **by** *fastforce*
then show *?thesis*
using *True implies-elim implies-intro.hyps*(*3*) *implies-intro.hyps*(*4*) *implies-intro.prems*(*1*)
trivial-pre-depr **by** (*simp add*: *implies-intro′-pre2 implies-intro.hyps*(*1*))
next
case *False*

hence *s*: *insert A* (Γ − {*N*}) = *insert A* Γ − {*N*} **by** *auto*

have *I*: Θ,*insert A* Γ ⊢ *B*
using *implies-intro.prems False* **by** (*auto intro*!: *implies-intro.IH*)

show *?thesis*
apply (*subst s*)
apply (*rule proves.implies-intro*)
using *implies-intro.hyps I* **by** *auto*
qed
next
case (*implies-elim* Γ$_1$ *A′ B* Γ$_2$)
show *?case*
using *proves.implies-elim implies-elim* **by** (*metis UnCI Un-insert-left finite-Un*)
next
case (*β-conversion* Γ *s T t x*)
then show *?case* **using** *proves.β-conversion* **by** *blast*
next
case (*eta t τ τ′*)
then show *?case* **using** *proves.eta* **by** *simp*
next
case (*of-class c T′ T* Γ)
then show *?case*
by (*simp add*: *proves.of-class*)
qed
corollary *weaken-proves*: Θ, Γ ⊢ *B* ⟹ *term-ok* Θ *A* ⟹ *typ-of A* = *Some propT*
⟹ *finite* Γ
⟹ Θ, *insert A* Γ ⊢ *B*
using *weaken-proves′* **by** (*metis insert-absorb*)

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*
by (*induction* Γ*2 arbitrary*: Γ *rule*: *finite-induct*) (*use weaken-proves* **in** *auto*)

lemma *no-tvsT-imp-subst-typ-unchanged*: *tvsT T* = *empty* ⟹ *subst-typ insts T*
= *T*
by (*simp add*: *no-tvsT-imp-tsubsT-unchanged tsubstT-simulates-subst-typ*)

lemma *subst-typ-fv*:
shows *apsnd* (*subst-typ insts*) ‘ *fv B* = *fv* (*subst-typ′ insts B*)
by (*induction B*) *auto*

lemma *subst-typ-fv-point*:
assumes (*x*, *τ*) ∈ *fv B*
shows (*x*, *subst-typ insts τ*) ∈ *fv* (*subst-typ′ insts B*)
using *subst-typ-fv* **by** (*metis apsnd-conv assms image-eqI*)

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)
using *assms* **proof** (*induction* τ)
  case (*Tv idn* τ)
  then show *?case*
    by (*cases lookup* ($\lambda x.$ x = (*idn*, τ)) *insts*)
      (*fastforce simp add*: *list-all-iff dest*: *lookup-present-eq-key'* *split*: *prod.splits*)+

qed (*auto simp add*: *list-all-iff lookup-present-eq-key'* *split*: *option.splits*)

lemma *subst-typ-comp-single-left*: *subst-typ* [*single*] (*subst-typ insts T*)
  = *subst-typ* (*map* (*apsnd* (*subst-typ* [*single*])) *insts*@[*single*]) *T*
proof (*induction T*)
  case (*Tv idn ty*)
  then show *?case* **by** (*induction insts*) *auto*
qed *auto*

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 (*induction T*)
  case (*Tv idn S*)
  then show *?case*
  proof (*cases lookup* ($\lambda x.$ x = (*idn*,*S*)) *insts*)
    case *None*
   hence *lookup* ($\lambda x.$ x = (*idn*, *S*)) (*map* (*apsnd* (*subst-typ* [*single*])) *insts*) = *None*

      by (*induction insts*) (*auto split*: *if-splits*)
    then show *?thesis*
      using *None* **apply** *simp*
    by (*metis eq-fst-iff list.set-map lookup.simps*(*2*) *lookup-None-iff subst-typ.simps*(*2*)

        subst-typ-comp subst-typ-nil the-default.simps(*1*))
  next
    case (*Some a*)
     hence *lookup* ($\lambda x.$ x = (*idn*, *S*)) (*map* (*apsnd* (*subst-typ* [*single*])) *insts*) =
Some (*subst-typ* [*single*] *a*)
      by (*induction insts*) (*auto split*: *if-splits*)
    then show *?thesis*
      using *Some* **apply** *simp*
      by (*metis subst-typ.simps*(*2*) *subst-typ-comp-single-left the-default.simps*(*2*))
  qed
qed *auto*

lemma *subst-typ'-comp-single-left*: *subst-typ'* [*single*] (*subst-typ'* *insts t*)
  = *subst-typ'* (*map* (*apsnd* (*subst-typ* [*single*])) *insts*@[*single*]) *t*

by (*induction t*) (*use subst-typ-comp-single-left* **in** *auto*)

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*
by (*induction t*) (*use subst-typ-comp-single-left-stronger* **in** *auto*)

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*)
using *assms* **proof** (*induction T*)
case (*Ty n Ts*)
have *I*: ∀ *x* ∈ *set Ts* . *typ-ok* Θ (*subst-typ insts x*)
using *Ty* **by** (*auto simp add*: *typ-ok-def list-all-iff split*: *option.splits*)
moreover have (∀ *x* ∈ *set Ts* . *typ-ok* Θ (*subst-typ insts x*)) =
(∀ *x* ∈ *set* (*map* (*subst-typ insts*) *Ts*) . *typ-ok* Θ *x*) **by** (*induction Ts*) *auto*
ultimately have *list-all* (*wf-type* (*sig* Θ)) (*map* (*subst-typ insts*) *Ts*)
using *list-allI typ-ok-def Ball-set typ-ok-def* **by** *fastforce*
then show *?case* **using** *Ty list.pred-mono-strong* **by** (*force split*: *option.splits*)
next
case (*Tv idn τ*)
then show *?case* **by** (*induction insts*) *auto*
qed

lemma *typ-ok-Ty*[*simp*]: *typ-ok* Θ (*Ty n Ts*) ⟹ *list-all* (*typ-ok* Θ) *Ts*
by (*auto simp add*: *typ-ok-def list.pred-mono-strong split*: *option.splits*)
lemma *typ-ok-sig-Ty*[*simp*]: *typ-ok-sig* Σ (*Ty n Ts*) ⟹ *list-all* (*typ-ok-sig* Σ) *Ts*
by (*auto simp add*: *list.pred-mono-strong split*: *option.splits*)

lemma *wf-theory-imp-wf-osig*: *wf-theory* Θ ⟹ *wf-osig* (*osig* (*sig* Θ))
by (*cases* Θ *rule*: *theory-full-exhaust*) *simp*

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

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−
have *complete-tcsigs* (*subclass oss*) (*tcsigs oss*)
using *assms*(*1*) **by** (*cases oss*) *simp*
thus *?thesis*
using *assms*(*2*−*4*) **by** (*auto simp add*: *class-les-def class-leq-def complete-tcsigs-def intro*!: *domI ranI*)

qed

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′*
using *assms*(*2*,*3*,*1*) **proof** (*induction* (*sub*,*tcs*) *T S arbitrary*: *S′ rule*: *has-sort.induct*)
case (*has-sort-Tv S S′ tcs a*)
then show *?case*
using *wf-osig.simps wf-subclass-loc.intro wf-subclass-loc.sort-leq-trans* **by** *blast*
next
case (*has-sort-Ty κ K S Ts*)
show *?case*
proof (*rule has-sort.has-sort-Ty*[**where** *dm*=*K*])
show *tcs κ* = *Some K*
using *has-sort-Ty.hyps*(*1*) **.**
next
show ∀ *C*∈*S′*. ∃ *Ss*. *K C* = *Some Ss* ∧ *list-all2* (*has-sort* (*sub*, *tcs*)) *Ts Ss*
proof (*rule ballI*)
fix *C* **assume** *C*: *C* ∈ *S′*
show ∃ *Ss*. *K C* = *Some Ss* ∧ *list-all2* (*has-sort* (*sub*, *tcs*)) *Ts Ss*
proof (*cases C* ∈ *S*)
case *True*
then show *?thesis*
using *list-all2-mono has-sort-Ty.hyps*(*2*) **by** *fastforce*
next
case *False*
from *this* **obtain** *C′* **where** *C′*:
C′ ∈ *S class-les sub C′ C*
by (*metis C class-les-def has-sort-Ty.prems*(*1*) *has-sort-Ty.prems*(*2*)
sort-leq-def
subclass.simps wf-osig-imp-wf-subclass-loc wf-subclass-loc.class-leq-antisym)
from *this* **obtain** *Ss′* **where** *Ss′*:
K C′ = *Some Ss′ list-all2* (*has-sort* (*sub*,*tcs*)) *Ts Ss′*
using *list-all2-mono has-sort-Ty.hyps*(*2*) **by** *fastforce*
from *this* **obtain** *Ss* **where** *Ss*: *K C* = *Some Ss*
using *has-sort-Ty.prems class-les-mgd C′*(*2*) *has-sort-Ty.hyps*(*1*) *wf-theory-imp-wf-osig*
by *force*
have *lengthSs′*: *length Ts* = *length Ss′*
using *Ss′*(*2*) *list-all2-lengthD* **by** *auto*
have *coregular*:
coregular-tcsigs sub tcs
using *has-sort-Ty.prems*(*2*) *wf-theory-imp-wf-osig wf-tcsigs-def*
by (*metis wf-osig.simps*)

hence *leq*: *list-all2* (*sort-leq sub*) *Ss′ Ss*
using *C′*(*2*) *Ss′*(*1*) *Ss has-sort-Ty.hyps*(*1*) *ranI*
by (*metis class-les-def coregular-tcsigs-def domI option.sel*)

have *list-all2* (*has-sort* (*sub*,*tcs*)) *Ts Ss*

```
      proof(rule list-all2-all-nthI)
        show length Ts = length Ss
          using Ss Ss'(1) lengthSs' wf-theory-imp-wf-osig leq list-all2-lengthD by
auto
      next
        fix n assume n: n < length Ts
        hence sort-leq sub (Ss' ! n) (Ss ! n)
          using leq by (simp add: lengthSs' list-all2-nthD)
        thus has-sort (sub,tcs) (Ts ! n) (Ss ! n)
      using has-sort-Ty.hyps(2) has-sort-Ty.prems(2) C'(1) Ss'(1) n list-all2-nthD
          by fastforce
      qed

      thus ∃ Ss. K C = Some Ss ∧ list-all2 (has-sort (sub, tcs)) Ts Ss
        using Ss by (simp)
    qed
  qed
 qed
qed

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'
by (metis has-sort-sort-leq-osig subclass.elims wf-theory-imp-wf-osig)

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
using assms proof(induction T arbitrary: S)
 case (Ty κ Ts)
 obtain cl tcs where cltcs: osig (sig Θ) = (cl, tcs)
   by fastforce
 moreover obtain K where tcsigs (osig (sig Θ)) κ = Some K
   using Ty.prems(2) has-sort.simps by auto
 ultimately have mgd: tcs κ = Some K
   by simp
 have has-sort (osig (sig Θ)) (subst-typ insts (Ty κ Ts)) S
   = has-sort (osig (sig Θ)) (Ty κ (map (subst-typ insts) Ts)) S
   by simp
 moreover have has-sort (osig (sig Θ)) (Ty κ (map (subst-typ insts) Ts)) S
 proof (subst cltcs, rule has-sort-Ty[of tcs, OF mgd], rule ballI)
   fix C assume C: C ∈ S
   obtain Ss where Ss: K C = Some Ss
     using C Ty.prems(2) mgd has-sort.simps cltcs by auto
   have list-all2 (has-sort (osig (sig Θ))) (map (subst-typ insts) Ts) Ss
   proof (rule list-all2-all-nthI)
     show length (map (subst-typ insts) Ts) = length Ss
```

```
    using C Ss Ty.prems(2) list-all2-lengthD mgd has-sort.simps cltcs by fastforce
  next
    fix n assume n: n < length (map (subst-typ insts) Ts)

    have list-all2 (has-sort (cl, tcs)) Ts Ss
      using C Ss Ty.prems(2) cltcs has-sort.simps mgd by auto
    hence 1: has-sort (osig (sig Θ)) (Ts ! n) (Ss ! n)
      using cltcs list-all2-conv-all-nth n by auto
    have has-sort (osig (sig Θ)) (subst-typ insts (Ts ! n)) (Ss ! n)
      using 1 n Ty.prems cltcs C Ss mgd Ty.IH by auto

    then show has-sort (osig (sig Θ)) (map (subst-typ insts) Ts ! n) (Ss ! n)
      using n by auto
  qed
  thus ∃ Ss. K C = Some Ss ∧ list-all2 (has-sort (cl, tcs)) (map (subst-typ insts)
Ts) Ss
    using Ss cltcs by simp
 qed
 ultimately show ?case
  by simp
next
 case (Tv idn S')
 show ?case
 proof(cases (lookup (λx. x = (idn, S')) insts))
  case None
  then show ?thesis using Tv by simp
 next
  case (Some res)
  hence ((idn, S'), res) ∈ set insts using lookup-present-eq-key' by fast
  hence has-sort (osig (sig Θ)) res S' using Tv
    using split-list by fastforce
  moreover have 1: sort-leq (subclass (osig (sig Θ))) S' S
    using Tv.prems(2) has-sort-Tv-imp-sort-leq by blast
  ultimately show ?thesis
    using Some Tv(2) has-sort-Tv-imp-sort-leq apply simp
    using assms(1) 1 has-sort-sort-leq by blast
 qed
qed


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)
using assms proof (induction t arbitrary: T Ts)
next
 case (App t1 t2)
 from this obtain RT where typ-of1 Ts t1 = Some (RT → T)
  using typ-of1-split-App-obtains by blast
```

hence *typ-of1* (*map* (*subst-typ insts*) *Ts*) (*subst-typ′ insts t1*) =
Some (*subst-typ insts* (*RT* → *T*)) **using** *App.IH*(*1*) **by** *blast*
moreover have *typ-of1* (*map* (*subst-typ insts*) *Ts*) (*subst-typ′ insts t2*) = *Some* (*subst-typ insts RT*)
using *App* ‹*typ-of1 Ts t1* = *Some* (*RT* → *T*)› *typ-of1-fun-typ* **by** *blast*
ultimately show *?case* **by** *simp*
qed (*fastforce split*: *if-splits simp add*: *bind-eq-Some-conv*)+

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*)
using *assms subst-typ-preserves-Some-typ-of1 typ-of-def* **by** *fastforce*

lemma *subst-typ′-incr-bv*:
subst-typ′ insts (*incr-bv inc lev t*) = *incr-bv inc lev* (*subst-typ′ insts t*)
by (*induction inc lev t rule*: *incr-bv.induct*) *auto*

lemma *subst-typ′-incr-boundvars*:
subst-typ′ insts (*incr-boundvars lev t*) = *incr-boundvars lev* (*subst-typ′ insts t*)
using *subst-typ′-incr-bv incr-boundvars-def* **by** *simp*

lemma *subst-typ′-subst-bv1*: *subst-typ′ insts* (*subst-bv1 t n u*)
= *subst-bv1* (*subst-typ′ insts t*) *n* (*subst-typ′ insts u*)
by (*induction t n u rule*: *subst-bv1.induct*) (*auto simp add*: *subst-typ′-incr-boundvars*)

lemma *subst-typ′-subst-bv*: *subst-typ′ insts* (*subst-bv t u*)
= *subst-bv* (*subst-typ′ insts t*) (*subst-typ′ insts u*)
using *subst-typ′-subst-bv1 subst-bv-def* **by** *simp*

lemma *subst-typ-no-tvsT-unchanged*:
∀(*f*, *s*) ∈ *set insts* . *f* ∉ *tvsT T* ⟹ *subst-typ insts T* = *T*
proof (*induction T*)
case (*Ty n Ts*)
then show *?case* **by** (*induction Ts*) (*fastforce split*: *prod.splits*)+
next
case (*Tv idn S*)
then show *?case*
by *simp* (*smt case-prodD case-prodE find-None-iff lookup-None-iff-find-None the-default.simps*(*1*))
qed

lemma *subst-typ′-no-tvs-unchanged*:
∀(*f*, *s*) ∈ *set insts* . *f* ∉ *tvs t* ⟹ *subst-typ′ insts t* = *t*
by (*induction t*) (*use subst-typ-no-tvsT-unchanged* **in** ‹*fastforce*+›)

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*)
using *assms term-ok′-subst-typ′ typ-ok-def*
by (*metis list.pred-mono-strong wf-theory-imp-is-std-sig wf-type-imp-typ-ok-sig*)

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*)
using *assms subst-typ-preserves-Some-typ-of wt-term-def subst-typ′-preserves-term-ok′*
by *auto*

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*
using *assms* **proof** (*induction T*)
case (*Ty n Ts*)
then show *?case* **by** (*induction Ts*) *auto*
qed *auto*

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*
using *assms* **proof** (*induction t arbitrary*: *lev*)
case (*Ct n T*)
then show *?case*
by (*simp add*: *subst-typ-rename-vars-cancel*)
next
case (*Fv idn T*)
then show *?case*
by (*clarsimp simp add*: *subst-typ-rename-vars-cancel*) (*metis subst-typ-rename-vars-cancel*)
next
case (*Abs T t*)
thus *?case*
by (*simp add*: *image-Un subst-typ-rename-vars-cancel*)
next
case (*App t1 t2*)
then show *?case*
by (*simp add*: *image-Un*)
qed *auto*

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*

using *assms* **proof** (*induction t arbitrary*: *i*)
qed *auto*

lemma *bind-fv-renamed-var*:
assumes $y \notin$ *fst* ' *fv t*
shows *bind-fv* (*y*, τ) (*subst-term* [((*x*, τ), *Fv y* τ)] *t*)
= *bind-fv* (*x*, τ) *t*
using *bind-fv2-renamed-var bind-fv-def assms* **by** *auto*

lemma *subst-typ'-rename-tvar-bind-fv2*:
assumes $y \notin$ *fst* ' *fv t*
assumes (*b*, *S*) $\notin$ *tvs t*
assumes (*b*, *S*) $\notin$ *tvsT* τ
shows *bind-fv2* (*y*, *subst-typ* [((*a*, *S*), *Tv b S*)] τ) *i*
(*subst-typ'* [((*a*,*S*), *Tv b S*)] (*subst-term* [((*x*, τ), *Fv y* τ)] *t*))
= *subst-typ'* [((*a*,*S*), *Tv b S*)] (*bind-fv2* (*x*, τ) *i t*)
using *assms* **proof** (*induction t arbitrary*: *i*)
qed *auto*

lemma *subst-typ'-rename-tvar-bind-fv*:
assumes $y \notin$ *fst* ' *fv t*
assumes (*b*, *S*) $\notin$ *tvs t*
assumes (*b*, *S*) $\notin$ *tvsT* τ
shows *bind-fv* (*y*, *subst-typ* [((*a*,*S*), *Tv b S*)] τ)
(*subst-typ'* [((*a*,*S*), *Tv b S*)] (*subst-term* [((*x*, τ), *Fv y* τ)] *t*))
= *subst-typ'* [((*a*,*S*), *Tv b S*)] (*bind-fv* (*x*, τ) *t*)
using *bind-fv-def assms subst-typ'-rename-tvar-bind-fv2* **by** *auto*

lemma *tvar-in-fv-in-tvs*: (*a*, τ) $\in$ *fv B* $\Longrightarrow$ (*x*, *S*) $\in$ *tvsT* τ $\Longrightarrow$ (*x*, *S*) $\in$ *tvs B*
by (*induction B*) *auto*

lemma *tvs-bind-fv2-subset*: *tvs* (*bind-fv2* (*a*, τ) *i B*) $\subseteq$ *tvs B*
by (*induction B arbitrary*: *i*) *auto*

lemma *tvs-bind-fv-subset*: *tvs* (*bind-fv* (*a*, τ) *B*) $\subseteq$ *tvs B*
using *tvs-bind-fv2-subset bind-fv-def* **by** *simp*

lemma *subst-typ-rename-tvar-preserves-eq*:
(*y*, *S*) $\notin$ *tvsT T* $\Longrightarrow$ (*y*, *S*) $\notin$ *tvsT* τ $\Longrightarrow$
subst-typ [((*x*, *S*), *Tv y S*)] *T* = *subst-typ* [((*x*, *S*), *Tv y S*)] τ $\Longrightarrow$ *T*=τ
proof (*induction T arbitrary*: τ)
case (*Ty n Ts*)
then show *?case*
proof (*induction* τ)
case (*Ty n Ts*)
then show *?case*
by *simp* (*smt list.inj-map-strong*)
next
case (*Tv n S*)

then show *?case*
by (*auto split*: *if-splits*)
qed
next
case (*Tv n S*)
then show *?case* **by** (*induction τ*) (*auto split*: *if-splits*)
qed

lemma *subst-typ'-subst-term-rename-var-swap*:
assumes *b* ∉ *fst* ‘ *fv B*
assumes (*y*, *S*) ∉ *tvs B*
assumes (*y*, *S*) ∉ *tvsT τ*
shows *subst-typ'* [((*x*, *S*), *Tv y S*)] (*subst-term* [((*a*, *τ*), *Fv b τ*)] *B*)
= *subst-term* [((*a*, (*subst-typ* [((*x*, *S*), *Tv y S*)] *τ*)), *Fv b* (*subst-typ* [((*x*, *S*), *Tv y S*)] *τ*))]
(*subst-typ'* [((*x*, *S*), *Tv y S*)] *B*)
using *assms* **proof** (*induction B*)
case (*Fv idn T*)
then show *?case* **using** *subst-typ-rename-tvar-preserves-eq* **by** *auto*
qed *auto*

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

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
using *tvar-not-in-term-imp-free-not-in-term* **by** *simp*

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−
obtain *fresh-idns* **where** *fresh-idns*:
length fresh-idns = *length insts*
∀ *idn* ∈ *set fresh-idns* .
idn ∉ *fst* ‘ (*fv B* ∪ (⋃ *t*∈*snd* ‘ *set insts* . (*fv t*)) ∪ (*fst* ‘ *set insts*)) ∪ *fst* ‘ (*FV* Γ)
distinct fresh-idns
using *distinct-fresh-rename-idns fresh-fresh-rename-idns length-fresh-rename-idns*

finite-FV finite
by (*metis finite-imageI*)
have *0*: *subst-term insts B*
= *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*)))) *B*)
using *fresh-idns distinct subst-term-combine′* **by** *simp*

from *fresh-idns vars a-ok typ-a free distinct* **have** *1*:
Θ, Γ ⊢ (*fold* (λ*single acc* . *subst-term* [*single*] *acc*)
(*zip* (*map fst insts*) (*map2 Fv fresh-idns* (*map snd* (*map fst insts*)))) *B*)
proof (*induction fresh-idns insts rule*: *rev-induct2*)
case *Nil*
then show *?case* **using** *B* **by** *simp*
next
case (*snoc x xs y ys*)
from *snoc* **have** *term-oky*: *term-ok* Θ (*Fv* (*fst* (*fst y*)) (*snd* (*fst y*)))
by (*auto simp add*: *wt-term-def split*: *prod.splits*)

have *1*: Θ, Γ ⊢ *fold* (λ*single*. *subst-term* [*single*])
(*zip* (*map fst ys*) (*map2 Fv xs* (*map snd* (*map fst ys*)))) *B*
apply (*rule snoc.IH*)
subgoal using *snoc.prems*(*1*) **by** (*clarsimp split*: *prod.splits*) (*smt UN-I Un-iff fst-conv image-iff*)
using *snoc.prems*(*2−7*) **by** *auto*

moreover obtain *yn n* **where** *ynn*: *fst y* = (*yn*, *n*) **by** *fastforce*
moreover have Θ,Γ ⊢ *subst-term* [(*fst y*, *Fv x n*)]
(*fold* (λ*single*. *subst-term* [*single*]) (*zip* (*map fst* (*ys*))
(*map2 Fv* (*xs*) (*map snd* (*map fst* (*ys*))))) *B*)
apply (*simp only*: *ynn*)
apply (*rule inst-var*[*of* Θ Γ (*fold* (λ*single*. *subst-term* [*single*]) (*zip* (*map fst* (*ys*))
(*map2 Fv* (*xs*) (*map snd* (*map fst* (*ys*))))) *B*) (*Fv x n*) *n yn*])
using *snoc.prems* ‹*wf-theory* Θ› *1* **apply** (*solves simp*)+
using *term-oky ynn* **apply** (*simp add*: *wt-term-def typ-of-def*)
using *term-oky ynn* **apply** (*simp add*: *wt-term-def typ-of-def*)
using *snoc.prems*(*6*) *ynn* **by** *auto*

moreover have *fold* (λ*single*. *subst-term* [*single*]) (*zip* (*map fst* (*ys* @ [*y*]))
(*map2 Fv* (*xs* @ [*x*]) (*map snd* (*map fst* (*ys* @ [*y*])))) *B*
= *subst-term* [(*fst y*, *Fv x* (*snd* (*fst y*)))]
(*fold* (λ*single*. *subst-term* [*single*]) (*zip* (*map fst* (*ys*))
(*map2 Fv* (*xs*) (*map snd* (*map fst* (*ys*))))) *B*)
using *snoc.hyps* **by** (*induction xs ys rule*: *list-induct2*) *simp-all*

ultimately show *?case* **by** *simp*
qed

```
  define point where point ≡ (fold (λsingle acc . subst-term [single] acc)
    (zip (map fst insts) (map2 Fv fresh-idns (map snd (map fst insts)))) B)

  from fresh-idns vars a-ok typ-a free distinct have 2:
    Θ, Γ ⊢ fold (λsingle acc . subst-term [single] acc)
      (zip (zip fresh-idns (map snd (map fst insts))) (map snd insts))
      point
  proof (induction fresh-idns insts rule: rev-induct2)
    case Nil
    then show ?case using B
      using 1 point-def by auto
  next
    case (snoc x xs y ys)

    from snoc have typ-ofy: typ-of (snd y) = Some (snd (fst y)) by auto

    have 1:  Θ,Γ ⊢ fold (λsingle. subst-term [single])
          (zip (zip xs (map snd (map fst ys))) (map snd ys))
          point
      apply (rule snoc.IH)
     subgoal using snoc.prems(1) by (clarsimp split: prod.splits) (smt UN-I Un-iff
fst-conv image-iff)
      using snoc.prems(2−7) by auto
    moreover obtain yn n where ynn: fst y = (yn, n) by fastforce
     moreover have Θ,Γ ⊢ subst-term [((x, snd (fst y)), snd y)] (fold (λsingle.
subst-term [single])
          (zip (zip (xs) (map snd (map fst (ys))))
            (map snd (ys)))
          point)
      apply (simp only: ynn) apply (rule inst-var)
      using snoc.prems ‹wf-theory Θ› 1 apply (solves simp)+
      using typ-ofy ynn apply (simp add: wt-term-def typ-of-def)
      using snoc.prems apply simp
      by (metis (full-types, opaque-lifting) UN-I fst-conv image-eqI)
    moreover have fold (λsingle. subst-term [single])
          (zip (zip (xs @ [x]) (map snd (map fst (ys @ [y]))))
            (map snd (ys @ [y])))
           point = subst-term [((x, snd (fst y)), snd y)] (fold (λsingle. subst-term
[single])
          (zip (zip (xs) (map snd (map fst (ys))))
            (map snd (ys)))
          point)
      using snoc.hyps by (induction xs ys rule: list-induct2) simp-all

    ultimately show ?case by simp
  qed

  from 0 1 2 show ?thesis using point-def by simp
qed
```

lemma *term-ok-eta-red-step*:
 ¬ *is-dependent t* ⟹ *term-ok* Θ (*Abs T* (*t* $ *Bv 0*)) ⟹ *term-ok* Θ (*decr 0 t*)
 unfolding *term-ok-def wt-term-def* **using** *term-ok'-decr eta-preserves-typ-of* **by** *simp blast*

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−
 have *eq-reflexive-ax* ∈ *axioms* Θ
  using *assms* **by** (*cases* Θ *rule*: *theory-full-exhaust*) *auto*
 moreover obtain τ **where** τ: *typ-of t* = *Some* τ **using** *assms wt-term-def* **by** *auto*
 moreover hence *typ-ok* Θ τ **using** *assms term-ok-imp-typ-ok* **by** *blast*
 ultimately have Θ, {} ⊢ *subst-typ'* [((*Var* (*STR* ′′′*a*′′, *0*), *full-sort*), τ)] *eq-reflexive-ax*
  using *axiom-subst-typ' assms* **by** (*simp del*: *term-ok-def*)
 hence Θ, {} ⊢ *subst-term* [((*Var* (*STR* ′′*x*′′, *0*), τ), *t*)]
  (*subst-typ'* [((*Var* (*STR* ′′′*a*′′, *0*), *full-sort*), τ)] *eq-reflexive-ax*)
  using τ *assms*(*1*) *assms*(*2*) *inst-var* **by** *auto*
 moreover have *subst-term* [((*Var* (*STR* ′′*x*′′, *0*), τ), *t*)]
  (*subst-typ'* [((*Var* (*STR* ′′′*a*′′, *0*), *full-sort*), τ)] *eq-reflexive-ax*)
  = *mk-eq t t*
  using τ **by** (*simp add*: *eq-axs-def typ-of-def*)
 ultimately show *?thesis*
  by *simp*
qed

lemma *unsimp-context*: Γ = {} ∪ Γ
 by *simp*

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, Γ ⊢ *mk-eq t t*
  by (*subst unsimp-context*) (*use assms proves-eq-reflexive-pre weaken-proves-set* **in** *blast*)

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−

have *eq-symmetric-ax* ∈ *axioms* Θ
using *assms* **by** (*cases* Θ *rule*: *theory-full-exhaust*) *auto*
moreover obtain τ **where** τ: *typ-of t* = *Some* τ **using** *assms wt-term-def* **by** *auto*

moreover hence *typ-ok* Θ τ **using** *assms term-ok-imp-typ-ok* **by** *blast*
ultimately have Θ, $\{\} \vdash$ *subst-typ'* [((*Var* (*STR* ′′′*a*′′, *0*), *full-sort*), τ)] *eq-symmetric-ax*
using *assms axiom-subst-typ'* **by** (*auto simp del*: *term-ok-def*)
hence Θ, $\{\} \vdash$ *subst-term* [((*Var* (*STR* ′′*x*′′, *0*), τ), *s*), ((*Var* (*STR* ′′*y*′′, *0*), τ), *t*)]
(*subst-typ'* [((*Var* (*STR* ′′′*a*′′, *0*), *full-sort*), τ)] *eq-symmetric-ax*)
using τ ‹*typ-ok* Θ τ› *term-ok-var assms* **by** (*fastforce intro*!: *inst-var-multiple simp add*: *eq-symmetric-ax-def*)
thus *?thesis*
using τ *assms*(*4*) **by** (*simp add*: *eq-axs-def typ-of-def*)
qed

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 ∀ *A*∈Γ. *term-ok* Θ *A* ∀ *A*∈Γ. *typ-of A* = *Some propT*
shows Θ, $\Gamma \vdash$ *mk-eq s t* $\longmapsto$ *mk-eq t s*
by (*subst unsimp-context*) (*use assms proves-eq-symmetric-pre weaken-proves-set* **in** *blast*)

lemma *proves-eq-symmetric2′*:
assumes *wf-theory* Θ
assumes *term-ok* Θ (*mk-eq s t*)
assumes *finite* Γ ∀ *A*∈Γ. *term-ok* Θ *A* ∀ *A*∈Γ. *typ-of A* = *Some propT*
shows Θ, $\Gamma \vdash$ *mk-eq s t* $\longmapsto$ *mk-eq t s*
proof−
have *term-ok* Θ *s term-ok* Θ *t*
using *assms wt-term-def term-ok-mk-eqD* **by** *blast*+
moreover have *typ-of s* = *typ-of t*
using *assms* **by** (*cases* Θ *rule*: *theory-full-exhaust*)
(*auto simp add*: *tinstT-def typ-of-def wt-term-def bind-eq-Some-conv*)
ultimately show *?thesis*
using *proves-eq-symmetric assms* **by** *blast*

qed

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*
using *proves.implies-elim*[*OF proves-eq-symmetric*[*OF assms*(*1*−*4*), *of* Γ] *assms*(*5*), *OF ctxt*] **by** *simp*

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−
have *eq-transitive-ax* $\in$ *axioms* Θ
using *assms* **by** (*cases* Θ *rule*: *theory-full-exhaust*) *auto*
moreover obtain τ **where** τ: *typ-of t* = *Some* τ **using** *assms wt-term-def* **by** *auto*
moreover hence *ok*: *typ-ok* Θ τ **using** *assms term-ok-imp-typ-ok* **by** *blast*
ultimately have Θ, {} $\vdash$ *subst-typ'* [((*Var* (*STR* ''*a*'', *0*), *full-sort*), τ)] *eq-transitive-ax*
using *assms axiom-subst-typ'* **by** (*auto simp del*: *term-ok-def*)
hence Θ, {} $\vdash$ *subst-term* [((*Var* (*STR* ''*x*'', *0*), τ), *s*), ((*Var* (*STR* ''*y*'', *0*), τ), *t*),
((*Var* (*STR* ''*z*'', *0*), τ), *u*)]
(*subst-typ'* [((*Var* (*STR* ''*a*'', *0*), *full-sort*), τ)] *eq-transitive-ax*)
using τ *assms ok term-ok-var* **by** (*fastforce intro*!: *inst-var-multiple simp add*: *eq-transitive-ax-def*)
moreover have *subst-term* [((*Var* (*STR* ''*x*'', *0*), τ), *s*), ((*Var* (*STR* ''*y*'', *0*), τ), *t*),
((*Var* (*STR* ''*z*'', *0*), τ), *u*)]
(*subst-typ'* [((*Var* (*STR* ''*a*'', *0*), *full-sort*), τ)] *eq-transitive-ax*)
= *mk-eq s t* $\longmapsto$ *mk-eq t u* $\longmapsto$ *mk-eq s u*
using τ *assms*(*5*−*6*) **apply** (*simp add*: *eq-axs-def typ-of-def*)
by (*metis option.sel the-default.simps*(*2*))
ultimately show *?thesis*
by *simp*
qed

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, Γ $\vdash$ *mk-eq s t* $\longmapsto$ *mk-eq t u* $\longmapsto$ *mk-eq s u*
by (*subst unsimp-context*) (*use assms proves-eq-transitive-pre weaken-proves-set*
in *blast*)

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, Γ $\vdash$ *mk-eq s t* $\longmapsto$ *mk-eq t u* $\longmapsto$ *mk-eq s u*
proof−
have *term-ok* Θ *s term-ok* Θ *t term-ok* Θ *u*
using *assms wt-term-def term-ok-mk-eqD* **by** *blast*+
moreover have *typ-of s* = *typ-of t*
using *assms* **by** (*cases* Θ *rule*: *theory-full-exhaust*)
(*auto simp add*: *tinstT-def typ-of-def wt-term-def bind-eq-Some-conv*)
moreover have *typ-of t* = *typ-of u*
using *assms* **by** (*cases* Θ *rule*: *theory-full-exhaust*)
(*auto simp add*: *tinstT-def typ-of-def wt-term-def bind-eq-Some-conv*)
ultimately show *?thesis* **using** *proves-eq-transitive assms* **by** *blast*
qed

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, Γ $\vdash$ *mk-eq s t* Θ, Γ $\vdash$ *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, Γ $\vdash$ *mk-eq s u*
proof−
note *1* = *proves-eq-transitive*[*OF assms*(*1*−*6*), *of* Γ]
note *2* = *proves.implies-elim*[*OF 1 assms*(*7*)]
note *3* = *proves.implies-elim*[*OF 2 assms*(*8*)]
thus *?thesis* **using** *ctxt* **by** *simp*
qed

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−
have *closed*: *is-closed A is-closed B*
using *assms*(*3*) *assms*(*5*) *typ-of-imp-closed* **by** *auto*

have *eq-intr-ax* ∈ *axioms* Θ
using *thy* **by** (*cases* Θ *rule*: *theory-full-exhaust*) *auto*

hence *1*: Θ, {} ⊢ *eq-intr-ax*
by (*simp add*: *axiom′ thy*)
hence Θ, {} ⊢ *subst-term* [((*Var* (*STR* ''*A*'', *0*), *propT*), *A*), ((*Var* (*STR* ''*B*'', *0*), *propT*), *B*)]
eq-intr-ax
using *assms term-ok-var propT-ok* **by** (*fastforce intro*!: *inst-var-multiple simp add*: *eq-intr-ax-def*)
thus *?thesis* **using** *assms* **by** (*simp add*: *eq-axs-def typ-of-def*)
qed

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 ∀ *A*∈Γ. *term-ok* Θ *A* ∀ *A*∈Γ. *typ-of A* = *Some propT*
shows Θ, Γ ⊢ (*A* ⟼ *B*) ⟼ (*B* ⟼ *A*) ⟼ *mk-eq A B*
by (*subst unsimp-context*) (*use assms proves-eq-intr-pre weaken-proves-set* **in** *blast*)

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, Γ ⊢ (*A* ⟼ *B*) Θ, Γ ⊢ (*B* ⟼ *A*)
assumes *ctxt*: *finite* Γ ∀ *A*∈Γ. *term-ok* Θ *A* ∀ *A*∈Γ. *typ-of A* = *Some propT*
shows Θ, Γ ⊢ *mk-eq A B*
proof–
note *1* = *proves-eq-intr*[*OF assms*(*1*−*5*), *of* Γ]
note *2* = *proves.implies-elim*[*OF 1 assms*(*6*)]
note *3* = *proves.implies-elim*[*OF 2 assms*(*7*)]
thus *?thesis* **using** *ctxt* **by** *simp*
qed

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, {} ⊢ *mk-eq A B* ⟼ *A* ⟼ *B*
proof–
have *closed*: *is-closed A is-closed B*
by (*simp-all add*: *assms*(*3*) *assms*(*5*) *typ-of-imp-closed*)
have *eq-elim-ax* ∈ *axioms* Θ
using *thy* **by** (*cases* Θ *rule*: *theory-full-exhaust*) *auto*
hence *1*: Θ, {} ⊢ *eq-elim-ax*
by (*simp add*: *axiom′ thy*)
hence Θ, {} ⊢ *subst-term* [((*Var* (*STR* ''*A*'', *0*), *propT*), *A*), ((*Var* (*STR* ''*B*'',

0), *propT*), *B*)]
eq-elim-ax
using *assms term-ok-var propT-ok* **by** (*fastforce intro*!: *inst-var-multiple simp add*: *eq-elim-ax-def*)
thus *?thesis*
using *assms* **by** (*simp add*: *eq-axs-def typ-of-def*)
qed

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 ∀ *A*∈Γ. *term-ok* Θ *A* ∀ *A*∈Γ. *typ-of A* = *Some propT*
shows Θ, Γ ⊢ *mk-eq A B* ⟼ *A* ⟼ *B*
by (*subst unsimp-context*) (*use assms proves-eq-elim-pre weaken-proves-set* **in** *blast*)

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, Γ ⊢ *mk-eq A B*
assumes *ctxt*: *finite* Γ ∀ *A*∈Γ. *term-ok* Θ *A* ∀ *A*∈Γ. *typ-of A* = *Some propT*
shows Θ, Γ ⊢ *A* ⟼ *B*
using *proves.implies-elim*[*OF proves-eq-elim*[*OF assms*(*1*−*5*)] *assms*(*6*), *of* Γ, *OF ctxt*] **by** *simp*

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, Γ ⊢ *mk-eq A B*
assumes *ctxt*: *finite* Γ ∀ *A*∈Γ. *term-ok* Θ *A* ∀ *A*∈Γ. *typ-of A* = *Some propT*
shows Θ, Γ ⊢ *B* ⟼ *A*
proof−
have Θ, Γ ⊢ *mk-eq B A*
by (*rule proves-eq-symmetric-rule*) (*use assms* **in** *simp-all*)
thus *?thesis* **by** (*intro proves-eq-elim-rule*) (*use assms* **in** *simp-all*)
qed

lemma *proves-eq-combination-pre*:
assumes *thy*: *wf-theory* Θ
assumes *f*: *term-ok* Θ *f typ-of f* = *Some* (τ → τ′)
assumes *g*: *term-ok* Θ *g typ-of g* = *Some* (τ → τ′)
assumes *x*: *term-ok* Θ *x typ-of x* = *Some* τ
assumes *y*: *term-ok* Θ *y typ-of y* = *Some* τ
shows Θ, {} ⊢ *mk-eq f g* ⟼ *mk-eq x y* ⟼ *mk-eq* (*f* $ *x*) (*g* $ *y*)
proof−
have *ok*: *typ-ok* Θ τ *typ-ok* Θ (τ → τ′) *typ-ok* Θ τ′

using *term-ok-betapply term-ok-imp-typ-ok thy typ-of-betaply thy x f* **by** *blast+*

have *eq-combination-ax* ∈ *axioms* Θ
using *thy* **by** (*cases* Θ *rule*: *theory-full-exhaust*) *auto*
moreover have *typ-ok* Θ τ *typ-ok* Θ τ′
using *assms term-ok-imp-typ-ok thy term-ok-betapply typ-of-betaply* **by** *meson+*
ultimately have *1*: Θ, {} ⊢ *subst-typ′*
[((*Var* (*STR* ′′′*a*′′, 0), *full-sort*), τ), ((*Var* (*STR* ′′′*b*′′, 0), *full-sort*), τ′)]
eq-combination-ax
using *assms axiom-subst-typ′* **by** (*simp del*: *term-ok-def*)
hence Θ, {} ⊢ *subst-term*
[((*Var* (*STR* ′′*f*′′, 0), τ → τ′), *f*), ((*Var* (*STR* ′′*g*′′, 0), τ → τ′), *g*),
((*Var* (*STR* ′′*x*′′, 0), τ), *x*), ((*Var* (*STR* ′′*y*′′, 0), τ), *y*)]
(*subst-typ′* [((*Var* (*STR* ′′′*a*′′, 0), *full-sort*), τ), ((*Var* (*STR* ′′′*b*′′, 0), *full-sort*),
τ′)]
eq-combination-ax)
using *assms term-ok-var ok* **by** (*fastforce intro*!: *inst-var-multiple simp add*:
eq-combination-ax-def)
thus *?thesis*
using *assms* **by** (*simp add*: *eq-axs-def typ-of-def*)
qed

lemma *proves-eq-combination*:
assumes *thy*: *wf-theory* Θ
assumes *f*: *term-ok* Θ *f typ-of f* = *Some* (τ → τ′)
assumes *g*: *term-ok* Θ *g typ-of g* = *Some* (τ → τ′)
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 ∀ *A*∈Γ. *term-ok* Θ *A* ∀ *A*∈Γ. *typ-of A* = *Some propT*
shows Θ, Γ ⊢ *mk-eq f g* ⟼ *mk-eq x y* ⟼ *mk-eq* (*f* $ *x*) (*g* $ *y*)
by (*subst unsimp-context*) (*use assms proves-eq-combination-pre weaken-proves-set*
in *blast*)

lemma *proves-eq-combination-rule*:
assumes *thy*: *wf-theory* Θ
assumes *f*: *term-ok* Θ *f typ-of f* = *Some* (τ → τ′)
assumes *g*: *term-ok* Θ *g typ-of g* = *Some* (τ → τ′)
assumes *x*: *term-ok* Θ *x typ-of x* = *Some* τ
assumes *y*: *term-ok* Θ *y typ-of y* = *Some* τ
assumes Θ, Γ ⊢ *mk-eq f g* Θ, Γ ⊢ *mk-eq x y*
assumes *ctxt*: *finite* Γ ∀ *A*∈Γ. *term-ok* Θ *A* ∀ *A*∈Γ. *typ-of A* = *Some propT*
shows Θ, Γ ⊢ *mk-eq* (*f* $ *x*) (*g* $ *y*)
proof−
note *1* = *proves-eq-combination*[*OF assms*(*1*−*9*), *of* Γ]
note *2* = *proves.implies-elim*[*OF 1 assms*(*10*)]
note *3* = *proves.implies-elim*[*OF 2 assms*(*11*)]
thus *?thesis* **using** *ctxt* **by** *simp*
qed

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–
have *ok-Apps*: *term-ok* Θ (*mk-eq f g*) *term-ok* Θ (*mk-eq x y*)
using *assms*(*2*−*3*) *proved-terms-well-formed-pre* **by** *auto*
hence *tyy*: *typ-of y* = *Some* τ **and** *tyg*: *typ-of g* = *Some* $(\tau \to \tau')$
using *term-ok-mk-eq-same-typ thy x f term-okD1* **by** *metis*+
moreover have *term-ok* Θ *x term-ok* Θ *y term-ok* Θ *f term-ok* Θ *g*
using *ok-Apps term-ok-mk-eqD* **by** *blast*+
ultimately show *?thesis* **using** *proves-eq-combination-rule assms* **by** *simp*
qed

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–
have Θ, $\Gamma \vdash$ *A* $\longmapsto$ *B* **using** *proves-eq-elim-rule*[*OF assms*(*1*−*5*) *eq ctxt*] **.**
thus Θ, $\Gamma \vdash$ *B* **using** *proves.implies-elim pA* **by** *fastforce*
qed

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*
by (*metis ctxt eq pA proved-terms-well-formed*(*1*) *proved-terms-well-formed*(*2*)
proves-eq-mp-rule term-ok-mk-eqD term-ok-mk-eq-same-typ thy)

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*: Θ, $\Gamma \vdash$ *mk-eq x y*
shows Θ, $\Gamma \vdash$ *mk-eq* (*P* \$ *x*) (*P* \$ *y*)
proof–

have Θ, $\Gamma \vdash$ *mk-eq P P* **using** *assms proves-eq-reflexive* **by** *blast*
thus *?thesis* **using** *proves-eq-combination-rule assms* **by** *blast*
qed

lemma *proves-beta-step-rule*:
assumes *thy*: *wf-theory* Θ
assumes *abs*: *term-ok* Θ (*Abs T t*) Θ, $\Gamma \vdash$ (*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 Θ, $\Gamma \vdash$ *subst-bv x t*
proof–
have Θ, $\Gamma \vdash$ *mk-eq* ((*Abs T t*) $ *x*) (*subst-bv x t*)
using *proves.*β*-conversion assms* **by** (*simp add*: *term-okD1*)
moreover have *term-ok* Θ (*Abs T t* $ *x*) **and** *tyAbs*: *typ-of* (*Abs T t* $ *x*) = *Some propT*
using *abs(2) proved-terms-well-formed* **by** *simp-all*
moreover have *tySub*: *typ-of* (*subst-bv x t*) = *Some propT*
using *tyAbs* **unfolding** *subst-bv-def typ-of-def*
using *typ-of1-subst-bv-gen'* **by** (*auto simp add*: *bind-eq-Some-conv split*: *if-splits*)
moreover have *term-ok* Θ (*subst-bv x t*)
proof–
have *term-ok'* (*sig* Θ) *t*
using *assms(2) term-ok'.simps(5) wt-term-def term-ok-def* **by** *blast*
hence *term-ok'* (*sig* Θ) (*subst-bv x t*)
using *term-ok'-subst-bv1 x(1)* **by** (*simp add*: *term-okD1 subst-bv-def*)
thus *?thesis*
using *x(1) wt-term-def term-ok'-subst-bv1 subst-bv-def tySub term-okD1* **by** *simp*
qed
ultimately show *?thesis* **apply** –
apply (*rule proves-eq-mp-rule*[**where** *A*=(*Abs T t*) $ *x*])
using *assms* **by** *simp-all*
qed

lemma *proves-add-param-rule*:
assumes *thy*: *wf-theory* Θ
assumes *ctxt*: *finite* Γ
assumes *eq*: Θ, $\Gamma \vdash$ *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 Θ, $\Gamma \vdash$ (*Ct STR ''Pure.all''* (($\tau \to$ *propT*) $\to$ *propT*) $
(*Abs* τ (*mk-eq'* τ' (*f* $ *Bv 0*) (*g* $ *Bv 0*))))
proof–
have *term-ok*: *term-ok* Θ (*mk-eq f g*)
using *eq(1) proved-terms-well-formed-pre* **by** *blast*
hence *term-ok'*: *term-ok* Θ *f term-ok* Θ *g*
apply (*simp add*: *eq(2) wt-term-def*)

```
    using ‹term-ok Θ (mk-eq f g)› wt-term-def typ-of-def  term-ok-app-eqD by blast
  hence typ-of f = typ-of g
    using thy term-ok by (cases Θ rule: theory-full-exhaust)
      (auto simp add: tinstT-def typ-of-def wt-term-def bind-eq-Some-conv)
  hence type': typ-of g = Some (τ → τ')
    using eq(2) by simp

  obtain x where x ∉ fst ` (fv (mk-eq f g) ∪ FV Γ)
    using finite-fv finite-FV infinite-fv-UNIV variant-variable-fresh ctxt
    by (meson finite-Un finite-imageI)
  hence free: (x,τ) ∉ fv (mk-eq f g) ∪ FV Γ
    by force
  hence Θ, Γ ⊢ mk-eq (Fv x τ) (Fv x τ)
    using ctxt proves-eq-reflexive term-ok-var thy type by presburger
  hence Θ, Γ ⊢ mk-eq (f $ Fv x τ) (g $ Fv x τ)
    apply −
    apply (rule proves-eq-combination-rule[where τ'=τ'])
    using assms term-ok' type' by (simp-all del: term-ok-def)
  hence Θ, Γ ⊢ mk-all x τ (mk-eq (f $ Fv x τ) (g $ Fv x τ))
    apply −
    apply (rule proves.forall-intro)
    using thy eq type free by simp-all
  moreover have mk-all x τ (mk-eq (f $ Fv x τ) (g $ Fv x τ))
    = (Ct STR ''Pure.all'' ((τ → propT) → propT) $
      (Abs τ (mk-eq' τ' (f $ Bv 0) (g $ Bv 0))))
    using free eq type type' bind-fv2-changed
    by (fastforce simp add: bind-fv-def bind-fv-unchanged typ-of-def)
  ultimately show ?thesis
    by simp
qed

lemma proves-add-abs-rule:
  assumes thy: wf-theory Θ
  assumes ctxt: finite Γ
  assumes eq: Θ, Γ ⊢ mk-eq f g typ-of f = Some (τ → τ')
  assumes type: typ-ok Θ τ
  assumes ctxt: finite Γ ∀ A∈Γ. term-ok Θ A ∀ A∈Γ. typ-of A = Some propT
  shows Θ, Γ ⊢ mk-eq (Abs τ (f $ Bv 0)) (Abs τ (g $ Bv 0))
proof−
  have ok: term-ok Θ f term-ok Θ g
    using eq(1) proved-terms-well-formed(2) term-ok-mk-eqD by blast+
  have g-ty: typ-of g = Some (τ → τ')
   by (metis eq(1) eq(2) proved-terms-well-formed(2) term-ok-mk-eq-same-typ thy)
  hence closed: is-closed f is-closed g
    using eq(2) typ-of-imp-closed by blast+

  have ok': term-ok Θ (Abs τ (f $ Bv 0)) term-ok Θ (Abs τ (g $ Bv 0))
    using type term-ok-eta-expand ok thy eq(2) g-ty by blast+
```

have *ok-ind*: *wf-term* (*sig* Θ) *f* *wf-term* (*sig* Θ) *g*
using *ok wt-term-def* **by** *simp-all*

note *1* = *proves.eta*[*OF thy ok-ind*(*1*) *typ-of-imp-has-typ*[*OF eq*(*2*)], *of* Γ]
note *2* = *proves.eta*[*OF thy ok-ind*(*2*) *typ-of-imp-has-typ*[*OF g-ty*], *of* Γ]

have *simp'*: *subst-bv x f* = *f subst-bv x g* = *g* **for** *x*
using *ok term-ok-subst-bv-no-change* **by** *auto*

have *s2*: Θ,Γ ⊢ *mk-eq g* (*Abs* τ (*g* $ *Bv 0*))
apply (*rule proves-eq-symmetric-rule*)
using *2* *ok'*(*2*) *ok*(*2*) *thy typ-of-eta-expand*[*OF g-ty*] *g-ty ctxt* **by** (*simp-all add*: *simp'*(*2*))

have *tr1*: Θ,Γ ⊢ *mk-eq* (*Abs* τ (*f* $ *Bv 0*)) *g*
using *1 eq*(*1*) *g-ty ok'*(*1*) *ok*(*1*) *ok*(*2*) *proves-eq-transitive-rule*[*OF thy - - - - - - - ctxt*]
typ-of-eta-expand[*OF eq*(*2*)] *eq*(*2*) **by** (*fastforce simp add*: *simp'*(*1*))

show *?thesis*
using *tr1 s2 proves-eq-transitive-rule*[*OF thy ok'*(*1*) *ok*(*2*) *ok'*(*2*)] *typ-of-eta-expand eq*(*2*) *g-ty*
ctxt
by *simp*
qed

lemma *proves-inst-bound-rule*:
assumes *thy*: *wf-theory* Θ
assumes *ctxt*: *finite* Γ ∀ *A*∈Γ . *term-ok* Θ *A* ∀ *A*∈Γ . *typ-of A* = *Some propT*
assumes *eq*: Θ, Γ ⊢ *mk-eq* (*Abs* τ *f*) (*Abs* τ *g*) *typ-of* (*Abs* τ *f*) = *Some* (τ → τ′)
assumes *x*: *term-ok* Θ *x typ-of x* = *Some* τ
assumes *ctxt*: *finite* Γ ∀ *A*∈Γ. *term-ok* Θ *A* ∀ *A*∈Γ. *typ-of A* = *Some propT*
shows Θ, Γ ⊢ *mk-eq* (*subst-bv x f*) (*subst-bv x g*)
proof−
have *term-ok* Θ (*mk-eq* (*Abs* τ *f*) (*Abs* τ *g*))
using *eq*(*1*) *proved-terms-well-formed*(*2*) **by** *blast*
hence *term-ok* Θ (*Abs* τ *f*) *term-ok* Θ (*Abs* τ *g*)
using *term-ok-mk-eqD* **by** *blast*+
hence *typ-of* (*Abs* τ *f*) = *typ-of* (*Abs* τ *g*)
using *thy* ‹*term-ok* Θ (*mk-eq* (*Abs* τ *f*) (*Abs* τ *g*))› **by** (*cases* Θ *rule*: *theory-full-exhaust*)
(*auto simp add*: *tinstT-def typ-of-def wt-term-def bind-eq-Some-conv*)
hence *typ-of* (*Abs* τ *g*) = *Some* (τ → τ′)
using *eq*(*2*) **by** *simp*

have Θ, Γ ⊢ *mk-eq x x*
by (*simp add*: *ctxt proves-eq-reflexive thy x*(*1*) *del*: *term-ok-def*)
hence *1*: Θ, Γ ⊢ *mk-eq* (*Abs* τ *f* $ *x*) (*Abs* τ *g* $ *x*)

using *proves-eq-combination-rule*[*OF thy* ‹*term-ok* Θ (*Abs* τ *f*)› *eq*(*2*) ‹*term-ok* Θ (*Abs* τ *g*)›
‹*typ-of* (*Abs* τ *g*) = *Some* (τ → τ′)› *x x eq*(*1*) - *ctxt*]
by *blast*

have Θ, Γ ⊢ *mk-eq* (*Abs* τ *f* $ *x*) (*subst-bv x f*)
apply (*rule* β-*conversion*)
using *thy x* ‹*term-ok* Θ (*Abs* τ *f*)› **by** (*simp-all add*: *wt-term-def*)

have *term-ok* Θ (*Abs* τ *f* $ *x*) **using** ‹*term-ok* Θ (*Abs* τ *f*)› *x*
‹Θ,Γ ⊢ *mk-eq* (*Abs* τ *f* $ *x*) (*Abs* τ *g* $ *x*)› *proved-terms-well-formed*(*1*)
wt-term-def typ-of1-split-App-obtains typ-of-def
by (*meson proved-terms-well-formed*(*2*) *term-ok-mk-eqD*)
have *term-ok* Θ (*Abs* τ *g* $ *x*) **using** ‹*term-ok* Θ (*Abs* τ *g*)› *x*
‹Θ,Γ ⊢ *mk-eq* (*Abs* τ *f* $ *x*) (*Abs* τ *g* $ *x*)› *proved-terms-well-formed*(*1*)
wt-term-def typ-of1-split-App-obtains typ-of-def
by (*meson proved-terms-well-formed*(*2*) *term-ok-mk-eqD*)

have *typ-of* (*subst-bv x f*) = *Some* τ′
using ‹*typ-of* (*Abs* τ *f*) = *Some* (τ → τ′)› *x*(*2*) *typ-of-def typ-of-betapply* **by** *auto*
moreover have *term-ok*′ (*sig* Θ) (*subst-bv x f*)
using ‹*term-ok* Θ (*Abs* τ *f*)› *substn-subst-0*′ *term-ok*′*-subst-bv2 wt-term-def x*(*1*) **by** *auto*
ultimately have *term-ok* Θ (*subst-bv x f*)
by (*simp add*: *wt-term-def*)

have *typ-of* (*Abs* τ *f* $ *x*) = *typ-of* (*subst-bv x f*)
using ‹*typ-of* (*Abs* τ *f*) = *typ-of* (*Abs* τ *g*)› *typ-of-def* ‹*typ-of* (*Abs* τ *g*) = *Some* (τ → τ′)›
‹*typ-of* (*subst-bv x f*) = *Some* τ′› *typ-of-Abs-body-typ*′ *x*(*2*) **by** *fastforce*

have *typ-of* (*Abs* τ *f* $ *x*) = *typ-of* (*Abs* τ *g* $ *x*)
using ‹*typ-of* (*Abs* τ *f*) = *typ-of* (*Abs* τ *g*)› *typ-of-def* **by** *auto*

have *2*: Θ, Γ ⊢ *mk-eq* (*subst-bv x f*) (*Abs* τ *f* $ *x*)
apply − **apply** (*rule proves-eq-symmetric-rule*)
using *thy* **apply** *blast*
using ‹*term-ok* Θ (*subst-bv x f*)› **apply** *blast*
using ‹*term-ok* Θ (*Abs* τ *f* $ *x*)› **apply** *blast*
using ‹*typ-of* (*Abs* τ *f* $ *x*) = *typ-of* (*subst-bv x f*)› **apply** *blast*
using ‹Θ,Γ ⊢ *mk-eq* (*Abs* τ *f* $ *x*) (*subst-bv x f*)› **apply** *blast*
using *ctxt* **by** *blast*+

have *3*: Θ, Γ ⊢ *mk-eq* (*Abs* τ *g* $ *x*) (*subst-bv x g*)
apply (*rule* β-*conversion*)
using *thy x* ‹*term-ok* Θ (*Abs* τ *g*)› **by** (*simp-all add*: *wt-term-def*)

have *term-ok* Θ (*subst-bv x g*)

using ‹*term-ok* Θ (*Abs* τ *g* $ *x*)› ‹*term-ok* Θ (*Abs* τ *g*)› ‹*typ-of* (*Abs* τ *f* $ *x*) = *typ-of* (*Abs* τ *g* $ *x*)›
‹*typ-of* (*Abs* τ *f* $ *x*) = *typ-of* (*subst-bv x f*)› ‹*typ-of* (*Abs* τ *g*) = *Some* (τ → τ′)›
‹*typ-of* (*subst-bv x f*) = *Some* τ′› *betapply.simps*(*1*) *subst-bv-def term-ok′.simps*(*5*)
term-ok′-subst-bv1 wt-term-def typ-of-betaply x(*1*) *x*(*2*)
by (*meson 3 proved-terms-well-formed*(*2*) *term-ok-mk-eqD*)

have *typ-of* (*subst-bv x f*) = *typ-of* (*Abs* τ *g* $ *x*)
using ‹*typ-of* (*Abs* τ *f* $ *x*) = *typ-of* (*Abs* τ *g* $ *x*)›
‹*typ-of* (*Abs* τ *f* $ *x*) = *typ-of* (*subst-bv x f*)› **by** *auto*

have *typ-of* (*Abs* τ *g* $ *x*) = *typ-of* (*subst-bv x g*)
using ‹*typ-of* (*Abs* τ *f*) = *typ-of* (*Abs* τ *g*)› *eq*(*2*) *typ-of-betapply typ-of-def x*(*2*) **by** *auto*

have *c1*: Θ, Γ ⊢ *mk-eq* (*subst-bv x f*) (*Abs* τ *g* $ *x*)
apply (*rule proves-eq-transitive-rule*[**where** *t*=*Abs* τ *f* $ *x*];
(*use assms 1 2* ‹*term-ok* Θ (*subst-bv x f*)› **in** ‹*solves simp*›)*?*)
using ‹*term-ok* Θ (*Abs* τ *f* $ *x*)› **apply** *blast*
using ‹*term-ok* Θ (*Abs* τ *g* $ *x*)› **apply** *blast*
using ‹*typ-of* (*Abs* τ *f* $ *x*) = *typ-of* (*subst-bv x f*)› **apply** *simp*
using ‹*typ-of* (*Abs* τ *f* $ *x*) = *typ-of* (*Abs* τ *g* $ *x*)› **apply** *blast*
done
show *?thesis*
apply (*rule proves-eq-transitive-rule*[**where** *t*=*Abs* τ *g* $ *x*];
(*use assms 1 2* ‹*term-ok* Θ (*subst-bv x f*)› **in** ‹*solves simp*›)*?*)
using ‹*term-ok* Θ (*Abs* τ *g* $ *x*)›
‹*term-ok* Θ (*subst-bv x g*)›
‹*typ-of* (*subst-bv x f*) = *typ-of* (*Abs* τ *g* $ *x*)›
‹*typ-of* (*Abs* τ *g* $ *x*) = *typ-of* (*subst-bv x g*)›
‹Θ,Γ ⊢ *mk-eq* (*subst-bv x f*) (*Abs* τ *g* $ *x*)›
‹Θ,Γ ⊢ *mk-eq* (*Abs* τ *g* $ *x*) (*subst-bv x g*)› **by** *simp-all*
qed

lemma *proves-descend-abs-rule*:
assumes *thy*: *wf-theory* Θ
assumes *eq*: Θ, Γ ⊢ *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′) ∉ *FV* Γ *typ-ok* Θ τ′
assumes *ctxt*: *finite* Γ ∀ *A*∈Γ. *term-ok* Θ *A* ∀ *A*∈Γ. *typ-of A* = *Some propT*
shows Θ, Γ ⊢ *mk-eq s t*
proof−
have *abs-ok*: *term-ok* Θ (*Abs-fv x* τ′ *s*) *term-ok* Θ (*Abs-fv x* τ′ *t*)
using *eq proved-terms-well-formed wt-term-def typ-of1-split-App typ-of-def*
by (*meson term-ok-mk-eqD*)+
obtain τ **where** τ*1*: *typ-of* (*Abs-fv x* τ′ *s*) = *Some* (τ′ → τ)
by (*smt eq proved-terms-well-formed-pre typ-of1-split-App-obtains typ-of-Abs-body-typ′*

typ-of-def)
hence *τ2*: *typ-of* (*Abs-fv x τ′ t*) = *Some* (*τ′* → *τ*)
by (*metis eq*(*1*) *proved-terms-well-formed*(*2*) *term-ok-mk-eq-same-typ thy*)

have *add-param*: Θ, Γ ⊢ *mk-eq*
(*Abs τ′* (*bind-fv* (*x*, *τ′*) *s*) $ *Fv x τ′*)
(*Abs τ′* (*bind-fv* (*x*, *τ′*) *t*) $ *Fv x τ′*)
apply(*rule proves-eq-combination-rule*; *use assms abs-ok τ1 τ2* **in** ‹(*solves simp*) *?*›)
using *proves-eq-reflexive term-ok-var thy x*(*2*) *wt-term-def ctxt* **by** *blast*+

have *βs*: Θ, Γ ⊢ *mk-eq*
(*Abs τ′* (*bind-fv* (*x*, *τ′*) *s*) $ *Fv x τ′*)
(*subst-bv* (*Fv x τ′*) (*bind-fv* (*x*, *τ′*) *s*))
by (*rule proves.β-conversion*; *use assms abs-ok τ1 τ2* **in** ‹(*solves* ‹*simp add*: *wt-term-def*›) *?*›)
moreover have *simps*: *subst-bv* (*Fv x τ′*) (*bind-fv* (*x*, *τ′*) *s*) = *s*
using *subst-bv-bind-fv typ-of-imp-closed eq*(*2*) **by** *blast*
ultimately have *βs*: Θ, Γ ⊢ *mk-eq* (*Abs τ′* (*bind-fv* (*x*, *τ′*) *s*) $ *Fv x τ′*) *s*
by *simp*

have *t1*: *term-ok* Θ *s*
using *βs proved-terms-well-formed*(*2*) *wt-term-def typ-of-def*
using *term-ok-app-eqD* **by** *blast*
have *t2*: *term-ok* Θ (*Abs-fv x τ′ s* $ *term.Fv x τ′*)
using *βs* ‹*term-ok* Θ *s*› *proved-terms-well-formed*(*2*) *term-ok′.simps*(*4*)
wt-term-def term-ok-mk-eq-same-typ thy
by (*meson term-ok-mk-eqD*)

have *βs-rev*: Θ, Γ ⊢ *mk-eq s* (*Abs τ′* (*bind-fv* (*x*, *τ′*) *s*) $ *Fv x τ′*)
apply (*rule proves-eq-symmetric-rule*; *use assms abs-ok τ1 τ2 t1 t2* **in** ‹(*solves simp*) *?*›)
using *βs proved-terms-well-formed*(*2*) *term-ok-mk-eq-same-typ thy* **apply** *blast*
using *βs* **by** *simp*

have *βt*: Θ, Γ ⊢ *mk-eq*
(*Abs τ′* (*bind-fv* (*x*, *τ′*) *t*) $ *Fv x τ′*)
(*subst-bv* (*Fv x τ′*) (*bind-fv* (*x*, *τ′*) *t*))
by (*rule proves.β-conversion*; *use assms abs-ok τ1 τ2 t1 t2* **in** ‹(*solves* ‹*simp add*: *wt-term-def*›) *?*›)
moreover have *simpt*: *subst-bv* (*Fv x τ′*) (*bind-fv* (*x*, *τ′*) *t*) = *t*
using *subst-bv-bind-fv typ-of-imp-closed eq*(*3*) **by** *blast*
ultimately have *βt*: Θ, Γ ⊢ *mk-eq* (*Abs τ′* (*bind-fv* (*x*, *τ′*) *t*) $ *Fv x τ′*) *t*
by *simp*

have *t3*: *term-ok* Θ (*Abs-fv x τ′ t* $ *term.Fv x τ′*)
using *βs add-param proved-terms-well-formed*(*2*) *t1 term-ok′.simps*(*4*)
wt-term-def term-ok-mk-eq-same-typ thy
by (*meson term-ok-mk-eqD*)

have *t4*: *typ-of s* = *typ-of* (*Abs-fv x τ′ t* $ *term.Fv x τ′*)
by (*metis βs add-param proved-terms-well-formed*(*2*) *term-ok-mk-eq-same-typ thy*)
have *t5*: *typ-of s* = *typ-of* (*Abs-fv x τ′ s* $ *Fv x τ′*)
using *βs-rev proved-terms-well-formed*(*2*) *term-ok-mk-eq-same-typ thy* **by** *blast*
have *t6*: *typ-of* (*Abs-fv x τ′ s* $ *Fv x τ′*) = *typ-of* (*Abs-fv x τ′ t* $ *term.Fv x τ′*)
using *t4 t5* **by** *auto*
have *half*: Θ, Γ ⊢ *mk-eq s* (*Abs τ′* (*bind-fv* (*x*, *τ′*) *t*) $ *Fv x τ′*)
apply (*rule proves-eq-transitive-rule*[**where** *t*=*Abs τ′* (*bind-fv* (*x*, *τ′*) *s*) $ *Fv x τ′*]
; *use assms abs-ok τ1 τ2 t1 t2 t3 t4 t5 t6* **in** ‹(*solves simp*)*?*›)
using *βs-rev* **apply** *blast*
using *add-param* **by** *blast*

have *t7*: *term-ok* Θ *t*
using *βt proved-terms-well-formed*(*2*) *t1 t4 term-ok′.simps*(*4*) *wt-term-def term-ok-mk-eq-same-typ thy*
by (*meson term-ok-app-eqD*)
have *t8*: *typ-of* (*Abs-fv x τ′ t* $ *term.Fv x τ′*) = *typ-of t*
using *βt proved-terms-well-formed*(*2*) *term-ok-mk-eq-same-typ thy* **by** *blast*

show *?thesis*
apply (*rule proves-eq-transitive-rule*[**where** *t*=*Abs τ′* (*bind-fv* (*x*, *τ′*) *t*) $ *Fv x τ′*]
; *use assms abs-ok τ1 τ2 t1 t2 t3 t4 t5 t6 t7 t8* **in** ‹(*solves simp*)*?*›)
using *half* **apply** *blast*
using *βt* **by** *blast*
qed

lemma *obtain-fresh-variable*:
assumes *finite* Γ
obtains *x* **where** (*x*,*τ*) ∉ *fv t* ∪ *FV* Γ
using *assms finite-fv finite-FV*
by (*metis finite-Un finite-imageI fst-conv image-eqI variant-variable-fresh*)
lemma *obtain-fresh-variable′*:
assumes *finite* Γ
obtains *x* **where** (*x*,*τ*) ∉ *fv t* ∪ *fv u* ∪ *FV* Γ
using *assms finite-fv finite-FV*
by (*metis finite-Un finite-imageI fst-conv image-eqI variant-variable-fresh*)

lemma *proves-eq-abstract-rule-pre*:
assumes *thy*: *wf-theory* Θ
assumes *A*: *term-ok* Θ *f typ-of f* = *Some* (*τ* → *τ′*)
assumes *B*: *term-ok* Θ *g typ-of g* = *Some* (*τ* → *τ′*)
shows Θ, {} ⊢ (*Ct STR ''Pure.all''* ((*τ* → *propT*) → *propT*) $ *Abs τ* (*mk-eq′ τ′* (*f* $ *Bv 0*) (*g* $ *Bv 0*)))
⟼ *mk-eq* (*Abs τ* (*f* $ *Bv 0*)) (*Abs τ* (*g* $ *Bv 0*))
proof–

have *eq-abstract-rule-ax* ∈ *axioms* Θ
using *thy* **by** (*cases* Θ *rule*: *theory-full-exhaust*) *auto*
moreover have *ok2*: *typ-ok* Θ (τ → τ′)
using *assms*(*2*) *assms*(*3*) *term-ok-imp-typ-ok thy* **by** *blast*
moreover hence *ok3*: *typ-ok* Θ τ′
using *thy A*(*2*) **by** (*cases* Θ *rule*: *theory-full-exhaust*) *auto*
moreover have *ok1*: *typ-ok* Θ τ
using *thy A*(*2*) *ok2* **by** (*cases* Θ *rule*: *theory-full-exhaust*) *auto*
ultimately have *1*: Θ, {} ⊢ *subst-typ′*
[((*Var* (*STR* ′′′*a*′′, *0*), *full-sort*), τ), ((*Var* (*STR* ′′′*b*′′, *0*), *full-sort*), τ′)]
eq-abstract-rule-ax
using *assms axiom-subst-typ′* **by** (*simp del*: *term-ok-def*)
hence Θ, {} ⊢ *subst-term* [((*Var* (*STR* ′′*g*′′, *0*), τ → τ′), *g*),
((*Var* (*STR* ′′*f*′′, *0*), τ → τ′), *f*)] (*subst-typ′*
[((*Var* (*STR* ′′′*a*′′, *0*), *full-sort*), τ), ((*Var* (*STR* ′′′*b*′′, *0*), *full-sort*), τ′)]
eq-abstract-rule-ax)
using *ok1 ok2 ok3 assms term-ok-var* **by** (*fastforce intro*!: *inst-var-multiple simp add*: *eq-abstract-rule-ax-def*)
moreover have *subst-term* [((*Var* (*STR* ′′*g*′′, *0*), τ → τ′), *g*),
((*Var* (*STR* ′′*f*′′, *0*), τ → τ′), *f*)] (*subst-typ′*
[((*Var* (*STR* ′′′*a*′′, *0*), *full-sort*), τ), ((*Var* (*STR* ′′′*b*′′, *0*), *full-sort*), τ′)]
eq-abstract-rule-ax)
= (*Ct STR* ′′*Pure.all*′′ ((τ → *propT*) → *propT*) $ *Abs* τ (*mk-eq′* τ′ (*f* $ *Bv 0*)
(*g* $ *Bv 0*)))
⟼ *mk-eq* (*Abs* τ (*f* $ *Bv 0*)) (*Abs* τ (*g* $ *Bv 0*))
using *assms typ-of1-weaken-Ts* **by** (*fastforce simp add*: *eq-axs-def typ-of-def*)
ultimately show *?thesis*
using *assms* **by** *simp*
qed

lemma *proves-eq-abstract-rule*:
assumes *thy*: *wf-theory* Θ
assumes *A*: *term-ok* Θ *f typ-of f* = *Some* (τ → τ′)
assumes *B*: *term-ok* Θ *g typ-of g* = *Some* (τ → τ′)
assumes *ctxt*: *finite* Γ ∀ *A*∈Γ. *term-ok* Θ *A* ∀ *A*∈Γ. *typ-of A* = *Some propT*
shows Θ, Γ ⊢ (*Ct STR* ′′*Pure.all*′′ ((τ → *propT*) → *propT*) $ *Abs* τ (*mk-eq′* τ′
(*f* $ *Bv 0*) (*g* $ *Bv 0*)))
⟼ *mk-eq* (*Abs* τ (*f* $ *Bv 0*)) (*Abs* τ (*g* $ *Bv 0*))
by (*subst unsimp-context*) (*use assms proves-eq-abstract-rule-pre weaken-proves-set*
in *blast*)

lemma *proves-eq-abstract-rule-rule*:
assumes *thy*: *wf-theory* Θ
assumes *A*: *term-ok* Θ *f typ-of f* = *Some* (τ → τ′)
assumes *B*: *term-ok* Θ *g typ-of g* = *Some* (τ → τ′)
assumes Θ, Γ ⊢ (*Ct STR* ′′*Pure.all*′′ ((τ → *propT*) → *propT*) $ *Abs* τ (*mk-eq′*
τ′ (*f* $ *Bv 0*) (*g* $ *Bv 0*)))
assumes *ctxt*: *finite* Γ ∀ *A*∈Γ. *term-ok* Θ *A* ∀ *A*∈Γ. *typ-of A* = *Some propT*
shows Θ, Γ ⊢ *mk-eq* (*Abs* τ (*f* $ *Bv 0*)) (*Abs* τ (*g* $ *Bv 0*))

proof−
note *1* = *proves-eq-abstract-rule*[**where** Γ=Γ, *OF assms*(*1*−*5*) *ctxt*]
note *2* = *proves.implies-elim*[*OF 1 assms*(*6*)]
thus *?thesis* **using** *ctxt* **by** *simp*
qed

lemma *proves-eq-ext-rule*:
assumes *thy*: *wf-theory* Θ
assumes *f*: *term-ok* Θ *f typ-of f* = *Some* (τ → τ′)
assumes *g*: *term-ok* Θ *g typ-of g* = *Some* (τ → τ′)
assumes *prem*: Θ, Γ ⊢ *Ct STR* ''*Pure.all*'' ((τ → *propT*) → *propT*) $ *Abs* τ (*mk-eq*′ τ′ (*f* $ *Bv 0*) (*g* $ *Bv 0*))
assumes *ctxt*: *finite* Γ ∀ *A*∈Γ. *term-ok* Θ *A* ∀ *A*∈Γ. *typ-of A* = *Some propT*
shows Θ, Γ ⊢ *mk-eq f g*
proof−
obtain *x* **where** *x*: (*x*,τ) ∉ *FV* Γ (*x*,τ) ∉ *fv f* (*x*,τ) ∉ *fv g*
by (*meson Un-iff ctxt*(*1*) *obtain-fresh-variable*′)
have *closed*: *is-closed f is-closed g*
using *f g has-typ-imp-closed term-ok-def wt-term-def* **by** *blast*+

have *term-ok* Θ (*Abs* τ (*mk-eq*′ τ′ (*f* $ *Bv 0*) (*g* $ *Bv 0*)))
using *prem proved-terms-well-formed*(*2*) *term-ok-app-eqD* **by** *blast*

have *subst-bv* (*Fv x* τ) (*f* $ *Bv 0*) = *f* $ *Fv x* τ
using *Core.subst-bv-def f*(*1*) *term-ok-subst-bv-no-change* **by** *auto*
moreover have *subst-bv* (*Fv x* τ) (*g* $ *Bv 0*) = *g* $ *Fv x* τ
using *Core.subst-bv-def g*(*1*) *term-ok-subst-bv-no-change* **by** *auto*
ultimately have *subst-bv* (*Fv x* τ) (*mk-eq*′ τ′ (*f* $ *Bv 0*) (*g* $ *Bv 0*))
= *mk-eq*′ τ′ (*f* $ *Fv x* τ) (*g* $ *Fv x* τ)
by (*simp add*: *Core.subst-bv-def*)
hence *simp*: *Abs* τ (*mk-eq*′ τ′ (*f* $ *Bv 0*) (*g* $ *Bv 0*)) · *Fv x* τ = *mk-eq* (*f* $ *Fv x* τ) (*g* $ *Fv x* τ)
using *f g* **by** (*auto simp add*: *typ-of-def*)
hence *simp*′: *subst-bv* (*Fv x* τ) (*mk-eq*′ τ′ (*f* $ *Bv 0*) (*g* $ *Bv 0*)) = *mk-eq*′ τ′ (*f* $ *Fv x* τ) (*g* $ *Fv x* τ)
using *f g* **by** (*auto simp add*: *typ-of-def*)

have Θ, Γ ⊢ *mk-eq*′ τ′ (*f* $ *Fv x* τ) (*g* $ *Fv x* τ)
apply (*subst simp*′[*symmetric*])
apply (*rule forall-elim*[**where** τ=τ])
using *prem* **apply** *blast*
apply *simp*
using ‹*term-ok* Θ (*Abs* τ (*mk-eq*′ τ′ (*f* $ *Bv 0*) (*g* $ *Bv 0*)))› *term-ok*′.*simps*(*1*) *term-ok*′.*simps*(*5*) *term-okD1* **by** *blast*
moreover have *typ-of* (*f* $ *Fv x* τ) = *Some* τ′ *typ-of* (*g* $ *Fv x* τ) = *Some* τ′
using *f*(*2*) *g*(*2*) **by** (*simp-all add*: *typ-of-def*)
ultimately have *1*: Θ, Γ ⊢ *mk-eq* (*f* $ *Fv x* τ) (*g* $ *Fv x* τ)
by *simp*
have *core*: Θ, Γ ⊢ *mk-eq* (*Abs* τ (*f* $ *Bv 0*)) (*Abs* τ (*g* $ *Bv 0*))

apply (*rule proves-eq-abstract-rule-rule*[*OF thy f g - ctxt*])
using *prem* **by** *blast*
have Θ, Γ ⊢ *mk-eq* (*Abs τ* (*f* $ *Bv 0*)) *f*
using *f proves.eta term-okD1 thy* **by** *blast*
have *left*: Θ, Γ ⊢ *mk-eq f* (*Abs τ* (*f* $ *Bv 0*))
apply (*rule proves-eq-symmetric-rule*[*OF thy f*(*1*) - - - *ctxt*])
using ‹Θ,Γ ⊢ *mk-eq* (*Abs τ* (*f* $ *Bv 0*)) (*Abs τ* (*g* $ *Bv 0*))› *proved-terms-well-formed*(*2*) *term-ok-mk-eqD* **apply** *blast*
apply (*simp add*: *Logic.typ-of-eta-expand f*(*2*))
using ‹Θ,Γ ⊢ *mk-eq* (*Abs τ* (*f* $ *Bv 0*)) *f*› **by** *blast*

have *right*: Θ, Γ ⊢ *mk-eq* (*Abs τ* (*g* $ *Bv 0*)) *g*
using *g proves.eta term-okD1 thy* **by** *blast*

show *?thesis*
apply (*rule proves-eq-transitive-rule*[**where** *t*=*Abs τ* (*f* $ *Bv 0*), *OF thy f*(*1*) - *g*(*1*) - - *left* - *ctxt*])
using ‹Θ,Γ ⊢ *mk-eq* (*Abs τ* (*f* $ *Bv 0*)) *f*› *proved-terms-well-formed*(*2*) *term-ok-mk-eqD* **apply** *blast*
apply (*simp add*: *Logic.typ-of-eta-expand f*(*2*))
apply (*simp add*: *Logic.typ-of-eta-expand f*(*2*) *g*(*2*))
apply (*rule proves-eq-transitive-rule*[**where** *t*=*Abs τ* (*g* $ *Bv 0*), *OF thy* - - *g*(*1*) - - *core right ctxt*])
using ‹Θ,Γ ⊢ *mk-eq* (*Abs τ* (*f* $ *Bv 0*)) *f*› *proved-terms-well-formed*(*2*) *term-ok-mk-eqD* **apply** *blast*
using ‹Θ,Γ ⊢ *mk-eq* (*Abs τ* (*g* $ *Bv 0*)) *g*› *proved-terms-well-formed*(*2*) *term-ok-mk-eqD* **apply** *blast*
by (*simp add*: *Logic.typ-of-eta-expand f*(*2*) *g*(*2*))+
qed

lemma *bind-fv2-idem*[*simp*]:
bind-fv2 (*x*, *τ*) *lev1* (*bind-fv2* (*x*, *τ*) *lev2 t*) = *bind-fv2* (*x*, *τ*) *lev2 t*
by (*induction* (*x*,*τ*) *lev2 t arbitrary*: *lev1 rule*: *bind-fv2.induct*) *auto*
corollary *bind-fv-idem*[*simp*]:
bind-fv (*x*, *τ*) (*bind-fv* (*x*, *τ*) *t*) = *bind-fv* (*x*, *τ*) *t*
using *bind-fv-def bind-fv2-idem* **by** *simp*
corollary *bind-fv-Abs-fv*[*simp*]: *bind-fv* (*x*, *τ*) (*Abs-fv x τ t*) = *Abs-fv x τ t*
by (*simp add*: *bind-fv-def*)

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*)
by *simp*
lemma *bind-fv* (*x*,*τ*) (*mk-eq′ τ′ s t*) = *mk-eq′ τ′* (*bind-fv* (*x*,*τ*) *s*) (*bind-fv* (*x*,*τ*) *t*)
by (*simp add*: *bind-fv-def*)

lemma *term-ok-Abs-fvI*: *term-ok* Θ *s* ⟹ *typ-ok* Θ *τ* ⟹ *term-ok* Θ (*Abs-fv x τ s*)
by (*auto simp add*: *wt-term-def term-ok′-bind-fv typ-of-Abs-bind-fv*)

lemma *proves-eq-abstract-rule-derived-rule*:
assumes *thy*: *wf-theory* Θ
assumes *x*: (*x*, τ) ∉ *FV* Γ *typ-ok* Θ τ
assumes *ctxt*: *finite* Γ ∀ *A*∈Γ. *term-ok* Θ *A* ∀ *A*∈Γ. *typ-of A* = *Some propT*
assumes *eq*: Θ, Γ ⊢ *mk-eq s t*
shows Θ, Γ ⊢ *mk-eq* (*Abs* τ (*bind-fv* (*x*, τ) *s*)) (*Abs* τ (*bind-fv* (*x*, τ) *t*))
proof−
obtain τ′ **where** *s*: *typ-of s* = *Some* τ′
by (*meson eq option.exhaust-sel proved-terms-well-formed*(*2*) *term-okD2 term-ok-app-eqD*)
have *t*: *typ-of t* = *Some* τ′
by (*metis eq proved-terms-well-formed*(*2*) *s term-ok-mk-eq-same-typ thy*)

have *ok*: *term-ok* Θ *s term-ok* Θ *t*
using *eq proved-terms-well-formed*(*2*) *term-ok-mk-eqD* **by** *blast*+

have *closed*: *is-closed s is-closed t*
using *eq has-typ-imp-closed proved-terms-well-formed*(*2*) *term-ok-def term-ok-mk-eqD wt-term-def* **by** *blast*+

have *is-closed* (*mk-eq s t*)
using *eq proved-terms-closed* **by** *blast*
hence *Abs* τ (*bind-fv* (*x*, τ) (*mk-eq s t*)) · *Fv x* τ = *mk-eq s t*
using *betapply-Abs-fv* **by** *auto*
have Θ, Γ ⊢ *mk-all x* τ (*mk-eq s t*)
using *eq forall-intro thy typ-ok-def x*(*1*) *x*(*2*) **by** *blast*

have Θ, Γ ⊢ *mk-eq* (*Abs* τ (*bind-fv* (*x*, τ) *s*) $ *Fv x* τ) (*subst-bv* (*Fv x* τ) (*bind-fv* (*x*, τ) *s*))
using *term-ok-Abs-fvI*[*OF ok*(*1*) *x*(*2*)] *wf-term.intros*(*1*) *typ-ok-def x*(*2*)
by (*auto intro*!: *β-conversion*[*OF thy*])
moreover have *subst-bv* (*Fv x* τ) (*bind-fv* (*x*, τ) *s*) = *s*
by (*simp add*: *closed*(*1*) *subst-bv-bind-fv*)
ultimately have *unfs*: Θ, Γ ⊢ *mk-eq* (*Abs* τ (*bind-fv* (*x*, τ) *s*) $ *Fv x* τ) *s*
by *simp*
have Θ, Γ ⊢ *mk-eq* (*Abs* τ (*bind-fv* (*x*, τ) *t*) $ *Fv x* τ) (*subst-bv* (*Fv x* τ) (*bind-fv* (*x*, τ) *t*))
using *term-ok-Abs-fvI*[*OF ok*(*2*) *x*(*2*)] *wf-term.intros*(*1*) *typ-ok-def x*(*2*)
by (*auto intro*!: *β-conversion*[*OF thy*])

moreover have *subst-bv* (*Fv x* τ) (*bind-fv* (*x*, τ) *t*) = *t*
by (*simp add*: *closed*(*2*) *subst-bv-bind-fv*)
ultimately have *unft*: Θ, Γ ⊢ *mk-eq* (*Abs* τ (*bind-fv* (*x*, τ) *t*) $ *Fv x* τ) *t*
by *simp*

have *prem*:
Θ, Γ ⊢ *mk-eq* (*Abs* τ (*bind-fv* (*x*, τ) *s*) $ *Fv x* τ) (*Abs* τ (*bind-fv* (*x*, τ) *t*) $ *Fv x* τ)
apply (*rule proves-eq-transitive-rule*[**where** *t*=*s*, *OF thy - - - - - - - ctxt*])
using *ok*(*1*) *term-ok-mk-eqD unfs unft proved-terms-well-formed*(*2*) *term-ok-mk-eq-same-typ*

thy
apply (*all blast*)[*4*]
apply (*metis proved-terms-well-formed*(*2*) *s t term-ok-mk-eq-same-typ thy unft*)
using *unfs* **apply** *blast*
subgoal
apply (*rule proves-eq-transitive-rule*[**where** *t*=*t*, *OF thy ok - - - - - ctxt*])
using *proved-terms-well-formed*(*2*) *term-ok-mk-eqD unft* **apply** *blast*
apply (*simp add*: *s t*)
apply (*metis proved-terms-well-formed*(*2*) *term-ok-mk-eq-same-typ thy unft*)
using *eq* **apply** *simp*
subgoal apply (*rule proves-eq-symmetric-rule*[*OF thy ok*(*2*) *- - - ctxt*])
using *proved-terms-well-formed*(*2*) *term-ok-mk-eqD unft* **apply** *blast*
using *proved-terms-well-formed*(*2*) *term-ok-mk-eq-same-typ thy unft* **apply** *blast*
using *unft* **apply** *blast*
done
done
done
hence Θ, Γ ⊢ *mk-all x τ*
(*mk-eq* (*Abs τ* (*bind-fv* (*x*, *τ*) *s*) $ *Fv x τ*) (*Abs τ* (*bind-fv* (*x*, *τ*) *t*) $ *Fv x τ*))
using *forall-intro thy typ-ok-def x*(*1*) *x*(*2*) **by** *blast*
moreover have *mk-all x τ*
(*mk-eq* (*Abs τ* (*bind-fv* (*x*, *τ*) *s*) $ *Fv x τ*) (*Abs τ* (*bind-fv* (*x*, *τ*) *t*) $ *Fv x τ*))
= *mk-all x τ*
(*mk-eq*′ *τ*′ (*Abs τ* (*bind-fv* (*x*, *τ*) *s*) $ *Fv x τ*) (*Abs τ* (*bind-fv* (*x*, *τ*) *t*) $ *Fv x τ*))
using *bind-fv2-preserves-type s t typ-of-def* **by** (*fastforce simp add*: *bind-fv-def typ-of-def*)+
moreover have *mk-all x τ*
(*mk-eq*′ *τ*′ (*Abs τ* (*bind-fv* (*x*, *τ*) *s*) $ *Fv x τ*) (*Abs τ* (*bind-fv* (*x*, *τ*) *t*) $ *Fv x τ*)) =
Ct STR ″*Pure.all*″ ((*τ* → *propT*) → *propT*) $ *Abs τ*
(*mk-eq*′ *τ*′ (*Abs τ* (*bind-fv* (*x*, *τ*) *s*) $ *Bv 0*) (*Abs τ* (*bind-fv* (*x*, *τ*) *t*) $ *Bv 0*))
by (*simp add*: *bind-fv-def*)
ultimately have *pre-ext*: Θ, Γ ⊢ *Ct STR* ″*Pure.all*″ ((*τ* → *propT*) → *propT*) $ *Abs τ*
(*mk-eq*′ *τ*′ (*Abs τ* (*bind-fv* (*x*, *τ*) *s*) $ *Bv 0*) (*Abs τ* (*bind-fv* (*x*, *τ*) *t*) $ *Bv 0*))
by *simp*
show *?thesis*
apply (*rule proves-eq-ext-rule*[**where** *τ*=*τ* **and** *τ*′=*τ*′, *OF thy - - - - - ctxt*])
using *proved-terms-well-formed*(*2*) *term-ok-app-eqD unfs* **apply** *blast*
apply (*simp add*: *s typ-of-Abs-bind-fv*)
using *proved-terms-well-formed*(*2*) *term-ok-app-eqD unft* **apply** *blast*
apply (*simp add*: *t typ-of-Abs-bind-fv*)
using *pre-ext* **by** *blast*
qed

lemma *proves-descend-abs-rule-iff*:

assumes *thy*: *wf-theory* Θ
assumes *ok*: *is-closed s is-closed t*
assumes *x*: $(x, \tau') \notin FV\ \Gamma\ typ\text{-}ok\ \Theta\ \tau'$
assumes *ctxt*: *finite* $\Gamma\ \forall A{\in}\Gamma.\ term\text{-}ok\ \Theta\ A\ \forall A{\in}\Gamma.\ typ\text{-}of\ A = Some\ propT$
shows $\Theta, \Gamma \vdash mk\text{-}eq\ s\ t$
$\longleftrightarrow \Theta, \Gamma \vdash mk\text{-}eq\ (Abs\ \tau'\ (bind\text{-}fv\ (x, \tau')\ s))\ (Abs\ \tau'\ (bind\text{-}fv\ (x, \tau')\ t))$
proof (*rule iffI*)
assume *asm*: $\Theta,\Gamma \vdash mk\text{-}eq\ s\ t$
hence *term-ok* Θ *s term-ok* Θ *t*
using *proved-terms-well-formed*(*2*) *term-ok-mk-eqD* **by** *blast*+
show $\Theta,\Gamma \vdash mk\text{-}eq\ (Abs\text{-}fv\ x\ \tau'\ s)\ (Abs\text{-}fv\ x\ \tau'\ t)$
by (*rule proves-eq-abstract-rule-derived-rule*[*OF thy x ctxt asm*])
next
assume *asm*: $\Theta,\Gamma \vdash mk\text{-}eq\ (Abs\text{-}fv\ x\ \tau'\ s)\ (Abs\text{-}fv\ x\ \tau'\ t)$
show $\Theta,\Gamma \vdash mk\text{-}eq\ s\ t$
using *assms asm proves-descend-abs-rule* **by** *blast*
qed

lemma *proves-descend-abs-rule'*:
assumes *thy*: *wf-theory* Θ
assumes *eq*: $\Theta, \Gamma \vdash mk\text{-}eq\ (Abs\ \tau'\ s)\ (Abs\ \tau'\ t)$
assumes *x*: $(x, \tau') \notin FV\ \Gamma\ typ\text{-}ok\ \Theta\ \tau'$
assumes *ctxt*: *finite* $\Gamma\ \forall A{\in}\Gamma.\ term\text{-}ok\ \Theta\ A\ \forall A{\in}\Gamma.\ typ\text{-}of\ A = Some\ propT$
shows $\Theta, \Gamma \vdash mk\text{-}eq\ (subst\text{-}bv\ (Fv\ x\ \tau')\ s)\ (subst\text{-}bv\ (Fv\ x\ \tau')\ t)$
proof−
have *abs-ok*: *term-ok* Θ $(Abs\ \tau'\ s)$ *term-ok* Θ $(Abs\ \tau'\ t)$
using *eq*(*1*) *option.distinct*(*1*) *proved-terms-well-formed term-ok'.simps*(*4*)
wt-term-def typ-of1-split-App typ-of-def
by (*smt term-ok-mk-eqD*)+

obtain τ **where** $\tau 1$: *typ-of* $(Abs\ \tau'\ s) = Some\ (\tau' \to \tau)$
by (*smt eq proved-terms-well-formed-pre typ-of1-split-App-obtains typ-of-Abs-body-typ' typ-of-def*)
hence $\tau 2$: *typ-of* $(Abs\ \tau'\ t) = Some\ (\tau' \to \tau)$
by (*metis eq*(*1*) *proved-terms-well-formed*(*2*) *term-ok-mk-eq-same-typ thy*)

have *add-param*: $\Theta, \Gamma \vdash mk\text{-}eq$
$(Abs\ \tau'\ s\ \$\ Fv\ x\ \tau')$
$(Abs\ \tau'\ t\ \$\ Fv\ x\ \tau')$
apply (*rule proves-eq-combination-rule*; *use assms abs-ok* $\tau 1\ \tau 2$ **in** ‹(*solves* ‹*simp del*: *term-ok-def*›)*?*›)
using *proves-eq-reflexive term-ok-var thy x*(*2*) *ctxt* **by** *blast*

have βs: $\Theta, \Gamma \vdash mk\text{-}eq$
$(Abs\ \tau'\ s\ \$\ Fv\ x\ \tau')$
$(subst\text{-}bv\ (Fv\ x\ \tau')\ s)$
by (*rule proves.*β*-conversion*; *use assms abs-ok* $\tau 1\ \tau 2$ **in** ‹(*solves* ‹*simp add*: *wt-term-def*›)*?*›)

have *t1*: *term-ok* Θ (*subst-bv* (*Fv x* τ′) *s*)
using β*s proved-terms-well-formed*(*2*) *wt-term-def typ-of-def*
using *term-ok-mk-eqD* **by** *blast*
have *t2*: *term-ok* Θ (*Abs* τ′ *s* $ *term.Fv x* τ′)
using β*s proved-terms-well-formed*(*2*) *t1 term-ok′.simps*(*4*) *wt-term-def term-ok-mk-eq-same-typ thy*
term-ok-mk-eqD **by** *blast*
have β*s-rev*: Θ, Γ ⊢ *mk-eq* (*subst-bv* (*Fv x* τ′) *s*) (*Abs* τ′ *s* $ *Fv x* τ′)
apply (*rule proves-eq-symmetric-rule*; *use assms abs-ok* τ*1* τ*2 t1 t2* **in** ‹(*solves simp*) *?*›)
using β*s proved-terms-well-formed*(*2*) *term-ok-mk-eq-same-typ thy* **apply** *blast*
using β*s* **by** *simp*

have β*t*: Θ, Γ ⊢ *mk-eq*
(*Abs* τ′ *t* $ *Fv x* τ′)
(*subst-bv* (*Fv x* τ′) *t*)
by (*rule proves.*β*-conversion*; *use assms abs-ok* τ*1* τ*2 t1* **in** ‹(*solves* ‹*simp add*: *wt-term-def*›) *?*›)

have *t3*: *term-ok* Θ (*Abs* τ′ *t* $ *term.Fv x* τ′)
using β*s add-param proved-terms-well-formed*(*2*) *t1 term-ok′.simps*(*4*)
wt-term-def term-ok-mk-eq-same-typ thy term-ok-mk-eqD
by *meson*
have *t4*: *typ-of* (*subst-bv* (*Fv x* τ′) *s*) = *typ-of* (*Abs* τ′ *t* $ *term.Fv x* τ′)
by (*metis* β*s add-param proved-terms-well-formed*(*2*) *term-ok-mk-eq-same-typ thy*)
have *t5*: *typ-of* (*subst-bv* (*Fv x* τ′) *s*) = *typ-of* (*Abs* τ′ *s* $ *Fv x* τ′)
using β*s-rev proved-terms-well-formed*(*2*) *term-ok-mk-eq-same-typ thy* **by** *blast*
have *t6*: *typ-of* (*Abs* τ′ *s* $ *Fv x* τ′) = *typ-of* (*Abs* τ′ *t* $ *term.Fv x* τ′)
using *t4 t5* **by** *auto*

have *half*: Θ, Γ ⊢ *mk-eq* (*subst-bv* (*Fv x* τ′) *s*) (*Abs* τ′ *t* $ *Fv x* τ′)
apply (*rule proves-eq-transitive-rule*[**where** *t*=*Abs* τ′ *s* $ *Fv x* τ′]
; *use assms abs-ok* τ*1* τ*2 t1 t2 t3 t4 t5 t6* **in** ‹(*solves simp*) *?*›)
using β*s-rev* **apply** *blast*
using *add-param* **by** *blast*

have *t7*: *term-ok* Θ (*subst-bv* (*Fv x* τ′) *t*)
using β*t proved-terms-well-formed*(*2*) *t1 t4 term-ok′.simps*(*4*) *wt-term-def term-ok-mk-eq-same-typ thy*
by (*meson term-ok-app-eqD*)
have *t8*: *typ-of* (*Abs* τ′ *t* $ *term.Fv x* τ′) = *typ-of* (*subst-bv* (*Fv x* τ′) *t*)
using β*t proved-terms-well-formed*(*2*) *term-ok-mk-eq-same-typ thy* **by** *blast*

show *?thesis*
apply (*rule proves-eq-transitive-rule*[**where** *t*=*Abs* τ′ *t* $ *Fv x* τ′]
; *use assms abs-ok* τ*1* τ*2 t1 t2 t3 t4 t5 t6 t7 t8* **in** ‹(*solves simp*) *?*›)
using *half* **apply** *blast*

```
    using βt by blast
qed

lemma proves-ascend-abs-rule':
  assumes thy: wf-theory Θ
  assumes x: (x, τ') ∉ FV Γ (x,τ') ∉ fv (mk-eq (Abs τ' s) (Abs τ' t)) typ-ok Θ τ'
  assumes eq: Θ, Γ ⊢ mk-eq (subst-bv (Fv x τ') s) (subst-bv (Fv x τ') t)
  assumes ctxt: finite Γ ∀ A∈Γ. term-ok Θ A ∀ A∈Γ. typ-of A = Some propT
  shows Θ, Γ ⊢ mk-eq (Abs τ' s) (Abs τ' t)
proof–
  have ok-ind: wf-type (sig Θ) τ'
    using x(3) by simp


  note 1 = proves-eq-abstract-rule-derived-rule[OF thy]
  have term-ok Θ (subst-bv (Fv x τ') s)
    using eq proved-terms-well-formed(2) wt-term-def typ-of-def
    by (meson term-ok-app-eqD)
  hence is-closed (subst-bv (Fv x τ') s)
    using wt-term-def typ-of-imp-closed by auto
  hence loose-s: ¬ loose-bvar s 1
    using is-closed-subst-bv by simp
  hence loose-s': (⋀x. 1 < x ⟹ ¬ loose-bvar1 s x)
    by (simp add: not-loose-bvar-imp-not-loose-bvar1-all-greater)
  moreover have  ¬ occs (case-prod Fv (x,τ')) s
  proof–
    have (x,τ') ∉ fv s
      using x(2) by auto
    thus ?thesis
      by (simp add: fv-iff-occs)
  qed
  ultimately have s: Abs-fv x τ' (subst-bv (term.Fv x τ') s) = Abs τ' s
    unfolding subst-bv-def bind-fv-def
      using bind-fv2-subst-bv1-cancel
      by (metis (full-types) case-prod-conv less-one linorder-neqE-nat
          loose-bvar1-imp-loose-bvar loose-s not-less-zero)

  have term-ok Θ (subst-bv (Fv x τ') t)
    using eq proved-terms-well-formed(2) wt-term-def typ-of-def
    by (meson term-ok-app-eqD)
  hence is-closed (subst-bv (Fv x τ') t)
    using wt-term-def typ-of-imp-closed by auto
  hence loose-s: ¬ loose-bvar t 1
    using is-closed-subst-bv by simp
  hence loose-s': (⋀x. 1 < x ⟹ ¬ loose-bvar1 t x)
    by (simp add: not-loose-bvar-imp-not-loose-bvar1-all-greater)
  moreover have  ¬ occs (case-prod Fv (x,τ')) t
  proof–
    have (x,τ') ∉ fv t
```

```
    using x(2) by auto
  thus ?thesis
    by (simp add: fv-iff-occs)
 qed
 ultimately have t: Abs-fv x τ' (subst-bv (term.Fv x τ') t) = Abs τ' t
  unfolding subst-bv-def bind-fv-def
    using bind-fv2-subst-bv1-cancel
  by (metis (full-types) case-prod-conv less-one linorder-neqE-nat loose-bvar1-imp-loose-bvar

      loose-s not-less-zero)

 from 1 s t show ?thesis
  using ctxt  eq x(1) x(3) by fastforce
qed

lemma proves-descend-abs-rule-iff':
 assumes thy: wf-theory Θ
 assumes x: (x, τ') ∉ FV Γ (x, τ') ∉ fv (mk-eq (Abs τ' s) (Abs τ' t)) typ-ok Θ τ'
 assumes ctxt: finite Γ ∀ A∈Γ. term-ok Θ A ∀ A∈Γ. typ-of A = Some propT
 shows Θ, Γ ⊢ mk-eq (subst-bv (Fv x τ') s) (subst-bv (Fv x τ') t)
  ⟷ Θ, Γ ⊢ mk-eq (Abs τ' s) (Abs τ' t)
 apply (rule iffI)
 using assms proves-ascend-abs-rule' apply simp
 using assms proves-descend-abs-rule' by simp

lemma proves-beta-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 beta: t →β 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)
using beta term-ok' free proof(induction t u arbitrary: vs rule: beta.induct)
 case (beta T s t)
 have ok: term-ok Θ (subst-bvs (map (case-prod Fv) vs) (Abs T s))
   term-ok Θ (subst-bvs (map (case-prod Fv) vs) t)
   using beta.prems(1) apply simp-all
   using term-ok-app-eqD term-ok-def by blast+

 have ∀ x ∈ set (map (case-prod Fv) vs) . is-closed x
   using beta.prems(2) by auto
 hence simp: subst-bvs (map (case-prod Fv) vs) (Abs T s)
   = Abs T (subst-bvs1' s 1 (map (case-prod Fv) vs))
   by auto
 hence ok': term-ok Θ (Abs T (subst-bvs1' s 1 (map (case-prod Fv) vs)))
   using ok by simp
```

have *T*: *typ-of* (*subst-bvs* (*map* (*case-prod Fv*) *vs*) *t*) = *Some T*
using *ok*(*2*) *wt-term-def typ-of-beta-redex-arg simp*
using *beta.prems*(*1*) *subst-bvs-App*
by (*metis term-okD2*)

have *ok-unf*: *wt-term* (*sig* Θ) (*Abs T* (*subst-bvs1′ s 1* (*map* (*case-prod Fv*) *vs*)))
wf-term (*sig* Θ) (*subst-bvs* (*map* (*case-prod Fv*) *vs*) *t*)
using *ok*(*2*) *ok′ wt-term-def* **by** *simp-all*

have *subst-bvs* (*map* (λ*a*. *case a of* (*a*, *b*) ⇒ *term.Fv a b*) *vs*)
(*Abs T s* $ *t*) =
Abs T (*subst-bvs1′ s 1* (*map* (*case-prod Fv*) *vs*)) $ *subst-bvs* (*map* (*case-prod Fv*) *vs*) *t*
by (*simp add*: *simp*)
moreover have *subst-bvs* (*map* (*case-prod Fv*) *vs*) (*subst-bv2 s 0 t*)
= (*subst-bv* (*subst-bvs* (*map* (*case-prod Fv*) *vs*) *t*)
(*subst-bvs1′ s 1* (*map* (*case-prod Fv*) *vs*)))
using *subst-bvs1′-subst-bv2*[*symmetric*] *subst-bvs-subst-bvs1′*
by *simp* (*metis One-nat-def Suc-eq-plus1 map-map simp subst-bvs1.simps*(*2*) *subst-bvs1-subst-bvs1′*
subst-bvs-def substn-subst-0′ term.inject(*4*))
ultimately show *?case*
using *β-conversion*[*OF thy ok-unf*, *of* Γ] *T* **by** *simp*
next
case (*appL s t u*)
hence *ok*: *term-ok* Θ (*subst-bvs* (*map* (*case-prod Fv*) *vs*) *s*)
term-ok Θ (*subst-bvs* (*map* (*case-prod Fv*) *vs*) *u*)
by (*metis subst-bvs-App term-ok-app-eqD*)+
moreover have ∀ *a* ∈ *set vs*. *case a of* (*x*, *τ*) ⇒ (*x*, *τ*) ∉ *fv s* ∪ *FV* Γ
using *appL* **by** *simp*
ultimately have Θ,Γ ⊢ *mk-eq* (*subst-bvs* (*map* (*case-prod Fv*) *vs*) *s*)
(*subst-bvs* (*map* (*case-prod Fv*) *vs*) *t*)
using *appL.IH* **by** *blast*
moreover have Θ,Γ ⊢ *mk-eq* (*subst-bvs* (*map* (*case-prod Fv*) *vs*) *u*)
(*subst-bvs* (*map* (*case-prod Fv*) *vs*) *u*)
using *proves-eq-reflexive*[*OF thy ok*(*2*), *of* Γ, *OF finite ctxt*] **by** *blast*
moreover obtain *τ* **where** *τ*: *typ-of*
(*subst-bvs* (*map* (*case-prod Fv*) *vs*) *u*) = *Some τ*
using *ok wt-term-def* **by** *auto*
moreover obtain *τ′* **where** *typ-of*
(*subst-bvs* (*map* (*case-prod Fv*) *vs*) *s*) = *Some* (*τ* → *τ′*)
using *τ appL.prems*(*1*) *not-None-eq subst-bvs-App wt-term-def typ-of1-arg-typ typ-of-def*
by (*metis term-okD2*)
ultimately show *?case*
using *proves-eq-combination-rule-better thy finite ctxt* **by** *simp*
next
case (*appR s t u*)
hence *ok*: *term-ok* Θ (*subst-bvs* (*map* (*case-prod Fv*) *vs*) *s*)

term-ok Θ (*subst-bvs* (*map* (*case-prod Fv*) *vs*) *u*)
by (*metis subst-bvs-App term-ok-app-eqD*)+
moreover have ∀ *a* ∈ *set vs. case a of* (*x*, τ) ⇒ (*x*, τ) ∉ *fv s* ∪ *FV* Γ
using *appR* **by** *simp*
ultimately have Θ,Γ ⊢ *mk-eq* (*subst-bvs* (*map* (*case-prod Fv*) *vs*) *s*)
(*subst-bvs* (*map* (*case-prod Fv*) *vs*) *t*)
using *appR.IH* **by** *blast*
moreover have Θ,Γ ⊢ *mk-eq* (*subst-bvs* (*map* (*case-prod Fv*) *vs*) *u*)
(*subst-bvs* (*map* (*case-prod Fv*) *vs*) *u*)
using *proves-eq-reflexive*[*OF thy ok*(*2*), *of* Γ, *OF finite ctxt*] **by** *blast*
moreover obtain τ **where** τ: *typ-of*
(*subst-bvs* (*map* (*case-prod Fv*) *vs*) *s*) = *Some* τ
using *ok wt-term-def* **by** *auto*
moreover obtain τ′ **where** *typ-of*
(*subst-bvs* (*map* (*case-prod Fv*) *vs*) *u*) = *Some* (τ → τ′)
using τ *appR.prems*(*1*) *not-None-eq subst-bvs-App wt-term-def typ-of1-arg-typ typ-of-def*
by (*metis term-okD2*)
ultimately show *?case*
using *proves-eq-combination-rule-better thy finite ctxt* **by** *simp*
next
case (*abs s t T*)
have ∀ *a* ∈ *set vs. case a of* (*x*, τ) ⇒ (*x*, τ) ∉ *fv s* ∪ *FV* Γ
using *abs.prems*(*2*) **by** *auto*

have ∀ *v*∈*set* (*map* (*case-prod Fv*) *vs*) . *is-closed v*
by *auto*

hence *simp*: *mk-eq* (*subst-bvs* (*map* (*case-prod Fv*) *vs*) (*Abs T s*))
(*subst-bvs* (*map* (*case-prod Fv*) *vs*) (*Abs T t*))
= *mk-eq* (*Abs T* (*subst-bvs1′ s 1* (*map* (*case-prod Fv*) *vs*)))
(*Abs T* (*subst-bvs1′ t 1* (*map* (*case-prod Fv*) *vs*)))
by *simp*

have *T-ok*: *typ-ok* Θ *T*
using *abs.prems term-ok-Types-typ-ok simp thy* **by** *auto*

have *1*: *finite* (*fv* (*mk-eq* (*Abs T* (*subst-bvs1′ s 1* (*map* (*case-prod Fv*) *vs*)))
(*Abs T* (*subst-bvs1′ t 1* (*map* (*case-prod Fv*) *vs*)))) ∪ *FV* Γ ∪ *fv s*)
using *finite finite-fv finite-FV* **by** *simp*
hence ∃ *x* . (*x*,*T*) ∉ (*fv* (*mk-eq* (*Abs T* (*subst-bvs1′ s 1* (*map* (*case-prod Fv*) *vs*)))
(*Abs T* (*subst-bvs1′ t 1* (*map* (*case-prod Fv*) *vs*)))) ∪ *FV* Γ ∪ *fv s*)
proof −
have ⋀*v t P*. (*v*, *t*) ∉ *P* ∨ *v* ∈ *fst* ‘ *P*
by (*metis* (*no-types*) *fst-conv image-eqI*)
then show *?thesis*
using *1 variant-variable-fresh finite-Un finite-imageI fst-conv image-eqI* **by** *smt*
qed

from *this*
obtain x **where** x: $(x,T) \notin (fv\ (mk\text{-}eq\ (Abs\ T\ (subst\text{-}bvs1'\ s\ 1\ (map\ (case\text{-}prod\ Fv)\ vs)))$
$(Abs\ T\ (subst\text{-}bvs1'\ t\ 1\ (map\ (case\text{-}prod\ Fv)\ vs)))) \cup FV\ \Gamma \cup fv\ s)$
by *fastforce*
hence x: $(x,\ T) \notin fv\ (mk\text{-}eq\ (Abs\ T\ (subst\text{-}bvs1'\ s\ 1\ (map\ (case\text{-}prod\ Fv)\ vs)))$
$(Abs\ T\ (subst\text{-}bvs1'\ t\ 1\ (map\ (case\text{-}prod\ Fv)\ vs))))$
$(x,T) \notin FV\ \Gamma\ (x,\ T) \notin fv\ s$
by *auto*

have *ok*: *term-ok* Θ $(Abs\ T\ (subst\text{-}bvs1'\ s\ 1\ (map\ (case\text{-}prod\ Fv)\ vs)))$
using *abs.prems*(*1*) *simp* **by** *auto*

thm *subst-bvs-extend-lower-level*
have *combine*: $(subst\text{-}bv\ (term.Fv\ x\ T)$
$(subst\text{-}bvs1'\ s\ 1\ (map\ (\lambda(x,\ y).\ term.Fv\ x\ y)\ vs))) =$
$(subst\text{-}bvs\ (map\ (case\text{-}prod\ Fv)\ ((x,T)\#vs))\ s)$
using *subst-bvs-extend-lower-level*
using ‹$\forall\ v{\in}set\ (map\ (\lambda(x,\ y).\ term.Fv\ x\ y)\ vs).\ is\text{-}closed\ v$› **by** *auto*
have *1*: $\Theta,\Gamma \vdash mk\text{-}eq\ (subst\text{-}bvs\ (map\ (case\text{-}prod\ Fv)\ ((x,T)\#vs))\ s)$
$(subst\text{-}bvs\ (map\ (case\text{-}prod\ Fv)\ ((x,T)\#vs))\ t)$
apply(*rule abs.IH*)
using *ok* **apply** (*metis combine term-ok-subst-bv*)
using *x abs.prems*(*2*) **by** *auto*
have $\Theta,\ \Gamma \vdash mk\text{-}eq$
$(Abs\ T\ (subst\text{-}bvs1'\ s\ 1\ (map\ (case\text{-}prod\ Fv)\ vs)))$
$(Abs\ T\ (subst\text{-}bvs1'\ t\ 1\ (map\ (case\text{-}prod\ Fv)\ vs)))$
apply (*rule proves-ascend-abs-rule'*[**where** $x{=}x$])
using *thy* **apply** *simp*
using *x* **apply** *simp*
using *x* **apply** *simp*
using *T-ok* **apply** *simp*
using *1* ‹$\forall\ v{\in}set\ (map\ (\lambda(x,\ y).\ term.Fv\ x\ y)\ vs).\ is\text{-}closed\ v$› *subst-bvs-extend-lower-level*

finite ctxt **by** *auto*
then show *?case*
using *simp* **by** *auto*
qed

lemma *subst-bvs-empty*[*simp*]: *subst-bvs* [] $t = t$
by (*simp add*: *subst-bvs-subst-bvs1'*)

lemma *proves-beta-step*:
assumes *thy*: *wf-theory* Θ
assumes *finite*: *finite* Γ
assumes *term-ok*: *term-ok* Θ 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 Θ, $\Gamma \vdash$ *mk-eq t u*
proof−
have *unsimpt*: *t* = *subst-bvs* (*map* (*case-prod Fv*) []) *t*
by *simp*
moreover have *unsimpu*: *u* = *subst-bvs* (*map* (*case-prod Fv*) []) *u*
by *simp*
ultimately have *unsimp*: *mk-eq t u* = *mk-eq*
(*subst-bvs* (*map* (*case-prod Fv*) []) *t*)
(*subst-bvs* (*map* (*case-prod Fv*) []) *u*)
by *simp*
show *?thesis*
apply (*subst unsimp*)
apply (*rule proves-beta-step-pre*)
using *assms* **by** *simp-all*
qed

lemma *proves-beta-steps*:
assumes *thy*: *wf-theory* Θ
assumes *finite*: *finite* Γ
assumes *term-ok*: *term-ok* Θ *t*
assumes *beta*: $t \rightarrow_\beta^* u$
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*
using *beta term-ok* **proof** (*induction rule*: *rtranclp.induct*)
case (*rtrancl-refl a*)
then show *?case* **using** *finite ctxt* **by** (*simp add*: *proves-eq-reflexive thy*)
next
case (*rtrancl-into-rtrancl a b c*)
hence Θ,$\Gamma \vdash$ *mk-eq a b* **by** *simp*
moreover have Θ,$\Gamma \vdash$ *mk-eq b c*
using *proves-beta-step rtrancl-into-rtrancl.hyps*(*2*)
using *beta-star-preserves-term-ok local.finite rtrancl-into-rtrancl.hyps*(*1*)
rtrancl-into-rtrancl.prems thy finite ctxt **by** *blast*
ultimately show *?case*
by (*meson finite ctxt proved-terms-well-formed*(*2*) *proves-eq-transitive-rule*[*OF thy - - - - - - - finite ctxt*]
term-ok-mk-eqD term-ok-mk-eq-same-typ thy)
qed

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*: $\forall A{\in}\Gamma$. *term-ok* Θ *A* $\forall A{\in}\Gamma$. *typ-of A* = *Some propT*
shows Θ, $\Gamma \vdash$ *mk-eq t u*
using *finite ctxt*
by (*simp add*: *beta-norm-imp-beta-reds local.beta local.finite proves-beta-steps term-ok thy*

del: *term-ok-def*)

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*
using *assms proves-eq-mp-rule-better*[*OF thy - - finite ctxt*] *proves-beta-norm*[*OF thy finite - - ctxt*]
proved-terms-well-formed(*2*)
by *blast*

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* →η *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*)
using *eta term-ok' free* **proof**(*induction t u arbitrary*: *vs rule*: *eta.induct*)
case (*eta s T*)

have *closeds*: ∀ *x* ∈ *set* (*map* (*case-prod Fv*) *vs*) . *is-closed x*
using *eta.prems*(*2*) **by** *auto*
hence *simp*: *subst-bvs* (*map* (*case-prod Fv*) *vs*) (*Abs T* (*s* $ *Bv 0*))
= *Abs T* (*subst-bvs1'* (*s* $ *Bv 0*) *1* (*map* (*case-prod Fv*) *vs*))
by *auto*
hence *simp'*: *subst-bvs* (*map* (*case-prod Fv*) *vs*) (*Abs T* (*s* $ *Bv 0*))
= *Abs T* (*subst-bvs1' s 1* (*map* (*case-prod Fv*) *vs*) $ *Bv 0*)
by *auto*

have *closed*: *is-closed* (*subst-bvs* (*map* (*case-prod Fv*) *vs*) (*Abs T* (*s* $ *Bv 0*)))
using *eta*(*2*) *wt-term-def typ-of-imp-closed* **by** *auto*
hence *no-loose1*: ¬ *loose-bvar* (*subst-bvs1' s 1* (*map* (*case-prod Fv*) *vs*)) *1*
unfolding *is-open-def*
by (*metis One-nat-def Suc-eq-plus1 loose-bvar.simps*(*2*) *loose-bvar.simps*(*3*) *simp subst-bvs1'.simps*(*3*))
have *not-dependent*: ¬ *is-dependent* (*subst-bvs1' s 1* (*map* (*case-prod Fv*) *vs*))
using *is-closed-subst-bvs1'-closeds*
by (*simp add*: *closeds eta.hyps*)

have *decr-simp*: *subst-bv x* (*subst-bvs1' s 1* (*map* (*case-prod Fv*) *vs*))
= *subst-bvs* (*map* (*case-prod Fv*) *vs*) (*decr 0 s*) **for** *x*
apply (*simp add*: *closeds eta.hyps subst-bvs-decr*)

using *is-dependent-def no-loose-bvar1-subst-bv2-decr not-dependent substn-subst-0′*
by *auto*
have *ok*: *term-ok* Θ (*subst-bvs1′ s 1* (*map* (*case-prod Fv*) *vs*))
by (*metis One-nat-def Suc-leI eta.prems*(*1*) *is-dependent-def le-eq-less-or-eq*
loose-bvar-decr-unchanged loose-bvar-iff-exist-loose-bvar1 no-loose1 not-dependent
simp′
term-ok-eta-red-step)
hence *ok-ind*: *wf-term* (*sig* Θ) (*subst-bvs1′ s 1* (*map* (*case-prod Fv*) *vs*))
using *wt-term-def* **by** *simp*

obtain τ **where** *typ-of* (*Abs T* (*subst-bvs1′* (*s* $ *Bv 0*) *1* (*map* (*case-prod Fv*)
vs))) = *Some* (*T* → τ)
using *eta.prems*(*1*) *simp wt-term-def typ-of-Abs-body-typ′*
by (*smt has-typ-iff-typ-of typ-of-def term-ok-def*)
hence *ty*: *typ-of* (*subst-bvs1′ s 1* (*map* (*case-prod Fv*) *vs*)) = *Some* (*T* → τ)
using *eta.eta eta-preserves-typ-of is-closed-decr-unchanged not-dependent*
ok simp simp′ wt-term-def typ-of-imp-closed
by (*metis* (*no-types*, *lifting*) *has-typ-imp-closed term-ok-def*)

then show *?case*
using *proves.eta*[*OF thy ok-ind*, *of* - - Γ] *ty decr-simp simp′*
by (*simp add*: *closeds eta.hyps subst-bvs-decr typ-of-imp-closed*)
next
case (*appL s t u*)
hence *ok*: *term-ok* Θ (*subst-bvs* (*map* (*case-prod Fv*) *vs*) *s*)
term-ok Θ (*subst-bvs* (*map* (*case-prod Fv*) *vs*) *u*)
by (*metis subst-bvs-App term-ok-app-eqD*)+
moreover have ∀ *a* ∈ *set vs. case a of* (*x*, τ) ⇒ (*x*, τ) ∉ *fv s* ∪ *FV* Γ
using *appL* **by** *simp*
ultimately have Θ,Γ ⊢ *mk-eq* (*subst-bvs* (*map* (*case-prod Fv*) *vs*) *s*)
(*subst-bvs* (*map* (*case-prod Fv*) *vs*) *t*)
using *appL.IH* **by** *blast*
moreover have Θ,Γ ⊢ *mk-eq* (*subst-bvs* (*map* (*case-prod Fv*) *vs*) *u*)
(*subst-bvs* (*map* (*case-prod Fv*) *vs*) *u*)
using *proves-eq-reflexive*[*OF thy ok*(*2*), *of* Γ, *OF finite ctxt*] **by** *blast*
moreover obtain τ **where** τ: *typ-of*
(*subst-bvs* (*map* (*case-prod Fv*) *vs*) *u*) = *Some* τ
using *ok wt-term-def* **by** *auto*
moreover obtain τ′ **where** *typ-of*
(*subst-bvs* (*map* (*case-prod Fv*) *vs*) *s*) = *Some* (τ → τ′)
using τ *appL.prems*(*1*) *not-None-eq subst-bvs-App wt-term-def typ-of1-arg-typ*
typ-of-def
by (*smt has-typ-iff-typ-of typ-of-def term-ok-def*)
ultimately show *?case*
using *proves-eq-combination-rule-better thy finite ctxt* **by** *simp*
next
case (*appR s t u*)
hence *ok*: *term-ok* Θ (*subst-bvs* (*map* (*case-prod Fv*) *vs*) *s*)
term-ok Θ (*subst-bvs* (*map* (*case-prod Fv*) *vs*) *u*)

by (*metis subst-bvs-App term-ok-app-eqD*)+
moreover have $\forall\, a \in$ *set vs. case a of* $(x, \tau) \Rightarrow (x, \tau) \notin$ *fv s* $\cup$ *FV* Γ
using *appR* **by** *simp*
ultimately have $\Theta,\Gamma \vdash$ *mk-eq* (*subst-bvs* (*map* (*case-prod Fv*) *vs*) *s*)
(*subst-bvs* (*map* (*case-prod Fv*) *vs*) *t*)
using *appR.IH* **by** *blast*
moreover have $\Theta,\Gamma \vdash$ *mk-eq* (*subst-bvs* (*map* (*case-prod Fv*) *vs*) *u*)
(*subst-bvs* (*map* (*case-prod Fv*) *vs*) *u*)
using *proves-eq-reflexive*[*OF thy ok*(*2*), *of* Γ, *OF finite ctxt*] **by** *blast*
moreover obtain τ **where** τ: *typ-of*
(*subst-bvs* (*map* (*case-prod Fv*) *vs*) *s*) = *Some* τ
using *ok wt-term-def* **by** *auto*
moreover obtain τ' **where** *typ-of*
(*subst-bvs* (*map* (*case-prod Fv*) *vs*) *u*) = *Some* $(\tau \to \tau')$
using τ *appR.prems*(*1*) *not-None-eq subst-bvs-App wt-term-def typ-of1-arg-typ typ-of-def*
by (*metis term-okD2*)
ultimately show *?case*
using *proves-eq-combination-rule-better thy finite ctxt* **by** *simp*
next
case (*abs s t T*)
have $\forall\, a \in$ *set vs. case a of* $(x, \tau) \Rightarrow (x, \tau) \notin$ *fv s* $\cup$ *FV* Γ
using *abs.prems*(*2*) **by** *auto*

have $\forall\, v{\in}$*set* (*map* (*case-prod Fv*) *vs*) . *is-closed v*
by *auto*

hence *simp*: *mk-eq* (*subst-bvs* (*map* (*case-prod Fv*) *vs*) (*Abs T s*))
(*subst-bvs* (*map* (*case-prod Fv*) *vs*) (*Abs T t*))
= *mk-eq* (*Abs T* (*subst-bvs1'* *s 1* (*map* (*case-prod Fv*) *vs*)))
(*Abs T* (*subst-bvs1'* *t 1* (*map* (*case-prod Fv*) *vs*)))
by *simp*

have *T-ok*: *typ-ok* Θ *T*
using *abs.prems term-ok-Types-typ-ok simp thy* **by** *auto*

have *1*: *finite* (*fv* (*mk-eq* (*Abs T* (*subst-bvs1'* *s 1* (*map* (*case-prod Fv*) *vs*)))
(*Abs T* (*subst-bvs1'* *t 1* (*map* (*case-prod Fv*) *vs*)))) $\cup$ *FV* $\Gamma \cup$ *fv s*)
using *finite finite-fv finite-FV* **by** *simp*
hence $\exists\, x$. $(x,T) \notin$ (*fv* (*mk-eq* (*Abs T* (*subst-bvs1'* *s 1* (*map* (*case-prod Fv*) *vs*)))
(*Abs T* (*subst-bvs1'* *t 1* (*map* (*case-prod Fv*) *vs*)))) $\cup$ *FV* $\Gamma \cup$ *fv s*)
proof −
have $\bigwedge v\ t\ P$. (*v*::*variable*, *t*::*typ*) $\notin P \vee v \in$ *fst* ‘ *P*
by (*metis* (*no-types*) *fst-conv image-eqI*)
then show *?thesis*
using *1 variant-variable-fresh finite-Un finite-imageI fst-conv image-eqI*
by *smt*
qed
from *this*

obtain x **where** x: $(x,T) \notin (fv\ (mk\text{-}eq\ (Abs\ T\ (subst\text{-}bvs1'\ s\ 1\ (map\ (case\text{-}prod\ Fv)\ vs)))$
$(Abs\ T\ (subst\text{-}bvs1'\ t\ 1\ (map\ (case\text{-}prod\ Fv)\ vs)))) \cup FV\ \Gamma \cup fv\ s)$
by *fastforce*
hence x: $(x,\ T) \notin fv\ (mk\text{-}eq\ (Abs\ T\ (subst\text{-}bvs1'\ s\ 1\ (map\ (case\text{-}prod\ Fv)\ vs)))$
$(Abs\ T\ (subst\text{-}bvs1'\ t\ 1\ (map\ (case\text{-}prod\ Fv)\ vs))))$
$(x,T) \notin FV\ \Gamma\ (x,\ T) \notin fv\ s$
by *auto*

have *ok*: $term\text{-}ok\ \Theta\ (Abs\ T\ (subst\text{-}bvs1'\ s\ 1\ (map\ (case\text{-}prod\ Fv)\ vs)))$
using *abs.prems*(*1*) *simp* **by** *auto*

have *combine*: $(subst\text{-}bv\ (Fv\ x\ T)$
$(subst\text{-}bvs1'\ s\ 1\ (map\ (case\text{-}prod\ Fv)\ vs))) =$
$(subst\text{-}bvs\ (map\ (case\text{-}prod\ Fv)\ ((x,T)\#vs))\ s)$
using *subst-bvs-extend-lower-level*
using ‹$\forall\ v{\in}set\ (map\ (\lambda(x,\ y).\ term.Fv\ x\ y)\ vs).\ is\text{-}closed\ v$› **by** *auto*
have *1*: $\Theta,\Gamma \vdash mk\text{-}eq\ (subst\text{-}bvs\ (map\ (case\text{-}prod\ Fv)\ ((x,T)\#vs))\ s)$
$(subst\text{-}bvs\ (map\ (case\text{-}prod\ Fv)\ ((x,T)\#vs))\ t)$
apply(*rule abs.IH*)
using *ok combine* **apply** (*metis term-ok-subst-bv*)
using *x abs.prems*(*2*) **by** *auto*
have $\Theta,\ \Gamma \vdash mk\text{-}eq$
$(Abs\ T\ (subst\text{-}bvs1'\ s\ 1\ (map\ (case\text{-}prod\ Fv)\ vs)))$
$(Abs\ T\ (subst\text{-}bvs1'\ t\ 1\ (map\ (case\text{-}prod\ Fv)\ vs)))$
apply (*rule proves-ascend-abs-rule'*[**where** $x{=}x$])
using *thy* **apply** *simp*
using *x* **apply** *simp*
using *x* **apply** *simp*
using *T-ok* **apply** *simp*
using *1* ‹$\forall\ v{\in}set\ (map\ (\lambda(x,\ y).\ term.Fv\ x\ y)\ vs).\ is\text{-}closed\ v$› *subst-bvs-extend-lower-level*
finite ctxt **by** *auto*
then show *?case*
using *simp* **by** *auto*
qed

lemma *proves-eta-step*:
assumes *thy*: *wf-theory* Θ
assumes *finite*: *finite* Γ
assumes *term-ok*: $term\text{-}ok\ \Theta\ t$
assumes *eta*: $t \rightarrow_\eta u$
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−
have *unsimpt*: $t = subst\text{-}bvs\ (map\ (case\text{-}prod\ Fv)\ []))\ t$
by *simp*
moreover have *unsimpu*: $u = subst\text{-}bvs\ (map\ (case\text{-}prod\ Fv)\ []))\ u$
by *simp*
ultimately have *unsimp*: $mk\text{-}eq\ t\ u = mk\text{-}eq$

(*subst-bvs* (*map* (*case-prod Fv*) []) *t*)
(*subst-bvs* (*map* (*case-prod Fv*) []) *u*)
by *simp*
show *?thesis*
apply (*subst unsimp*)
apply (*rule proves-eta-step-pre*)
using *assms* **by** *simp-all*
qed

lemma *proves-eta-steps*:
assumes *thy*: *wf-theory* Θ
assumes *finite*: *finite* Γ
assumes *term-ok*: *term-ok* Θ *t*
assumes *eta*: $t \rightarrow_\eta^* u$
assumes *ctxt*: ∀ *A*∈Γ. *term-ok* Θ *A* ∀ *A*∈Γ. *typ-of A* = *Some propT*
shows Θ, Γ ⊢ *mk-eq t u*
using *eta term-ok* **proof** (*induction rule*: *rtranclp.induct*)
case (*rtrancl-refl a*)
then show *?case* **using** *finite ctxt* **by** (*simp add*: *proves-eq-reflexive thy*)
next
case (*rtrancl-into-rtrancl a b c*)
hence Θ,Γ ⊢ *mk-eq a b* **by** *simp*
moreover have Θ,Γ ⊢ *mk-eq b c*
using *proves-eta-step rtrancl-into-rtrancl.hyps*(*2*) *eta-star-preserves-term-ok local.finite*
rtrancl-into-rtrancl.hyps(*1*) *rtrancl-into-rtrancl.prems thy finite ctxt*
by *blast*
ultimately show *?case*
by (*meson proved-terms-well-formed*(*2*) *proves-eq-transitive-rule*[*OF thy - - - - - - - - finite ctxt*]
term-ok-mk-eqD term-ok-mk-eq-same-typ thy)
qed

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*: ∀ *A*∈Γ. *term-ok* Θ *A* ∀ *A*∈Γ. *typ-of A* = *Some propT*
shows Θ, Γ ⊢ *mk-eq t u*
using *finite ctxt*
by (*simp add*: *eta-norm-imp-eta-reds local.eta local.finite proves-eta-steps term-ok thy del*: *term-ok-def*)

lemma *eta-norm-preserves-proves*:
assumes *thy*: *wf-theory* Θ
assumes *finite*: *finite* Γ
assumes *term-ok*: Θ, Γ ⊢ *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 $\Theta, \Gamma \vdash u$
using *assms proves-eq-mp-rule-better*[*OF thy - - finite ctxt*]
proves-eta-norm[*OF thy finite - - ctxt*] *proved-terms-well-formed*(*2*) **by** *blast*

lemma *beta-eta-norm-preserves-proves*:
assumes *thy*: *wf-theory* Θ
assumes *finite*: *finite* Γ
assumes *term-ok*: $\Theta, \Gamma \vdash t$
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 $\Theta, \Gamma \vdash u$
using *beta-eta beta-norm-preserves-proves*[*OF thy finite - - ctxt*]
eta-norm-preserves-proves[*OF thy finite - - ctxt*] *finite term-ok thy* **by** *blast*

lemma *forall-elim'*:
assumes *thy*: *wf-theory* Θ
assumes *all*: $\Theta, \Gamma \vdash$ *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 $\Theta, \Gamma \vdash B \cdot a$
proof(*cases is-Abs B*)
case *True*
from *this* **obtain** t T **where** *Abs*: B = *Abs* T t
using *is-Abs-def* **by** *auto*
have $T = \tau$
by (*smt Abs all list.inject proved-terms-well-formed*(*1*) *typ.inject*(*1*) *typ-of1.simps*(*1*)

typ-of-Abs-body-typ' typ-of-def typ-of-fun)
then show *?thesis*
using *True Abs all a* **by** (*auto intro*: *forall-elim*[**where** τ=τ])
next
case *False*

have *wf-B*: *wf-term* (*sig* Θ) B
using *all proved-terms-well-formed*(*2*) *term-okD1 term-ok-app-eqD* **by** *blast*
have *B-typ*: $\vdash_\tau B : \tau \to propT$
by (*metis* (*no-types*, *lifting*) *all proved-terms-well-formed*(*1*) *typ-of1.simps*(*1*)
typ-of-def

typ-of-fun typ-of-imp-has-typ)

have $B \cdot a = B$ \$ a
using *False* **by** (*metis betapply.elims term.discI*(*4*))
moreover have *Abs* τ (B \$ *Bv 0*) $\cdot$ $a = B$ \$ a
using *B-typ closed-subst-bv-no-change subst-bv-def typ-of-imp-closed*
by (*auto simp add*: *subst-bv-def incr-boundvars-def*)
ultimately have *simp*: $B \cdot a$ = *subst-bv* a (B \$ *Bv 0*)
by *auto*

have *1*: Θ, Γ ⊢ *mk-eq* (*Abs* τ (*B* $ *Bv 0*)) *B*
by (*rule proves.eta*[*OF thy wf-B B-typ*])
have *2*: Θ, Γ ⊢ *mk-eq B* (*Abs* τ (*B* $ *Bv 0*))
apply (*rule proves-eq-symmetric-rule*[*OF thy - - - 1 ctxt*])
using *wf-B B-typ term-ok-def wt-term-def* **apply** *blast*
using *1 proved-terms-well-formed*(*2*) *term-ok-mk-eqD* **apply** *blast*
using *B-typ Logic.typ-of-eta-expand* **by** *auto*
have *3*: Θ, Γ ⊢ *mk-eq* (*Ct STR ''Pure.all''* ((τ → *propT*) → *propT*)) (*Ct STR ''Pure.all''* ((τ → *propT*) → *propT*))
apply (*rule proves-eq-reflexive*[*OF thy - ctxt*])
using *all proved-terms-well-formed*(*2*) *term-ok-app-eqD* **by** *blast*

have *4*: Θ, Γ ⊢ *mk-eq*
(*Ct STR ''Pure.all''* ((τ → *propT*) → *propT*) $ *B*)
(*Ct STR ''Pure.all''* ((τ → *propT*) → *propT*) $ (*Abs* τ (*B* $ *Bv 0*)))
apply (*rule proves-eq-combination-rule-better*[*OF thy 3 2 - - ctxt*, **where** τ=(τ → *propT*) **and** τ′= *propT*])
using *typ-of-def* **apply** *auto*[*1*]
using *B-typ* **by** *blast*

have *5*: Θ, Γ ⊢ (*Ct STR ''Pure.all''* ((τ → *propT*) → *propT*) $ (*Abs* τ (*B* $ *Bv 0*)))
by (*rule proves-eq-mp-rule-better*[*OF thy 4 all ctxt*])

show *?thesis*
apply (*subst simp*)
apply (*rule proves.forall-elim*[*OF 5*])
using *assms*(*3*) **apply** *blast*
using *assms*(*4*) **by** *blast*
qed
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
  by (induction P) auto
lemma size P > 0
  by (induction P) auto
lemma size P ≥ depth P
  by (induction P) auto

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
  by (induction rule: partial-nth.induct) auto

lemma partial-nth-Some-imp-elem: partial-nth l n = Some x ⟹ x∈set l
  by (induction rule: partial-nth.induct) auto
```

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* {})
  by (*induction t lev u rule*: *subst-bv1.induct*) (*auto simp add*: *incr-boundvars-def*)

corollary *fv-subst-bvs-upper-bound*:
  assumes *is-closed t*
  shows *fv* (*subst-bvs us t*) ⊆ *fv t* ∪ (⋃*x*∈*set us* . (*fv x*))
  unfolding *subst-bvs-def*
  using *assms* **by** (*simp add*: *is-open-def no-loose-bvar-imp-no-subst-bvs1*)

lemma *fv-subst-bvs1-upper-bound*:
  fv (*subst-bvs1 t lev us*) ⊆ *fv t* ∪ (⋃*x*∈*set us* . (*fv x*))

proof (*induction t lev us rule*: *subst-bvs1.induct*)
  case (*1 n lev args*)
  then show *?case*
  proof (*induction args arbitrary*: *n lev*)
    case *Nil*
    then show *?case*
      by *simp*
  next
    case (*Cons a args*)
    then show *?case*
    by *simp* (*metis SUP-upper le-supI1 le-supI2 length-Suc-conv nth-mem set-ConsD set-eq-subset*)
  qed
qed (*auto simp add*: *incr-boundvars-def*)

lemma *typ-of-axiom*: *wf-theory thy* $\Longrightarrow$ *t* $\in$ *axioms thy* $\Longrightarrow$ *typ-of t* = *Some propT*

  by (*cases thy rule*: *theory-full-exhaust*) *simp*

fun *fv-Proof* :: *proofterm* $\Rightarrow$ (*variable* $\times$ *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* $\cup$ *fv-Proof p*
| *fv-Proof* (*Appt p t*) = *fv-Proof p* $\cup$ *fv t*
| *fv-Proof* (*AppP p1 p2*) = *fv-Proof p1* $\cup$ *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*
  by *simp*

lemma *typ-ok-contained-tvars-typ-ok*: *typ-ok thy ty* $\Longrightarrow$ (*idn*, *S*) $\in$ *tvsT ty* $\Longrightarrow$ *typ-ok thy* (*Tv idn S*)
  by (*induction ty*) (*use split-list typ-ok-Ty* **in** ‹*all* ‹*fastforce split*: *option.splits*››)

lemma *typ-ok-sig-contained-tvars-typ-ok-sig*:
  typ-ok-sig Σ *ty* $\Longrightarrow$ (*idn*, *S*) $\in$ *tvsT ty* $\Longrightarrow$ *typ-ok-sig* Σ (*Tv idn S*)
 by (*induction ty*) (*use split-list typ-ok-sig-Ty* **in** ‹*all* ‹*fastforce split*: *option.splits*››)

lemma *term-ok′-contained-tvars-typ-ok-sig*:
  term-ok′ Σ *t* $\Longrightarrow$ (*idn*, *S*) $\in$ *tvs t* $\Longrightarrow$ *typ-ok-sig* Σ (*Tv idn S*)

proof (*induction t*)
  case (*Ct n T*)
  hence *typ-ok-sig* Σ *T*
    by (*auto split*: *option.splits*)
  then show *?case*

```
    using typ-ok-sig-contained-tvars-typ-ok-sig Ct by auto
next
  case (Fv idn T)
  hence typ-ok-sig Σ T
    by (auto split: option.splits)
  then show ?case
    using typ-ok-sig-contained-tvars-typ-ok-sig Fv by auto
next
  case (Bv n)
  then show ?case by auto
next
  case (Abs T t)
  hence typ-ok-sig Σ T
    by (auto split: option.splits)
  then show ?case
    using typ-ok-sig-contained-tvars-typ-ok-sig Abs by fastforce
next
  case (App t1 t2)
  then show ?case
    by auto
qed

lemma term-ok-contained-tvars-typ-ok:
  term-ok thy t ⟹ (idn, S) ∈ tvs t ⟹ typ-ok thy (Tv idn S)
  using wt-term-def typ-ok-def term-ok'-contained-tvars-typ-ok-sig term-ok-def by
blast

lemma typ-ok-subst-typ:
  typ-ok thy T ⟹ ∀(-, ty) ∈ set insts . typ-ok thy ty ⟹ typ-ok thy (subst-typ
insts T)
proof (induction insts T rule: subst-typ.induct)
  case (1 insts n Ts)
  have typ-ok thy x if x∈set Ts for x
   by (metis (full-types) 1.prems(1) in-set-conv-decomp-first list-all-append list-all-simps(1)
      that typ-ok-Ty)
  hence typ-ok thy (subst-typ insts x) if x∈set Ts for x
    using that 1 by simp
  then show ?case
    using 1.prems(1) by (auto simp add: list-all-iff split: option.splits)
next
  case (2 insts idn S)
  then show ?case
  proof(cases (idn, S) ∈ set (map fst insts))
    case True
    obtain ty where ty: lookup (λk. k=(idn,S)) insts = Some ty
      by (metis (full-types) True lookup-None-iff not-Some-eq)
    hence subst-typ insts (Tv idn S) = ty
      by simp
    then show ?thesis
```

```
    using 2.prems(2) ty case-prodD lookup-present-eq-key' by fastforce
  next
    case False
    hence subst-typ insts (Tv idn S) = Tv idn S
    by (metis (mono-tags, lifting) lookup-None-iff subst-typ.simps(2) the-default.simps(1))
    then show ?thesis
      using 2.prems(1) by simp
  qed
qed

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 (induction insts T rule: subst-typ.induct)
  case (1 insts n Ts)
  have typ-ok-sig Σ x if x∈set Ts for x
    using 1.prems(1) split-list that typ-ok-sig-Ty by fastforce
  hence typ-ok-sig Σ (subst-typ insts x) if x∈set Ts for x
    using that 1 by simp
  then show ?case
    using 1.prems(1) by (auto simp add: list-all-iff split: option.splits)
next
  case (2 insts idn S)
  then show ?case
  proof(cases (idn, S) ∈ set (map fst insts))
    case True
    obtain ty where ty: lookup (λk. k=(idn,S)) insts = Some ty
      by (metis (full-types) True lookup-None-iff not-Some-eq)
    hence subst-typ insts (Tv idn S) = ty
      by simp
    then show ?thesis
      using 2.prems(2) ty case-prodD lookup-present-eq-key' by fastforce
  next
    case False
    hence subst-typ insts (Tv idn S) = Tv idn S
    by (metis (mono-tags, lifting) lookup-None-iff subst-typ.simps(2) the-default.simps(1))
    then show ?thesis
      using 2.prems(1) by simp
  qed
qed


lemma typ-ok-sig-imp-sortsT-ok-sig: typ-ok-sig Σ T ⟹ S ∈ SortsT T ⟹ wf-sort
(subclass (osig Σ)) S
  by (induction T) (use split-list in ‹all ‹fastforce simp add: wf-sort-def split:
option.splits››)

lemma term-ok'-imp-Sorts-ok-sig: term-ok' Σ t ⟹ S ∈ Sorts t ⟹ wf-sort
(subclass (osig Σ)) S
```

by (*induction t*) (*use typ-ok-sig-imp-sortsT-ok-sig* **in** ‹(*fastforce split*: *option.splits*)+›)

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*
using *assms* **proof**(*induction thy vs ns Hs P arbitrary*: *res rule*: *replay′.induct*)
case (*1 thy uu uv Hs t Tis*)
hence
ax: *t*∈*axioms thy*
and *insts*: *inst-ok thy Tis* **and** *t*: *term-ok thy t*
and *res*: *forall-intro-vars* (*subst-typ′ Tis t*) [] = *res*
by (*auto split*: *if-splits*)
hence *1*: *thy*, {} ⊢ *res*
using *res 1.prems*(*1*) *proved-terms-well-formed-pre*
using *axiom forall-intro-vars inst-ok-imp-wf-inst tsubst-simulates-subst-typ′*
by (*metis* (*no-types*, *lifting*) *empty-set*)
show *?case*
using *weaken-proves-set*[*of set Hs*, *OF - 1*]
using *1.prems*(*2*) *1.prems*(*3*) **by** *auto*
next
case (*2 thy ux uy Hs n*)
hence *res* ∈ *set Hs* **using** *partial-nth-Some-imp-elem* **by** *simp*
then show *?case* **using** *proves.assume 2* **by** (*simp add*: *wt-term-def*)
next
case (*3 thy vs ns Hs T p*)

obtain *s′ ns′* **where** *names*: (*s′*,*ns′*) = *variant-variable* (*Free STR ″default″*) *ns*
by *simp*
from *this 3* **obtain** *bres* **where** *bres*: *replay′ thy* ((*s′*, *T*) # *vs*) *ns′ Hs p* = *Some bres*
by (*auto split*: *if-splits prod.splits*)
have *ns′* = *insert s′ ns* **using** *variant-variable-adds names*
by (*metis fst-conv snd-conv*)
have *s′* ∉ *ns* **using** *3.prems variant-variable-fresh names*
by (*metis fst-conv*)

hence $s' \notin fst$ ` FV $(set\ Hs)$ **using** *3.prems* **by** *blast*
hence *free*: $(s', T) \notin FV$ $(set\ Hs)$ **by** *force*

have *typ-ok*: *wf-type* $(sig\ thy)$ T
using *names 3.prems* **by** (*auto split*: *if-splits*)
have *I*:*thy*, *set Hs* $\vdash$ *bres*
apply (*rule 3.IH*[*OF* - *names*])
using *names 3.prems* **apply** (*solves* ‹*simp split*: *if-splits*›)+
using *names 3.prems* ‹$ns' = insert\ s'\ ns$› **apply** *fastforce*
using *3.prems*(*7*) ‹$ns' = insert\ s'\ ns$› **apply** *auto*[*1*]
using *3.prems*(*8*) ‹$ns' = insert\ s'\ ns$› **apply** *auto*[*1*]
using *3.prems*(*6*) **apply** *fastforce*
using *3.prems*(*7*) ‹$ns' = insert\ s'\ ns$› **apply** *auto*[*1*]
using *3.prems*(*8*) ‹$ns' = insert\ s'\ ns$› **apply** *auto*[*1*]
using *bres* **by** *fastforce*
have *res*: $res = mk\text{-}all\ s'\ T\ bres$ **using** *names bres 3* **by** (*auto split*: *if-splits prod.splits*)
show *?case* **using** *proves.forall-intro*[*OF* ‹*wf-theory thy*› *I free typ-ok*] *res* **by** *simp*
next
case (*4 thy vs ns Hs p t*)
from ‹$replay'\ thy\ vs\ ns\ Hs\ (Appt\ p\ t) = Some\ res$› **obtain** *rep t' b s fun1 fun2 propT1 propT2* $\tau\ \tau'$ **where**
conds: $replay'\ thy\ vs\ ns\ Hs\ p = Some\ rep$
$t' = subst\text{-}bvs\ (map\ (\lambda(x,y)\ .\ Fv\ x\ y)\ vs)\ t$
$typ\text{-}of\ t' = Some\ \tau'$
$\tau = \tau'$
term-ok thy t'
$s = STR\ ''Pure.all'' \wedge fun1 = STR\ ''fun'' \wedge fun2 = STR\ ''fun'' \wedge propT1 = STR\ ''prop'' \wedge propT2 = STR\ ''prop''$
$rep = Ct\ s\ (Ty\ fun1\ [Ty\ fun2\ [\tau, Ty\ propT1\ Nil], Ty\ propT2\ Nil])\ \$\ b$
and *res*: $res = (b \cdot t')$

by (*auto split*: *term.splits typ.splits list.splits if-splits option.splits simp add*: *Let-def*)

have *ctxt*: *finite* $(set\ Hs)$ $\forall A \in set\ Hs\ .\ term\text{-}ok\ thy\ A$ $\forall A \in set\ Hs\ .\ typ\text{-}of\ A = Some\ propT$
using *4* **by** *auto*

show *?case*
using *conds 4.prems ctxt*
by (*auto simp add*: *res wt-term-def simp del*: *FV-def*
intro!: *forall-elim'*[*OF 4.prems*(*1*) - - - *ctxt*] *4.IH*)
next
case (*5 thy vs ns Hs t p*)
from *this* **obtain** t' *rep* **where**
conds: $subst\text{-}bvs\ (map\ (\lambda(x,y)\ .\ Fv\ x\ y)\ vs)\ t = t'$

replay′ thy vs ns (t′#Hs) p = Some rep
typ-of t′ = Some propT term-ok thy t′
and *res*: *res = mk-imp t′ rep*
by (*auto split*: *term.splits typ.splits list.splits if-splits option.splits simp add*: *Let-def*)

show *?case*
proof (*cases t′∈ set Hs*)
case *True*
hence *s*: *set Hs = set (t′ # Hs)* **by** *auto*
hence *s′*: *set Hs = insert t′ (set Hs −{t′})* **by** *auto*

have *thy,set (t′ # Hs) ⊢ rep*
apply (*rule 5.IH*)
using *conds(4) 5.prems True* **by** (*auto simp add*: *conds(1) conds(2)*[*symmetric*] *conds(3)*)
hence *thy,set Hs − {t′} ⊢ t′ ⟼ rep*
using *implies-intro 5.prems(1) 5.prems(4) conds(3) conds(4) s*
using *has-typ-iff-typ-of term-ok′-imp-wf-term term-okD1* **by** *presburger*
then show *?thesis*
apply (*subst res*)
apply (*subst s′*)
apply (*rule weaken-proves*)
using *conds(3−4)* **by** *blast+*
next
case *False*
hence *s*: *set Hs = insert t′ (set Hs) − {t′}* **by** *auto*

have *FV (set (map (λ(x,y) . Fv x y) vs)) = set vs* **by** (*induction vs*) *auto*
hence *frees-bound*: *fv t′ ⊆ fv t ∪ set vs*
using *fv-subst-bvs1-upper-bound subst-bvs-def* **by** (*fastforce simp add*: *conds(1)*[*symmetric*])

have *pre*: *thy,set (t′ # Hs) ⊢ rep*
apply (*rule 5.IH*)
using *5.prems(5−8) conds(3−4) frees-bound*
by (*auto simp add*: *5.prems(1−4) conds(1) conds(2) image-subset-iff simp del*: *term-ok-def*)

show *?thesis*
apply (*subst res*) **apply** (*subst s*)
apply (*rule proves.implies-intro*; *use 5 conds* **in** ‹(*solves ‹simp add*: *wt-term-def›*) *?›*)
using *pre* **by** *simp*
qed
next
case (*6 thy vs ns Hs p1 p2*)
from ‹*replay′ thy vs ns Hs (AppP p1 p2) = Some res*› **obtain** *fn1 fn2 prp1 prp2 prp3 A B A′ imp*
where

conds: *Option.bind* (*replay′ thy vs ns Hs p1*) *beta-eta-norm*
= *Some* (*Ct imp* (*Ty fn1* [*Ty prp1* [], *Ty fn2* [*Ty prp2* [], *Ty prp3* []]]) $ *A* $ *B*)
Option.bind (*replay′ thy vs ns Hs p2*) *beta-eta-norm* = *Some A′*
imp = *STR ''Pure.imp''* ∧ *fn1* = *STR ''fun''* ∧ *fn2* = *STR ''fun''*
∧ *prp1* = *STR ''prop''* ∧ *prp2* = *STR ''prop''* ∧ *prp3* = *STR ''prop''* ∧ *A*=*A′*
and *res*: *res* = *B*
by (*auto split*: *term.splits typ.splits list.splits if-splits option.splits simp add*: *Let-def*)

obtain *C* **where** *C*: *Option.bind* (*replay′ thy vs ns Hs p1*) *beta-eta-norm* = *Some* (*C* ⟼ *res*)
using *conds res* **by** *blast*
from *this* **obtain** *pre pre-C* **where** *pre*: *replay′ thy vs ns Hs p1* = *Some pre*
and *pre-C*: *replay′ thy vs ns Hs p2* = *Some pre-C*
by (*meson bind-eq-Some-conv conds*(*2*))

from *pre C* **have** *norm-pre*: *beta-eta-norm pre* = *Some* (*C* ⟼ *res*) **by** *simp*
from *pre-C pre C conds* **have** *norm-pre-C*: *beta-eta-norm pre-C* = *Some C* **by** *auto*

have *thy*, *set Hs* ⊢ *pre-C*
by (*rule 6.IH*(*2*)) (*use 6.prems conds* **in** ‹*auto simp add*: *pre pre-C*›)
hence *I1*: *thy*, *set Hs* ⊢ *C*
using *beta-eta-norm-preserves-proves norm-pre-C* ‹*wf-theory thy*›
using *6.prems*(*2*) *6.prems*(*3*) **by** *blast*

have *thy*, *set Hs* ⊢ *pre*
by (*rule 6.IH*(*1*)) (*use 6.prems conds* **in** ‹*auto simp add*: *pre pre-C*›)
hence *I2*: *thy*, *set Hs* ⊢ *C* ⟼ *res*
using *beta-eta-norm-preserves-proves norm-pre* ‹*wf-theory thy*›
using *6.prems*(*2*) *6.prems*(*3*) **by** *blast*

from *I1 I2* **have** *thy*, *set Hs* ∪ *set Hs* ⊢ *res* **using** *proves.implies-elim* **by** *blast*
thus *?case* **by** *simp*
next
case (*7 thy vs ns Hs ty c*)
from *this* **obtain** *fun it ity prop* **where** *conds*: *has-sort* (*osig* (*sig thy*)) *ty* {*c*}
typ-ok thy ty const-type (*sig thy*) (*const-of-class c*)
= *Some* (*Ty fun* [*Ty it* [*ity*], *Ty prop* []]) *ity* = *tvariable STR '''a''*
fun = *STR ''fun''* *prop* = *STR ''prop''* *it* = *STR ''itself''*
and *res*: *res* = (*mk-of-class ty c*)
by (*auto split*: *term.splits typ.splits list.splits if-splits option.splits*)

from *res* **have** *res* = *mk-of-class ty c* **by** *auto*
moreover have *thy*,*set Hs* ⊢ *mk-of-class ty c*
by (*rule proves.of-class*[**where** *T*=*ty*, *OF 7.prems*(*1*)]) (*use conds* **in** *auto*)

ultimately show *?case* **by** *simp*

next
case (*8 thy ux uy Hs n*)
hence *res* ∈ *set Hs*
by (*metis not-None-eq option.inject replay′.simps*(*8*))
then show *?case* **using** *proves.assume 8* **by** (*simp add*: *wt-term-def*)
qed

lemma *finite-fv-Proof*: *finite* (*fv-Proof P*)
by (*induction P*) *auto*

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−
obtain *res′* **where** *res′*: *replay′ thy vs ns Hs P* = *Some res′*
using *replay′-sound-pre assms bind-eq-Some-conv* **by** *metis*
moreover have *beta-eta-norm res′* = *Some res*
using *res′ assms*(*8*) **by** *auto*
moreover have *thy*, *set Hs* ⊢ *res′*
using *res′ assms replay′-sound-pre* **by** *simp*
ultimately show *?thesis*
using *beta-eta-norm-preserves-proves assms*(*1*−*3*) **by** *blast*
qed

lemma
assumes *wf-theory thy*
assumes *replay′′ thy* [] (*fst* ‘ *fv-Proof P*) [] *P* = *Some res*
shows *thy*, *set* [] ⊢ *res*
using *assms finite-fv-Proof replay′-sound-pre replay′′-sound*[**where** *vs*=[]
and *ns*=*fst* ‘ *fv-Proof P* **and** *P*=*P* **and** *Hs*=[]]
by *simp*

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*
  apply (*rule replay''-sound*[**where** *vs*=[] **and** *ns*=(*fst* ‘ (*fv-Proof P* ∪ *FV* (*set* (*hyps P*)))) **and** *P*=*P* **and** *Hs*=*hyps P*]
  ; (*use assms finite-fv-Proof replay'-sound-pre* **in** ‹*solves simp*›)*?*)
  by *blast*+

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*
  using *replay''-sound-pre-hyps assms* **by** (*simp split*: *if-splits*)

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*
  using *check-proof-def replay-sound-pre-hyps* **by** *blast*

lemma *check-proof-really-sound*:
  assumes *check-proof thy P res*
  shows *thy*, *set* (*hyps P*) ⊩ *res*
proof−
  have *wf-theory thy*
    using *assms check-proof-def* **by** *blast*
  moreover have *Some res* = *replay thy P*
    by (*metis assms check-proof-def*)
  moreover hence ∀ *x*∈*set* (*hyps P*) . *term-ok thy x* ∧ *typ-of x* = *Some propT*
    by (*metis not-None-eq replay-def*)

ultimately show *?thesis*
by (*meson assms check-proof-sound has-typ-iff-typ-of proved-terms-well-formed*(*1*) *proves'-def*
term-ok-def wt-term-def)
qed

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*
by (*simp add*: *exe-ars-conds-def*)

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*
by (*auto simp add*: *coregular-tcsigs-def complete-tcsigs-def consistent-length-tcsigs-def all-normalized-and-ex-tcsigs-def sort-ex-def wf-sort-def*)

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*)
by (*induction arss*) (*auto simp add*: *rev-image-eqI*)

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*
by (*force simp add*: *exe-ars-conds-def*)

lemma *exe-ars-conds arss*
⟹ (*name*, *arsl*) ∈ *set arss* ∧ *map-of arsl* = *ars*
⟹ *map-of* (*map* (*apsnd map-of*) *arss*) *name* = *Some ars*
by (*force simp add*: *exe-ars-conds-def*)

lemma *consistent-length-tcsigs-imp-exe-consistent-length-tcsigs*:
exe-ars-conds arss ⟹ *consistent-length-tcsigs* (*translate-ars arss*)
⟹ *exe-consistent-length-tcsigs arss*
unfolding *consistent-length-tcsigs-def exe-consistent-length-tcsigs-def*
apply (*clarsimp simp add*: *exe-ars-conds-def*)
by (*metis in-alist-imp-in-map-of map-of-is-SomeI ranI snd-conv translate-ars.simps*)

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−
{
fix *ars ss*$_1$ *ss*$_2$
assume *p*: *ars* ∈ *ran* (*map-of* (*map* (*apsnd map-of*) *arss*)) *ss*$_1$ ∈ *ran ars ss*$_2$ ∈ *ran ars*
from *p*(*1*) **obtain** *name* **where** *map-of* (*map* (*apsnd map-of*) *arss*) *name* = *Some ars*
by (*meson in-range-if-ex-key*)
from *this* **obtain** *arsl* **where** (*name*, *arsl*) ∈ *set arss map-of arsl* = *ars*
using *assms*(*1*) **by** (*auto simp add*: *exe-ars-conds-def*)
from *this* **obtain** *c1 c2* **where** *ars c1* = *Some ss*$_1$ *ars c2* = *Some ss*$_2$
by (*metis in-range-if-ex-key p*(*2*) *p*(*3*))

hence (*c1*, ss_1) ∈ *set arsl* (*c2*, ss_2) ∈ *set arsl*
by (*simp-all add*: ‹*map-of arsl* = *ars*› *map-of-SomeD*)
hence *length* ss_1 = *length* ss_2
using *assms*(*2*) ‹(*name*, *arsl*) ∈ *set arss*›
by (*fastforce simp add*: *exe-consistent-length-tcsigs-def*)
}
note *1* = *this*
show *?thesis*
by (*simp add*: *consistent-length-tcsigs-def exe-consistent-length-tcsigs-def*) (*use 1* **in** *blast*)
qed

lemma *consistent-length-tcsigs-iff-exe-consistent-length-tcsigs*:
exe-ars-conds arss ⟹
consistent-length-tcsigs (*translate-ars arss*) ⟷ *exe-consistent-length-tcsigs arss*
using *consistent-length-tcsigs-imp-exe-consistent-length-tcsigs*
exe-consistent-length-tcsigs-imp-consistent-length-tcsigs **by** *blast*

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−
{
fix *ars a b y*
assume *p*: *ars* ∈ *ran* (*map-of* (*map* (*apsnd map-of*) *arss*))
(*a*, *b*) ∈ *set cs ars a* = *Some y*

from *p*(*1*) **obtain** *name* **where** *map-of* (*map* (*apsnd map-of*) *arss*) *name* = *Some ars*
by (*meson in-range-if-ex-key*)
from *this* **obtain** *arsl* **where** (*name*, *arsl*) ∈ *set arss map-of arsl* = *ars*
using *assms*(*1*) **by** (*auto simp add*: *exe-ars-conds-def*)
hence (*a*, *y*) ∈ *set arsl*
by (*simp add*: *map-of-SomeD p*(*3*))
hence∃ *y*. *ars b* = *Some y*
using *assms*(*2*) ‹(*name*, *arsl*) ∈ *set arss*›
apply (*clarsimp simp add*: *exe-complete-tcsigs-def*)
by (*metis* (*no-types*, *lifting*) ‹*map-of arsl* = *ars*› *case-prodD domD domI dom-map-of-conv-image-fst*
p(*2*) *p*(*3*) *snd-conv*)
}
note *1* = *this*
show *?thesis*
by (*simp add*: *complete-tcsigs-def exe-complete-tcsigs-def*) (*use 1* **in** *blast*)

qed

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*
unfolding *complete-tcsigs-def exe-complete-tcsigs-def exe-ars-conds-def*
by (*metis* (*mono-tags*, *lifting*) *case-prod-unfold dom-map-of-conv-image-fst in-alist-imp-in-map-of in-range-if-ex-key map-of-SomeD ran-distinct*)

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*
using *exe-complete-tcsigs-imp-complete-tcsigs complete-tcsigs-imp-exe-complete-tcsigs*
by *blast*

definition *exe-coregular-tcsigs* (*cs* :: (*class* $\times$ *class*) *list*) *arss*
$\equiv$ ($\forall$ *ars* $\in$ *snd* ‘ *set arss* .
$\forall$ c_1 $\in$ *fst* ‘ *set ars*. $\forall$ c_2 $\in$ *fst* ‘ *set ars*.
(*class-leq* (*set cs*) c_1 c_2 $\longrightarrow$
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–
{
fix *ars* c_1 c_2 *ss1 ss2*
assume *p*: *ars* $\in$ *ran* (*map-of* (*map* (*apsnd map-of*) *arss*)) *ars* c_1 = *Some ss1* *ars* c_2 = *Some ss2*
class-leq (*set cs*) c_1 c_2
from *p*(*1*) **obtain** *name* **where** *map-of* (*map* (*apsnd map-of*) *arss*) *name* = *Some ars*
by (*meson in-range-if-ex-key*)
from *this* **obtain** *arsl* **where** (*name*, *arsl*) $\in$ *set arss map-of arsl* = *ars*
using *assms*(*1*) **by** (*auto simp add*: *exe-ars-conds-def*)
from *this* **obtain** *c1 c2* **where** *ars c1* = *Some ss1 ars c2* = *Some ss2 class-leq* (*set cs*) *c1 c2*
using *p*(*2*) *p*(*3*) *p*(*4*) **by** *blast*
hence (*c1*, *ss1*) $\in$ *set arsl* (*c2*, *ss2*) $\in$ *set arsl*
by (*simp-all add*: ‹*map-of arsl* = *ars*› *map-of-SomeD*)
hence *lookup* (λ*x*. *x*=*c1*) *arsl* = *Some ss1 lookup* (λ*x*. *x*=*c2*) *arsl* = *Some ss2*
by (*metis* ‹(*name*, *arsl*) $\in$ *set arss*› *assms*(*1*) *exe-ars-conds-def* *image-eqI lookup-present-eq-key snd-conv*)+
hence *list-all2* (*sort-leq* (*set cs*)) *ss1 ss2*
using *assms*(*2*) ‹(*name*, *arsl*) $\in$ *set arss*› ‹(*c1*, *ss1*) $\in$ *set arsl*› ‹(*c2*, *ss2*) $\in$ *set arsl*›
‹*class-leq* (*set cs*) *c1 c2*›
by (*fastforce simp add*: *exe-coregular-tcsigs-def*)

```
  }
  note 1 = this
  show ?thesis
    by (auto simp add: coregular-tcsigs-def exe-coregular-tcsigs-def) (use 1 in blast)

qed

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−
  {
    fix name ars c1 ss1 c2 ss2
    assume p: (name, ars) ∈ set arss (c1, ss1) ∈ set ars (c2, ss2) ∈ set ars
      class-leq (set cs) c1 c2

    have s1: (lookup (λx. x = c1) ars) = Some ss1
    using assms(1) lookup-present-eq-key p(1) p(2) by (force simp add: exe-ars-conds-def)
    have s2: (lookup (λx. x = c2) ars) = Some ss2
    using assms(1) lookup-present-eq-key p(1) p(3) by (force simp add: exe-ars-conds-def)
     have list-all2 (sort-leq (set cs)) (the (lookup (λx. x = c1) ars)) (the (lookup
(λx. x = c2) ars))
      using assms apply (simp add: coregular-tcsigs-def s1 s2 exe-ars-conds-def)
       by (metis domIff in-alist-imp-in-map-of map-of-is-SomeI option.distinct(1)
option.sel
          p(1) p(2) p(3) p(4) ranI snd-conv translate-ars.simps)
  }
  note 1 = this
  show ?thesis
    by (auto simp add: coregular-tcsigs-def exe-coregular-tcsigs-def) (use 1 in blast)
qed

lemma coregular-tcsigs-iff-exe-coregular-tcsigs:
  exe-ars-conds arss ⟹ coregular-tcsigs (set cs) (translate-ars arss) ⟷ exe-coregular-tcsigs
cs arss
  using coregular-tcsigs-imp-exe-coregular-tcsigs exe-coregular-tcsigs-imp-coregular-tcsigs
by blast

lemma wf-subclass sub ⟹ Field sub = Domain sub
  using refl-on-def by fastforce

definition [simp]: exefield rel = List.union (map fst rel) (map snd rel)
lemma Field-set-code: Field (set rel) = set (exefield rel)
  by (induction rel) fastforce+

lemma class-ex-rec: finite r ⟹ class-ex (insert (a,b) r) c = (a=c ∨ b=c ∨ class-ex
r c)
  by (induction r rule: finite-induct) (auto simp add: class-ex-def)
```

definition [*simp*]: *execlass-ex rel c* = *List.member* (*exefield rel*) *c*
lemma *execlass-ex-code*: *class-ex* (*set rel*) *c* = *execlass-ex rel c*
by (*metis Field-set-code class-ex-def execlass-ex-def in-set-member*)

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*
by (*simp add*: *execlass-ex-code sort-ex-class-ex*)

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*
by (*simp add*: *class-leq-def class-les-def member-def*)

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*
by (*auto simp add*: *normalize-sort-def List.member-def list-ex-iff class-leq-def class-les-def*)

lemma *normalized-sort-code*[*code*]: *normalized-sort* (*set cs*) *s* = *exenormalized-sort cs s*
using *exenormalized-sort-def normalize-sort-code* **by** *presburger*

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*
using *normalized-sort-code sort-ex-code assms*
by (*simp add*: *sort-ex-code wf-sort-def*)

declare *exewf-sort-def*[*code del*]
lemma [*code*]: *exewf-sort sub S* ≡ (*S* = {} ∨ *exenormalized-sort sub S* ∧ *exesort-ex sub S*)
by *simp* (*smt ball-empty bot-set-def empty-Collect-eq*)

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−
have *ac*: *alist-conds arss*
using *assms*(*1*) *exe-ars-conds-def* **by** *blast*
{

fix *s ars*
assume *a1*: (*s*, *ars*) ∈ *set arss*
fix *c Ss*
assume *a2*: (*c*,*Ss*) ∈ *set ars*
fix *S*
assume *a3*: *S* ∈ *set Ss*

have *map-of ars* ∈ *ran* (*map-of* (*map* (*apsnd map-of*) *arss*))
using *ac a1* **by** (*metis in-alist-imp-in-map-of ranI translate-ars.simps*)
moreover have *Ss* ∈ *ran* (*map-of ars*)
using *a1 a2 assms*(*1*) **by** (*metis exe-ars-conds-def map-of-is-SomeI ranI ran-distinct*)
ultimately have *wf-sort* (*set cs*) *S*
using *assms*(*2*) *a1 a2 a3* **by** (*auto simp add*: *all-normalized-and-ex-tcsigs-def*)
}
thus *?thesis*
using *normalize-sort-code wf-sort-def*
by (*clarsimp simp add*: *all-normalized-and-ex-tcsigs-def exe-all-normalized-and-ex-tcsigs-def exe-ars-conds-def wf-sort-def wf-sort-code normalize-sort-def sort-ex-code*)
qed

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−
{
fix *ars ss s*
assume *p*: *ars* ∈ *ran* (*map-of* (*map* (*apsnd map-of*) *arss*))
ss ∈ *ran ars s* ∈ *set ss*

from *p*(*1*) **obtain** *name* **where** *map-of* (*map* (*apsnd map-of*) *arss*) *name* = *Some ars*
by (*meson in-range-if-ex-key*)
from *this* **obtain** *arsl* **where** (*name*, *arsl*) ∈ *set arss map-of arsl* = *ars*
using *assms*(*1*) **by** (*auto simp add*: *exe-ars-conds-def*)
from *this* **obtain** *c* **where** *c*: *ars c* = *Some ss*
using *in-range-if-ex-key p*(*2*) **by** *force*
have *exewf-sort cs s*
by (*metis* (*no-types*, *opaque-lifting*) ‹(*name*, *arsl*) ∈ *set arss*› ‹*map-of arsl* = *ars*› *assms*(*1*) *assms*(*2*)
exe-all-normalized-and-ex-tcsigs-def exe-ars-conds-def image-iff p(*2*) *p*(*3*) *ran-distinct snd-conv*)
hence *wf-sort* (*set cs*) *s*
by (*simp add*: *normalize-sort-code sort-ex-code wf-sort-def*)
}
note *1* = *this*
show *?thesis*
using *1* **by** (*clarsimp simp add*: *wf-sort-def all-normalized-and-ex-tcsigs-def*

exe-all-normalized-and-ex-tcsigs-def)
qed

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*
using *all-normalized-and-ex-tcsigs-imp-exe-all-normalized-and-ex-tcsigs exe-all-normalized-and-ex-tcsigs-imp*
by *blast*

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*
using *all-normalized-and-ex-tcsigs-iff-exe-all-normalized-and-ex-tcsigs*
consistent-length-tcsigs-imp-exe-consistent-length-tcsigs
coregular-tcsigs-iff-exe-coregular-tcsigs exe-complete-tcsigs-iff-complete-tcsigs
exe-consistent-length-tcsigs-imp-consistent-length-tcsigs exe-wf-tcsigs-def wf-tcsigs-def

by *blast*

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*)
by (*induction l*) (*auto simp add*: *antisym-def*)

lemma *antisym-imp-exe-antisym*: *antisym* (*set l*) ⟹ *exe-antisym l*
proof (*induction l*)
case *Nil*
then show *?case*
by *simp*
next
case (*Cons a l*)
then show *?case*
by (*simp add*: *antisym-def*) (*metis exe-antisym.simps*(*2*) *surj-pair*)
qed

lemma *antisym-iff-exe-antisym*: *antisym* (*set l*) = *exe-antisym l*
using *antisym-imp-exe-antisym exe-antisym-imp-antisym* **by** *blast*

definition *exe-wf-subclass 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*

by (*simp add*: *antisym-iff-exe-antisym exe-wf-subclass-def*)

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*)
using *exe-coregular-tcsigs-imp-coregular-tcsigs exe-complete-tcsigs-imp-complete-tcsigs*
exe-complete-tcsigs-imp-complete-tcsigs exe-all-normalized-and-ex-tcsigs-imp-all-normalized-and-ex-tcsigs
exe-consistent-length-tcsigs-imp-consistent-length-tcsigs
by (*cases oss*) (*auto simp add*: *exe-wf-subclass-def exe-antisym-imp-antisym*
exe-osig-conds-def)

lemma *classes-translate*: *exe-osig-conds oss* ⟹ *subclass* (*translate-osig oss*) = *set*
(*execlasses oss*)
by (*cases oss*) *simp-all*

lemma *tcsigs-translate*: *exe-osig-conds oss*
⟹ *tcsigs* (*translate-osig oss*) = *translate-ars* (*exetcsigs oss*)
by (*cases oss*) *simp-all*

lemma *wf-osig-translate-imp-exe-osig-conds*:
wf-osig (*translate-osig oss*) ⟹ *exe-osig-conds oss*
using *illformed-osig-not-wf-osig* **by** (*metis translate-osig.elims*)

lemma *wf-osig-imp-exe-wf-osig*:
assumes *wf-osig* (*translate-osig oss*) **shows** *exe-wf-osig oss*
apply (*cases translate-osig oss*)
using *classes-translate tcsigs-translate assms wf-osig-translate-imp-exe-osig-conds*

by (*metis* (*full-types*) *exe-osig-conds-def exe-wf-osig-def subclass.simps tcsigs.simps*
wf-classes-iff-exe-wf-classes wf-osig.simps wf-tcsigs-iff-exe-wf-tcsigs)

lemma *wf-osig-iff-exe-wf-osig*: *wf-osig* (*translate-osig oss*) ⟷ *exe-wf-osig oss*
using *exe-wf-osig-imp-wf-osig wf-osig-imp-exe-wf-osig* **by** *blast*

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*

$\times$ *sort*) $\rightharpoonup$ *typ*) *option*
where
raw-match (*Tv v S*) *T subs* =
(*case subs* (*v*,*S*) *of*
None $\Rightarrow$ *Some* (*subs*((*v*,*S*) := *Some T*))
| *Some U* $\Rightarrow$ (*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* $\Rightarrow$ *typ* $\Rightarrow$ ((*variable* $\times$ *sort*) $\rightharpoonup$ *typ*) $\Rightarrow$ ((*variable* $\times$ *sort*) $\rightharpoonup$ *typ*) *option*
where
raw-match′ (*Tv v S*) *T subs* =
(*case subs* (*v*,*S*) *of*
None $\Rightarrow$ *Some* (*subs*((*v*,*S*) := *Some T*))
| *Some U* $\Rightarrow$ (*if U* = *T then Some subs else None*))
| *raw-match*′ (*Ty a Ts*) (*Ty b Us*) *subs* =
(*if a*=*b* $\wedge$ *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*)
by *pat-completeness auto*
termination proof (*relation measure* (λ(*T*, *U*, *subs*) . *size T* + *size U*), *goal-cases*)
case *1*
then show *?case*
by *auto*
next
case (*2 a Ts b Us subs x xa y xb aa*)
hence *length Ts* = *length Us a*=*b*
by *auto*
from *this 2*(*2*−) **show** *?case*
by (*induction Ts Us rule*: *list-induct2*) *auto*
qed

lemma *length-neq-imp-not-raw-matches*: *length Ts* $\neq$ *length Us* $\Longrightarrow$ *raw-matches Ts Us subs* = *None*
by (*induction Ts Us subs rule*: *raw-match-raw-matches.induct*(*2*) [**where** *P* = λ*T U subs* . *True*])
(*auto cong*: *Option.bind-cong*)

lemma *raw-match T U subs* = *raw-match*′ *T U subs*

proof (*induction T U subs rule*: *raw-match-raw-matches.induct*(*1*)
[**where** *Q* = λ*Ts Us subs* . *raw-matches Ts Us subs*
= (*if length Ts* = *length Us*
then fold (λ(*T*, *U*) *subs* . *Option.bind subs* (*raw-match*′ *T U*)) (*zip Ts Us*)
(*Some subs*)
else None)])
case (*4 T Ts U Us subs*)
then show *?case*
proof (*cases raw-match T U subs*)
case *None*
then show *?thesis*
proof (*cases length Ts* = *length Us*)
case *True*
then show *?thesis* **using** *4 None* **by** (*induction Ts Us rule*: *list-induct2*) *auto*
next
case *False*
then show *?thesis* **using** *4 None length-neq-imp-not-raw-matches* **by** *auto*
qed
next
case (*Some a*)
then show *?thesis* **using** *4* **by** *auto*
qed
qed *simp-all*

lemma *raw-match*′*-map-le*: *raw-match*′ *T U subs* = *Some subs*′ ⟹ *map-le subs subs*′
proof (*induction T U subs arbitrary*: *subs*′ *rule*: *raw-match*′*.induct*)
case (*2 a Ts b Us subs*)
have *length Ts* = *length Us*
using *2.prems* **by** (*auto split*: *if-splits*)
moreover have *I*: (*a*,*b*) ∈ *set* (*zip Ts Us*) ⟹ *raw-match*′ *a b subs* = *Some subs*′
⟹ *subs* ⊆$_m$ *subs*′
for *a b subs subs*′
using *2.prems* **by** (*auto split*: *if-splits intro*: *2.IH*)
ultimately show *?case* **using** *2.prems*
proof (*induction Ts Us arbitrary*: *subs subs*′ *rule*: *rev-induct2*)
case *Nil*
then show *?case*
by (*auto split*: *if-splits*)
next
case (*snoc x xs y ys*)
then show *?case*
using *map-le-trans* **by** (*fastforce split*: *if-splits prod.splits simp add*: *bind-eq-Some-conv*)
qed
qed (*auto simp add*: *map-le-def split*: *if-splits option.splits*)

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′*
using *fold-matches-first-step-not-None assms* .
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′*
using *fold-matches-last-step-not-None assms* .

corollary *raw-match′-Type-conds*:
assumes *raw-match′* (*Ty a Ts*) (*Ty b Us*) *subs* = *Some subs′*
shows *a*=*b length Ts* = *length Us*
using *assms* **by** (*auto split*: *if-splits*)

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*
using *assms fold-matches-first-step-not-None*
by (*metis option.discI*)

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*
using *assms fold-matches-first-step-not-None′ raw-match′-Type-conds*
by (*metis* (*no-types*, *lifting*) *length-Cons nat.simps*(*1*) *raw-match′.simps*(*2*))

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*
using *assms fold-Option-bind-eq-Some-at-point-not-None′* **by** (*fastforce split*: *if-splits*)

lemma *raw-match′-tvsT-subset-dom-res*: *raw-match′ T U subs* = *Some subs′* ⟹ *tvsT T* ⊆ *dom subs′*
proof (*induction T U subs arbitrary*: *subs′ rule*: *raw-match′.induct*)

```
  case (2 a Ts b Us subs)
  have l: length Ts = length Us a = b using 2
    by (metis option.discI raw-match'.simps(2))+

  from this 2 have better-IH:
    (x, y) ∈ set (zip Ts Us) ⟹ raw-match' x y subs = Some subs' ⟹ tvsT x ⊆
dom subs'
    for x y subs subs' by simp
  from l 2.prems better-IH show ?case
  proof (induction Ts Us arbitrary: a b subs subs' rule: list-induct2)
    case Nil
    then show ?case by simp
  next
    case (Cons x xs y ys)
    obtain point where point: raw-match' x y subs = Some point
      and rest: raw-match' (Ty a xs) (Ty b ys) point = Some subs'
    by (metis (no-types, lifting) Cons.hyps Cons.prems(1) Cons.prems(2) fold-matches-first-step-not-None

          raw-match'.simps(2) raw-match'-Type-conds(2))

    have tvsT (Ty a xs) ⊆ dom subs'
      apply (rule Cons.IH[of - b point])
      using Cons.prems rest apply blast+
      by (metis Cons.prems(3) list.set-intros(2) zip-Cons-Cons)
    moreover have tvsT x ⊆ dom point
      by (metis Cons.prems(3) list.set-intros(1) point zip-Cons-Cons)
    moreover have dom point ⊆ dom subs'
      using map-le-implies-dom-le raw-match'-map-le rest by blast
    ultimately show ?case
      by auto
  qed
qed (auto split: option.splits if-splits prod.splits simp add: bind-eq-Some-conv)

lemma raw-match'-dom-res-subset-tvsT:
  raw-match' T U subs = Some subs' ⟹ dom subs' ⊆ tvsT T ∪ dom subs
proof (induction T U subs arbitrary: subs' rule: raw-match'.induct)
  case (2 a Ts b Us subs)
  have l: length Ts = length Us a = b using 2
    by (metis option.discI raw-match'.simps(2))+

  from this 2 have better-IH:
    (x, y) ∈ set (zip Ts Us) ⟹ raw-match' x y subs = Some subs'
      ⟹ dom subs' ⊆ tvsT x ∪ dom subs
    for x y subs subs' by blast
  from l 2.prems better-IH show ?case
  proof (induction Ts Us arbitrary: a b subs subs' rule: list-induct2)
    case Nil
    then show ?case by simp
  next
```

case (*Cons x xs y ys*)
obtain *point* **where** *first*: *raw-match′ x y subs = Some point*
and *rest*: *raw-match′* (*Ty a xs*) (*Ty b ys*) *point = Some subs′*
by (*metis* (*no-types*, *lifting*) *Cons.hyps Cons.prems*(*1*) *Cons.prems*(*2*) *fold-matches-first-step-not-None raw-match′.simps*(*2*) *raw-match′-Type-conds*(*2*))

from *first* **have** *dom point* ⊆ *tvsT x* ∪ *dom subs*
using *Cons.prems*(*3*) **by** *fastforce*
moreover have *dom subs′* ⊆ *tvsT* (*Ty a xs*) ∪ *dom point*
apply (*rule Cons.IH*)
using *Cons.prems*(*1*) **apply** *simp*
using *Cons.prems*(*2*) *rest* **apply** *simp*
by (*metis Cons.prems*(*3*) *list.set-intros*(*2*) *zip-Cons-Cons*)

ultimately show *?case* **using** *Cons.prems in-mono*
apply (*clarsimp split*: *option.splits if-splits prod.splits simp add*: *bind-eq-Some-conv domIff*)
by (*smt UN-iff Un-iff domIff in-mono option.distinct*(*1*))

qed
qed (*auto split*: *option.splits if-splits prod.splits simp add*: *bind-eq-Some-conv*)

corollary *raw-match′-dom-res-eq-tvsT*:
raw-match′ T U subs = Some subs′ ⟹ *dom subs′ = tvsT T* ∪ *dom subs*
by (*simp add*: *map-le-implies-dom-le raw-match′-tvsT-subset-dom-res raw-match′-dom-res-subset-tvsT raw-match′-map-le subset-antisym*)

corollary *raw-match′-dom-res-eq-tvsT-empty*:
raw-match′ T U (λ*x*. *None*) = *Some subs′* ⟹ *dom subs′ = tvsT T*
using *raw-match′-dom-res-eq-tvsT* **by** *simp*

lemma *raw-match′-map-defined*: *raw-match′ T U subs = Some subs′* ⟹ *p*∈*tvsT T* ⟹ *subs′ p* ~= *None*
using *raw-match′-dom-res-eq-tvsT* **by** *blast*

lemma *raw-match′-extend-map-preserve*:
raw-match′ T U subs = Some subs′ ⟹ *map-le subs′ subs″* ⟹ *p*∈*tvsT T* ⟹ *subs″ p = subs′ p*
using *raw-match′-dom-res-eq-tvsT domIff map-le-implies-dom-le*
by (*simp add*: *map-le-def*)

abbreviation *convert-subs subs* ≡ (λ*v 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′*)
by (*induction T*) *auto*

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'')
apply (*rule map-eq-on-tvsT-imp-map-eq-on-typ*)
using *raw-match'-extend-map-preserve assms* **by** *metis*

lemma *raw-match'-produces-matcher*:
raw-match' T U subs = Some subs'
⟹ tsubstT T (convert-subs subs') = U
proof (*induction T U subs arbitrary*: *subs' rule*: *raw-match'.induct*)
case (*2 a Ts b Us subs*)
hence *l*: *length Ts = length Us a=b* **by** (*simp-all split*: *if-splits*)
from *this 2* **have** *better-IH*:
(*x*, *y*) ∈ *set* (*zip Ts Us*) *⟹ raw-match' x y subs = Some subs'*
⟹ tsubstT x (*convert-subs subs'*) = *y*
for *x y subs subs'* **by** *simp*
from *l better-IH* **show** *?case* **using** *2*
proof(*induction Ts Us arbitrary*: *subs subs' rule*: *list-induct2*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons x xs y ys*)
obtain *point* **where** *first*: *raw-match' x y subs = Some point*
and *rest*: *raw-match'* (*Ty a xs*) (*Ty b ys*) *point = Some subs'*
by (*metis* (*no-types*, *lifting*) *Cons.hyps Cons.prems*(*4*) *fold-matches-first-step-not-None l*(*2*) *length-Cons raw-match'.simps*(*2*))

have *tsubstT x* (*convert-subs point*) = *y*
using *Cons.prems*(*2*) *first* **by** *auto*
moreover have *map-le point subs'*
using *raw-match'-map-le rest* **by** *blast*
ultimately have *subs'-hd*: *tsubstT x* (*convert-subs subs'*) = *y*
using *raw-match'-extend-map-preserve' first* **by** *simp*

show *?case* **using** *Cons* **by** (*auto simp add*: *bind-eq-Some-conv subs'-hd first*)
qed
qed (*auto split*: *option.splits if-splits prod.splits simp add*: *bind-eq-Some-conv*)

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 (*induction T arbitrary*: *U ϱ*)
case (*Ty a Ts*)
then show *?case*
proof (*induction Ts arbitrary*: *U*)
case *Nil*
then show *?case* **by** *auto*
next

```
    case (Cons T Ts)
    then show ?case
      by auto
  qed
qed auto

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'
 using assms proof (induction T U subs arbitrary: lesubs subs' rule: raw-match'.induct)
  case (1 v S T subs lesubs subs')
  then show ?case
     by (force split: option.splits if-splits prod.splits simp add: bind-eq-Some-conv
map-le-def
        intro!: domI)
next
  case (2 a Ts b Us subs)
  hence l: length Ts = length Us a=b by (simp-all split: if-splits)
  from this 2 have better-IH:
    (x, y) ∈ set (zip Ts Us) ⟹ raw-match' x y subs = Some subs'
   ⟹ lesubs ⊆_m subs ⟹ ∃ lesubs'. raw-match' x y lesubs = Some lesubs' ∧ lesubs'
⊆_m subs'
    for x y subs lesubs subs' by simp
  from l better-IH show ?case using 2
  proof(induction Ts Us arbitrary: subs lesubs subs' rule: list-induct2)
    case Nil
    then show ?case by simp
  next
    case (Cons x xs y ys)
    obtain point where first: raw-match' x y subs = Some point
      and rest: raw-match' (Ty a xs) (Ty b ys) point = Some subs'
    by (metis (no-types, lifting) Cons.hyps Cons.prems(4) fold-matches-first-step-not-None
l(2) length-Cons raw-match'.simps(2))

    have ∃ lepoint. raw-match' x y lesubs = Some lepoint ∧ lepoint ⊆_m point
      using Cons first by auto
    from this obtain lepoint where
        comp-lepoint: raw-match' x y lesubs = Some lepoint and le-lepoint: lepoint
⊆_m point
      by auto

    have ∃ lesubs'. raw-match' (Ty a xs) (Ty b ys) lepoint = Some lesubs' ∧ lesubs'
⊆_m subs'
      using Cons rest le-lepoint by auto
    from this obtain lesubs' where
        comp-lesubs': raw-match' (Ty a xs) (Ty b ys) lepoint = Some lesubs'
        and le-lesubs': lesubs' ⊆_m subs'
      by auto
```

```
    show ?case using Cons.prems Cons.hyps comp-lepoint comp-lesubs' le-lesubs'
by auto
  qed
qed (auto split: option.splits if-splits prod.splits simp add: bind-eq-Some-conv)

lemma map-le-same-dom-imp-same-map: dom f = dom g ⟹ map-le f g ⟹ f =
g
  by (simp add: map-le-antisym map-le-def)

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−
  have dom subs' = tvsT T
    using assms(1) assms(2) raw-match'-dom-res-eq-tvsT by auto
  moreover obtain lesubs' where raw-match' T U lesubs = Some lesubs' map-le
lesubs' subs'
    using raw-match'-imp-raw-match'-on-map-le assms(1) assms(3) by blast
  moreover hence dom lesubs' = tvsT T
   using ‹dom subs' = tvsT T› map-le-implies-dom-le raw-match'-tvsT-subset-dom-res
by fastforce

  ultimately show ?thesis using map-le-same-dom-imp-same-map by metis
qed

corollary raw-match' T U subs = Some subs' ⟹ dom subs ⊆ tvsT T ⟹
  raw-match' T U (λp . None) = Some subs'
  using map-le-empty map-le-produces-same-raw-match' by blast

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)
using assms proof (induction T U subs arbitrary: restriction subs' rule: raw-match'.induct)
  case (1 v S T subs)
  then show ?case
    apply simp
   by (smt fun-upd-restrict-conv option.case-eq-if option.discI option.sel restrict-fun-upd)
next
  case (2 a Ts b Us subs)
  hence l: length Ts = length Us a=b by (simp-all split: if-splits)
  from this 2 have better-IH:
    (x, y) ∈ set (zip Ts Us) ⟹ raw-match' x y subs = Some subs' ⟹ tvsT x ⊆
restriction
    ⟹ raw-match' x y (subs |‘ restriction) = Some (subs' |‘ restriction)
    for x y subs restriction subs' by simp
```

```
  from l better-IH show ?case using 2
  proof(induction Ts Us arbitrary: subs subs' rule: list-induct2)
    case Nil
    then show ?case by simp
  next
    case (Cons x xs y ys)
    obtain point where first: raw-match' x y subs = Some point
      and rest: raw-match' (Ty a xs) (Ty b ys) point = Some subs'
    by (metis (no-types, lifting) Cons.hyps Cons.prems(4) fold-matches-first-step-not-None
l(2)
          length-Cons raw-match'.simps(2))

    have raw-match' x y (subs |` restriction)
      = Some (point |` restriction)
      using Cons first by simp

    moreover have raw-match' (Ty a xs) (Ty b ys) (point |` restriction)
      = Some (subs' |` restriction)
      using Cons rest by simp

     ultimately show ?case by (simp split: if-splits)
  qed
qed (auto split: option.splits if-splits prod.splits simp add: bind-eq-Some-conv)


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)
  using raw-match'-restriction assms by simp

lemma tinstT-imp-ex-raw-match':
  assumes tinstT T1 T2
  shows ∃subs. raw-match' T2 T1 (λp . None) = Some subs
proof−
  obtain ϱ where tsubstT T2 ϱ = T1 using assms tinstT-def by auto
  hence raw-match' T2 T1 (λ(idx, S). Some (ϱ idx S)) = Some (λ(idx, S). Some
(ϱ idx S))
    using tsubstT-matcher-imp-raw-match'-unchanged by auto

  hence raw-match' T2 T1 ((λ(idx, S). Some (ϱ idx S))|`tvsT T2)
    = Some ((λ(idx, S). Some (ϱ idx S))|`tvsT T2)
    using raw-match'-restriction-on-tvsT by simp
  moreover have dom ((λ(idx, S). Some (ϱ idx S))|`tvsT T2) = tvsT T2 by auto
  ultimately show ?thesis using map-le-produces-same-raw-match'
    using map-le-empty by blast
qed

lemma ex-raw-match'-imp-tinstT:
  assumes ∃subs. raw-match' T2 T1 (λp . None) = Some subs
```

shows *tinstT T1 T2*
proof–
obtain *subs* **where** *raw-match′ T2 T1* (λ*p* . *None*) = *Some subs*
using *assms* **by** *auto*
hence *tsubstT T2* (*convert-subs subs*) = *T1*
using *raw-match′-produces-matcher* **by** *blast*
thus *?thesis* **unfolding** *tinstT-def* **by** *fast*
qed

corollary *tinstT-iff-ex-raw-match′*:
tinstT T1 T2 ⟷ (∃ *subs*. *raw-match′ T2 T1* (λ*p* . *None*) = *Some subs*)
using *ex-raw-match′-imp-tinstT tinstT-imp-ex-raw-match′* **by** *blast*

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*
by *pat-completeness auto*
termination by *size-change*

lemma *raw-match-term-map-le*: *raw-match-term t u subs* = *Some subs′* ⟹ *map-le subs subs′*
by (*induction t u subs arbitrary*: *subs′ rule*: *raw-match-term.induct*)
(*auto split*: *if-splits prod.splits intro*: *map-le-trans raw-match′-map-le simp add*: *bind-eq-Some-conv*)

lemma *raw-match-term-tvs-subset-dom-res*:
raw-match-term t u subs = *Some subs′* ⟹ *tvs t* ⊆ *dom subs′*
proof (*induction t u subs arbitrary*: *subs′ rule*: *raw-match-term.induct*)
case (*4 T t U u subs*)
from *this* **obtain** *bsubs* **where** *bsubs*: *raw-match′ T U subs* = *Some bsubs*
by (*auto simp add*: *bind-eq-Some-conv raw-match′-produces-matcher*)
moreover hence *body*: *raw-match-term t u bsubs* = *Some subs′*
using *4.prems* **by** (*auto simp add*: *bind-eq-Some-conv raw-match′-produces-matcher*)

ultimately have *1*: *tvs t* ⊆ *dom subs′*
using *4* **by** *fastforce*

from *bsubs* **have** *tvsT T* ⊆ *dom bsubs*
using *raw-match′-tvsT-subset-dom-res* **by** *auto*

moreover have *bsubs* $\subseteq_m$ *subs'* **using** *raw-match-term-map-le body* **by** *blast*

ultimately have *2*: *tvsT T* $\subseteq$ *dom subs'*
using *map-le-implies-dom-le* **by** *blast*
then show *?case*
using *4.prems 1 2* **by** (*simp split*: *if-splits*)
next
case (*5 f u f' u' subs*)
from *this* **obtain** *fsubs* **where** *f*: *raw-match-term f f' subs* = *Some fsubs*
by (*auto simp add*: *bind-eq-Some-conv*)
hence *u*: *raw-match-term u u' fsubs* = *Some subs'*
using *5.prems* **by** *auto*

have *1*: *tvs u* $\subseteq$ *dom subs'*
using *f u 5.IH* **by** *auto*

have *tvs f* $\subseteq$ *dom fsubs*
using *5 f* **by** *simp*
moreover have *fsubs* $\subseteq_m$ *subs'* **using** *raw-match-term-map-le u* **by** *blast*
ultimately have *2*: *tvs f* $\subseteq$ *dom subs'*
using *map-le-implies-dom-le* **by** *blast*

then show *?case* **using** *1* **by** *simp*
qed (*use raw-match'-tvsT-subset-dom-res* **in** ‹*auto split*: *option.splits if-splits prod.splits*›)

lemma *raw-match-term-dom-res-subset-tvs*:
raw-match-term t u subs = *Some subs'* $\Longrightarrow$ *dom subs'* $\subseteq$ *tvs t* $\cup$ *dom subs*
proof (*induction t u subs arbitrary*: *subs' rule*: *raw-match-term.induct*)
case (*4 T t U u subs*)
from *this* **obtain** *bsubs* **where** *bsubs*: *raw-match' T U subs* = *Some bsubs*
by (*auto simp add*: *bind-eq-Some-conv raw-match'-produces-matcher*)
moreover hence *body*: *raw-match-term t u bsubs* = *Some subs'*
using *4.prems* **by** (*auto simp add*: *bind-eq-Some-conv raw-match'-produces-matcher*)

ultimately have *1*: *dom subs'* $\subseteq$ *tvs t* $\cup$ *dom bsubs*
using *4* **by** *fastforce*

from *bsubs* **have** *dom bsubs* $\subseteq$ *tvsT T* $\cup$ *dom bsubs*
using *raw-match'-dom-res-subset-tvsT* **by** *auto*

moreover have *subs* $\subseteq_m$ *bsubs* **using** *bsubs raw-match'-map-le* **by** *blast*

ultimately have *2*: *dom bsubs* $\subseteq$ *tvsT T* $\cup$ *dom subs*
using *bsubs raw-match'-dom-res-subset-tvsT* **by** *auto*
then show *?case*
using *4.prems 1 2* **by** (*auto split*: *if-splits*)

next
case (*5 f u f' u' subs*)
from *this* **obtain** *fsubs* **where** *f*: *raw-match-term f f' subs = Some fsubs*
by (*auto simp add*: *bind-eq-Some-conv*)
hence *u*: *raw-match-term u u' fsubs = Some subs'*
using *5.prems* **by** *auto*

have *1*: *dom fsubs ⊆ tvs f ∪ dom subs*
using *5 f u* **by** *simp*

have *dom subs' ⊆ tvs u ∪ dom fsubs*
using *5 f* **by** *simp*
moreover have *fsubs ⊆$_m$ subs'* **using** *raw-match-term-map-le u* **by** *blast*
ultimately have *2*: *dom subs' ⊆ tvs f ∪ tvs u ∪ dom subs*
by (*smt 1 Un-commute inf-sup-aci*(*6*) *subset-Un-eq*)
then show *?case* **using** *1* **by** *simp*
qed (*use raw-match'-dom-res-subset-tvsT* **in** ‹*auto split*: *option.splits if-splits prod.splits*›)

corollary *raw-match-term-dom-res-eq-tvs*:
raw-match-term t u subs = Some subs' ⟹ dom subs' = tvs t ∪ dom subs
by (*simp add*: *map-le-implies-dom-le raw-match-term-tvs-subset-dom-res*
raw-match-term-dom-res-subset-tvs raw-match-term-map-le subset-antisym)

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
using *raw-match-term-dom-res-eq-tvs domIff map-le-implies-dom-le*
by (*simp add*: *map-le-def*)

lemma *map-eq-on-tvs-imp-map-eq-on-term*:
(⋀*p* . *p∈tvs t ⟹ subs p = subs' p*)
⟹ tsubst t (*convert-subs subs*)
= tsubst t (*convert-subs subs'*)
by (*induction t*) (*use map-eq-on-tvsT-imp-map-eq-on-typ* **in** ‹*fastforce+*›)

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-subs subs''*)
apply (*rule map-eq-on-tvs-imp-map-eq-on-term*)
using *raw-match-term-extend-map-preserve assms* **by** *fastforce*

lemma *raw-match-term-produces-matcher*:
raw-match-term t u subs = Some subs'
⟹ tsubst t (*convert-subs subs'*) *= u*
proof (*induction t u subs arbitrary*: *subs' rule*: *raw-match-term.induct*)
case (*4 T t U u subs*)
from *this* **obtain** *bsubs* **where** *bsubs*: *raw-match' T U subs = Some bsubs*

```
    by (auto simp add: bind-eq-Some-conv raw-match'-produces-matcher)
  moreover hence body: raw-match-term t u bsubs = Some subs'
   using 4.prems by (auto simp add: bind-eq-Some-conv raw-match'-produces-matcher)

  ultimately have 1: tsubst t (convert-subs subs') = u
    using 4 by fastforce

  from bsubs have tsubstT T (convert-subs bsubs) = U
    using raw-match'-produces-matcher by blast

  moreover have bsubs ⊆m subs' using raw-match-term-map-le body by blast

  ultimately have 2: tsubstT T (convert-subs subs') = U
    using raw-match'-extend-map-preserve'[OF bsubs, of subs'] by simp

  then show ?case
    using 4.prems 1 2 by (simp split: if-splits)
next
  case (5 f u f' u' subs)
  from this obtain fsubs where f: raw-match-term f f' subs = Some fsubs
    by (auto simp add: bind-eq-Some-conv)
  hence u: raw-match-term u u' fsubs = Some subs'
    using 5.prems by auto

  have 1: tsubst u (convert-subs subs') = u'
    using f u 5.IH by auto

  have tsubst f (convert-subs fsubs) = f'
    using 5 f by simp
  moreover have fsubs ⊆m subs' using raw-match-term-map-le u by blast
  ultimately have 2: tsubst f (convert-subs subs') = f'
    using raw-match-extend-map-preserve'[OF f, of subs'] by simp

  then show ?case using raw-match'-extend-map-preserve' 1 by auto
qed (auto split: if-splits simp add: bind-eq-Some-conv raw-match'-produces-matcher)

lemma ex-raw-match-term-imp-tinst:
  assumes ∃ subs. raw-match-term t2 t1 (λp . None) = Some subs
  shows tinst t1 t2
proof−
  obtain subs where raw-match-term t2 t1 (λp . None) = Some subs
    using assms by auto
  hence tsubst t2 (convert-subs subs) = t1
    using raw-match-term-produces-matcher by blast
  thus ?thesis unfolding tinst-def by fast
qed

lemma tsubst-matcher-imp-raw-match-term-unchanged:
   tsubst t ϱ = u ⟹ raw-match-term t u (λ(idx, S). Some (ϱ idx S)) = Some
```

(λ(*idx*, *S*). *Some* (ϱ *idx S*))
by (*induction t arbitrary*: *u* ϱ) (*auto simp add*: *tsubstT-matcher-imp-raw-match'-unchanged*)

lemma *raw-match-term-restriction*:
assumes *raw-match-term t u subs* = *Some subs'*
assumes *tvs t* ⊆ *restriction*
shows *raw-match-term t u* (*subs*|‘*restriction*) = *Some* (*subs'*|‘*restriction*)
using *assms* **by** (*induction t u subs arbitrary*: *restriction subs' rule*: *raw-match-term.induct*)
(*use raw-match'-restriction* **in**
‹*auto split*: *option.splits if-splits prod.splits simp add*: *bind-eq-Some-conv*›)

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*)
using *raw-match-term-restriction assms* **by** *simp*

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 ∃ *lesubs'*. *raw-match-term t u lesubs* = *Some lesubs'* ∧ *map-le lesubs' subs'*
using *assms* **proof** (*induction t u subs arbitrary*: *lesubs subs' rule*: *raw-match-term.induct*)
case (*4 T t U u subs*)
from *this* **obtain** *bsubs* **where** *bsubs*: *raw-match' T U subs* = *Some bsubs*
by (*auto simp add*: *bind-eq-Some-conv raw-match'-produces-matcher*)
hence *body*: *raw-match-term t u bsubs* = *Some subs'*
using *4.prems* **by** (*auto simp add*: *bind-eq-Some-conv raw-match'-produces-matcher*)

from *bsubs 4* **obtain** *lebsubs* **where**
lebsubs: *raw-match' T U subs* = *Some lebsubs map-le lebsubs bsubs*
using *raw-match'-map-le map-le-trans*
by (*fastforce split*: *if-splits simp add*: *bind-eq-Some-conv raw-match'-produces-matcher*)
from *this* **obtain** *lesubs'* **where**
lesubs':*raw-match-term t u lebsubs* = *Some lesubs' map-le lesubs' subs'*
using *4.prems*
by (*auto split*: *if-splits simp add*: *bind-eq-Some-conv raw-match'-produces-matcher*)

show *?case*
using *lebsubs lesubs' 4* **apply** (*auto split*: *if-splits simp add*: *bind-eq-Some-conv*)
by (*meson raw-match'-imp-raw-match'-on-map-le*)
next
case (*5 f u f' u' subs*)
from *this* **obtain** *fsubs* **where** *f*: *raw-match-term f f' subs* = *Some fsubs*
by (*auto simp add*: *bind-eq-Some-conv*)
hence *u*: *raw-match-term u u' fsubs* = *Some subs'*
using *5.prems* **by** *auto*

from *5* **obtain** *lefsubs* **where**
lefsubs: *raw-match-term f f' subs* = *Some lefsubs map-le lefsubs fsubs*
using *raw-match-term-map-le map-le-trans f* **by** *auto*

from *this* **obtain** *lesubs'* **where**
lesubs':*raw-match-term u u' lefsubs = Some lesubs' map-le lesubs' subs'*
using *5.prems*
by (*auto split*: *if-splits simp add*: *bind-eq-Some-conv raw-match'-produces-matcher*)

from *lefsubs lesubs'* **show** *?case* **using** *5* **by** (*fastforce split*: *if-splits simp add*: *bind-eq-Some-conv*)
qed (*use raw-match'-imp-raw-match'-on-map-le* **in**
‹*auto split*: *option.splits if-splits prod.splits simp add*: *bind-eq-Some-conv*›)

corollary *map-le-produces-same-raw-match-term*:
assumes *raw-match-term t u subs = Some subs'*
assumes *dom subs ⊆ tvs t*
assumes *map-le lesubs subs*
shows *raw-match-term t u lesubs = Some subs'*
proof−
have *dom subs' = tvs t*
using *assms*(*1*) *assms*(*2*) *raw-match-term-dom-res-eq-tvs* **by** *auto*
moreover obtain *lesubs'* **where** *raw-match-term t u lesubs = Some lesubs' map-le lesubs' subs'*
using *raw-match-term-imp-raw-match-term-on-map-le assms*(*1*) *assms*(*3*) **by** *blast*
moreover hence *dom lesubs' = tvs t*
using ‹*dom subs' = tvs t*› *map-le-implies-dom-le raw-match-term-tvs-subset-dom-res* **by** *fastforce*

ultimately show *?thesis* **using** *map-le-same-dom-imp-same-map* **by** *metis*
qed

lemma *tinst-imp-ex-raw-match-term*:
assumes *tinst t1 t2*
shows ∃ *subs. raw-match-term t2 t1* (λ*p* . *None*) = *Some subs*
proof−
obtain ϱ **where** *tsubst t2* ϱ = *t1* **using** *assms tinst-def* **by** *auto*
hence *raw-match-term t2 t1* (λ(*idx*, *S*). *Some* (ϱ *idx S*)) = *Some* (λ(*idx*, *S*). *Some* (ϱ *idx S*))
using *tsubst-matcher-imp-raw-match-term-unchanged* **by** *auto*

hence *raw-match-term t2 t1* ((λ(*idx*, *S*). *Some* (ϱ *idx S*))|‘*tvs t2*)
= *Some* ((λ(*idx*, *S*). *Some* (ϱ *idx S*))|‘*tvs t2*)
using *raw-match-term-restriction-on-tvs* **by** *simp*
moreover have *dom* ((λ(*idx*, *S*). *Some* (ϱ *idx S*))|‘*tvs t2*) = *tvs t2* **by** *auto*
ultimately show *?thesis* **using** *map-le-produces-same-raw-match-term*
using *map-le-empty* **by** *blast*
qed

corollary *tinst-iff-ex-raw-match-term*:
tinst t1 t2 ⟷ (∃ *subs. raw-match-term t2 t1* (λ*p* . *None*) = *Some subs*)
using *ex-raw-match-term-imp-tinst tinst-imp-ex-raw-match-term* **by** *blast*

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*)
by (*pat-completeness*) *auto*
termination proof (*relation measure* (λ(*T*, *U*, *subs*) . *size T* + *size U*), *goal-cases*)
case *1*
then show *?case*
by *auto*
next
case (*2 a Ts b Us subs x xa y xb aa*)
hence *length Ts* = *length Us a*=*b*
by *auto*
from *this 2*(*2*−) **show** *?case*
by (*induction Ts Us rule*: *list-induct2*) *auto*
qed

corollary *assoc-match-Type-conds*:
assumes *assoc-match* (*Ty a Ts*) (*Ty b Us*) *subs* = *Some subs*′
shows *a*=*b length Ts* = *length Us*
using *assms* **by** (*auto split*: *if-splits*)

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*′
using *assms* **apply** (*simp split*: *option.splits*)
by (*metis fold-Option-bind-eq-Some-start-not-None*′ *not-None-eq*)

lemma *assoc-match-subset*: *assoc-match T U subs* = *Some subs*′ ⟹ *set subs* ⊆ *set subs*′
proof (*induction T U subs arbitrary*: *subs*′ *rule*: *assoc-match.induct*)
case (*2 a Ts b Us subs*)

hence *l*: *length Ts = length Us a = b* **by** (*simp-all split*: *if-splits*)
have *better-IH*: *(x, y) ∈ set (zip Ts Us) ⟹*
assoc-match x y subs = Some subs' ⟹ set subs ⊆ set subs'
for *x y subs subs'* **using** *2* **by** (*simp split*: *if-splits*)
from *l better-IH 2.prems* **show** *?case*
proof (*induction Ts Us arbitrary*: *subs rule*: *list-induct2*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons x xs y ys*)

obtain *point* **where** *first*: *assoc-match x y subs = Some point*
and *rest*: *assoc-match (Ty a xs) (Ty b ys) point = Some subs'*
using *fold-assoc-matches-first-step-not-None*
by (*metis* (*no-types*, *lifting*) *Cons.hyps Cons.prems assoc-match.simps*(*2*) *assoc-match-Type-conds*(*2*))

then show *?case*
using *Cons.IH Cons.prems*(*2*) **by** (*fastforce split*: *option.splits prod.splits if-splits*
simp add: *lookup-present-eq-key bind-eq-Some-conv*)
qed
qed (*auto split*: *option.splits prod.splits if-splits simp add*: *lookup-present-eq-key*)

lemma *assoc-match-distinct*: *assoc-match T U subs = Some subs' ⟹ distinct (map fst subs)*
⟹ distinct (map fst subs')
proof (*induction T U subs arbitrary*: *subs' rule*: *assoc-match.induct*)
case (*2 a Ts b Us subs*)
hence *l*: *length Ts = length Us a = b* **by** (*simp-all split*: *if-splits*)
have *better-IH*: *(x, y) ∈ set (zip Ts Us) ⟹*
assoc-match x y subs = Some subs' ⟹ distinct (map fst subs) ⟹ distinct (map fst subs')
for *x y subs subs'* **using** *2* **by** (*simp split*: *if-splits*)
from *l better-IH 2.prems* **show** *?case*
proof (*induction Ts Us arbitrary*: *subs subs' rule*: *list-induct2*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons x xs y ys*)

obtain *point* **where** *first*: *assoc-match x y subs = Some point*
and *rest*: *assoc-match (Ty a xs) (Ty b ys) point = Some subs'*
using *fold-assoc-matches-first-step-not-None*
by (*metis* (*no-types*, *lifting*) *Cons.hyps Cons.prems assoc-match.simps*(*2*) *assoc-match-Type-conds*(*2*))

have *dst-point*: *distinct (map fst point)*
apply (*rule Cons.prems*)

```
    using first Cons.prems by simp-all

   have distinct (map fst subs')
    apply (rule Cons.IH)
    using Cons.prems rest apply simp
    using Cons.prems apply auto[1]
    using rest apply simp
    using dst-point apply simp
    done

   then show ?case
    using Cons.IH Cons.prems(2) by simp
  qed
qed (auto split: option.splits prod.splits if-splits simp add: lookup-present-eq-key)



lemma lookup-eq-map-of-ap:
  shows lookup (λx. x=k) subs = map-of subs k
  by (induction subs arbitrary: k) auto


lemma raw-match'-assoc-match:
  shows raw-match' T U (map-of subs) = map-option map-of (assoc-match T U
subs)
  proof (induction T U map-of subs arbitrary: subs rule: raw-match'.induct)
case (1 v S T)
  then show ?case
   by (auto split: option.splits prod.splits simp add: lookup-present-eq-key lookup-eq-map-of-ap)
next
  case (2 a Ts b Us subs)
  then show ?case
  proof(cases (raw-match' (Ty a Ts) (Ty b Us) (map-of subs)))
    case None
    then show ?thesis
    proof (cases a = b ∧ length Ts = length Us)
      case True
      hence length Ts = length Us a = b by auto
      then show ?thesis using 2 None
      proof (induction Ts Us arbitrary: subs rule: list-induct2)
        case Nil
        then show ?case by simp
      next
        case (Cons x xs y ys)

     hence eq-hd: raw-match' x y (map-of subs) = map-option map-of (assoc-match
x y subs)
         by auto
```

```
        then show ?case
        proof(cases assoc-match x y subs)
          case None
          hence raw-match' x y (map-of subs) = None using eq-hd by simp
          then show ?thesis
        using fold-Option-bind-at-some-point-None-eq-None fold-assoc-matches-first-step-not-None
            Cons.prems
          by (auto split: option.splits prod.splits if-splits
              simp add: fold-Option-bind-eq-None-start-None)
        next
          case (Some res)
         hence raw-match' x y (map-of subs) = Some (map-of res) using eq-hd by
simp
          then show ?thesis
        using fold-assoc-matches-first-step-not-None fold-Option-bind-eq-Some-at-each-point-Some
            Cons.prems Cons.IH
          by (auto split: option.splits prod.splits if-splits
              simp add: fold-Option-bind-eq-None-start-None)
        qed
      qed
    next
      case False
      then show ?thesis using None 2 by auto
    qed
  next
    case (Some res)
    hence l: length Ts = length Us a = b by (simp-all split: if-splits)
    have better-IH: (x, y) ∈ set (zip Ts Us) ⟹
    raw-match' x y (map-of subs) = map-option map-of (assoc-match x y subs)
      for x y subs using 2 Some by (simp split: if-splits)
    from l better-IH Some 2.prems show ?thesis
    proof (induction Ts Us arbitrary: subs res rule: list-induct2)
      case Nil
      then show ?case by simp
    next
      case (Cons x xs y ys)

       obtain point where first: raw-match' x y (map-of subs) = Some (map-of
point)
        and rest: raw-match' (Ty a xs) (Ty b ys) (map-of point) = Some res
        using fold-matches-first-step-not-None Cons.prems
        by (simp split: option.splits prod.splits if-splits) (smt map-option-eq-Some)

      have 1: raw-match' x y (map-of subs) = map-option map-of (assoc-match x
y subs)
        using Cons.prems by simp

      have 2: raw-match' (Ty a xs) (Ty b ys) (map-of point)
        = map-option map-of (assoc-match (Ty a xs) (Ty b ys) point)
```

using *Cons rest* **by** *auto*

show *?case*
using *1 2 first rest*
apply (*simp split*: *if-splits option.splits prod.splits*)
by (*smt Cons.IH Cons.prems*(*2*) *assoc-match.simps*(*2*) *list.set-intros*(*2*) *map-option-eq-Some rest zip-Cons-Cons*)
qed
qed
qed (*auto split*: *option.splits prod.splits simp add*: *lookup-present-eq-key*)

lemma *dom-eq-and-eq-on-dom-imp-eq*: *dom m* = *dom m'* ⟹ ∀ *x*∈*dom m* . *m x* = *m' x* ⟹ *m* = *m'*
by (*simp add*: *map-le-def map-le-same-dom-imp-same-map*)

lemma *list-of-map*:
assumes *finite* (*dom subs*)
shows ∃ *l*. *map-of l* = *subs*
proof−
have *finite* {(*k*, *the* (*subs k*)) | *k* . *k*∈*dom subs*} **using** *assms* **by** *simp*
from *this* **obtain** *l* **where** *l*: *set l* = {(*k*, *the* (*subs k*)) | *k* . *k*∈*dom subs*}
using *finite-list* **by** *fastforce*

hence *dom* (*map-of l*) = *fst* ‘ {(*k*, *the* (*subs k*)) | *k* . *k*∈*dom subs*}
by (*simp add*: *dom-map-of-conv-image-fst*)
also have ... = *dom subs*
by (*simp add*: *Setcompr-eq-image domI image-image*)
finally have *dom* (*map-of l*) = *dom subs* **.**
moreover have *map-of l x* = *subs x* **if** *x*∈*dom subs* **for** *x*
using *that*
by (*smt l domIff fst-conv map-of-SomeD mem-Collect-eq option.collapse prod.sel*(*2*) *weak-map-of-SomeI*)
ultimately have *map-of l* = *subs*
by (*simp add*: *dom-eq-and-eq-on-dom-imp-eq*)
thus *?thesis* **..**
qed

corollary *tinstT-iff-assoc-match*[*code*]: *tinstT T1 T2* ⟷ *assoc-match T2 T1* [] ~= *None*
using *tinstT-iff-ex-raw-match' list-of-map raw-match'-assoc-match*
by (*smt map-of-eq-empty-iff map-option-is-None option.collapse option.distinct*(*1*))

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*
by *pat-completeness auto*
termination by *size-change*

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 (*induction t u map-of subs arbitrary*: *subs rule*: *raw-match-term.induct*)
case (*4 T t U u*)

then show *?case*
proof (*cases assoc-match T U subs*)
case *None*
then show *?thesis* **using** *raw-match′-assoc-match* **by** *simp*
next
case (*Some bsubs*)
hence *1*: *raw-match′ T U* (*map-of subs*) = *Some* (*map-of bsubs*)
using *raw-match′-assoc-match* **by** *simp*
hence *raw-match-term t u* (*map-of bsubs*) = *map-option map-of* (*assoc-match-term t u bsubs*)
using *4* **by** *blast*
then show *?thesis* **by** (*simp add*: *Some 1*)
qed
next
case (*5 f u f′ u′*)

from *5.hyps*(*1*) *5.hyps*(*2*) **have** *Option.bind* (*map-option map-of* (*assoc-match-term f f′ subs*))
(*raw-match-term u u′*) =
map-option map-of (*Option.bind* (*assoc-match-term f f′ subs*) (*assoc-match-term u u′*))
by (*smt None-eq-map-option-iff bind.bind-lunit bind-eq-None-conv option.collapse option.map-sel*)
with *5* **show** *?case*
using *raw-match′-assoc-match 5*
by (*auto split*: *option.splits prod.splits simp add*: *lookup-present-eq-key bind-eq-Some-conv bind-eq-None-conv*)
qed (*use raw-match′-assoc-match* **in** ‹*auto split*: *option.splits prod.splits*›)

corollary *tinst-iff-assoc-match-term*[*code*]: *tinst t1 t2* ⟷ *assoc-match-term t2 t1* [] ≠ *None*

proof
assume *tinst t1 t2*
from *this* **obtain** *asubs* **where** *raw-match-term t2 t1 Map.empty = Some asubs*
using *tinst-imp-ex-raw-match-term* **by** *blast*
from *this* **obtain** *csubs* **where** *assoc-match-term t2 t1 [] = Some csubs*
by (*metis empty-eq-map-of-iff map-option-eq-Some raw-match-term-assoc-match-term*)
thus *assoc-match-term t2 t1 [] ≠ None* **by** *simp*
next
assume *assoc-match-term t2 t1 [] ≠ None*
from *this* **obtain** *csubs* **where** *assoc-match-term t2 t1 [] = Some csubs*
by *blast*
from *this* **obtain** *asubs* **where** *raw-match-term t2 t1 Map.empty = Some asubs*
by (*metis empty-eq-map-of-iff option.simps*(*9*) *raw-match-term-assoc-match-term*)
thus *tinst t1 t2*
using *tinst-iff-ex-raw-match-term* **by** *blast*
qed

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*)
by (*simp add*: *ran-distinct*)

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 (λk. k=type) (exetcsigs sa))) .
      the (lookup (λk. 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–
  {
    fix type y
    assume p: exe-osig-conds sa trans (fst (translate-osig sa)) snd (translate-osig
sa) type = Some y
      hence exe-ars-conds (exetcsigs sa)
        using exe-osig-conds-def by blast
    from p have translate-ars (exetcsigs sa) type = Some y
      by (metis snd-conv translate-osig.elims)
    hence (type, y) ∈ set (map (apsnd map-of) (exetcsigs sa))
      using map-of-SomeD by force
    hence type ∈ fst ‘ set (exetcsigs sa) by force
    from this obtain x where lookup (λx. x=type) (exetcsigs sa) = Some x
      using key-present-imp-eq-lookup-finds-value by metis
    hence map-of x = y
      by (metis ‹exe-ars-conds (snd sa)› ‹translate-ars (snd sa) type = Some y›
          exe-ars-conds-def in-alist-imp-in-map-of lookup-eq-map-of-ap
          map-of-SomeD option.sel)
    have ∃ y. (type, y) ∈ set tao
      using ‹type ∈ fst ‘ set (exetcsigs sa)› assms(4) by auto
  }
  note 1 = this

  {
    fix ars type y
    assume p: exe-osig-conds sa
      trans (fst (translate-osig sa))
      ∀ x∈set cto. typ-ok-sig (map-of cto, map-of tao, translate-osig sa) (snd x)
      ars ∈ ran y
      snd (translate-osig sa) type = Some y

      hence exe-ars-conds (exetcsigs sa)
        using exe-osig-conds-def by blast
      from p(1–2) p(5) have translate-ars (exetcsigs sa) type = Some y
        by (metis snd-conv translate-osig.elims)
      hence (type, y) ∈ set (map (apsnd map-of) (exetcsigs sa))
        using map-of-SomeD by force
      hence dom: type ∈ fst ‘ set (exetcsigs sa) by force
      from this obtain x where x: lookup (λx. x=type) (exetcsigs sa) = Some x
        using key-present-imp-eq-lookup-finds-value by metis
      hence map-of x = y
        by (metis ‹exe-ars-conds (snd sa)› ‹translate-ars (snd sa) type = Some y›
          exe-ars-conds-def in-alist-imp-in-map-of lookup-eq-map-of-ap map-of-SomeD
option.sel)
```

```
    have ars ∈ snd ‘ set x
      by (metis ‹map-of x = y› image-iff in-range-if-ex-key map-of-SomeD p(4)
snd-conv)

    have type ∈ fst ‘ set tao
      apply (simp add: ‹type ∈ fst ‘ set (exetcsigs sa)› assms(4))
      using assms(4) dom by blast
     moreover have 1: (∀ ars ∈ snd ‘ set (the (lookup (λk. k=type) (exetcsigs
sa))) .
      the (lookup (λk. k=type) tao) = length ars)
      using ‹type ∈ fst ‘ set (exetcsigs sa)› assms(4) by blast

    ultimately have the (lookup (λk. k = type) tao) = length ars
      using ‹lookup (λx. x = type) (exetcsigs sa) = Some x› ‹map-of x = y›
        in-range-if-ex-key map-of-SomeD option.sel p(3) snd-conv
      by (simp add: 1 ‹ars ∈ snd ‘ set x›)
    hence the (map-of tao type) = length ars
    by (metis ‹the (lookup (λk. k = type) tao) = length ars› lookup-eq-map-of-ap)
  }
  note 2 = this
  {
   fix a b x y
   assume p: fst ‘ set b = fst ‘ set tao
     (x, y) ∈ set tao
     sa = (a, b)

   have x ∈ fst ‘ set b
     by (metis fst-conv image-iff p(1) p(2))
   from this obtain ars where lookup (λk. k=x) b = Some ars
     by (metis key-present-imp-eq-lookup-finds-value)
   hence (x,ars) ∈ set b
     by (simp add: lookup-present-eq-key′)
   hence lookup (λk. k=x) (map (apsnd map-of) b) = Some (map-of ars)
    by (metis assms(3) exe-ars-conds-def exe-osig-conds-def in-alist-imp-in-map-of
        lookup-eq-map-of-ap p(3) snd-conv translate-ars.simps)
   hence ∃ y. map-of (map (apsnd map-of) b) x = Some y
     by (metis lookup-eq-map-of-ap)
  }
  note 3 = this
  {
   fix a b x
   assume p: alist-conds cto
     x ∈ ran (map-of cto)
     sa = (a, b)
   have typ-ok-sig (map-of cto, map-of tao, set a,  map-of (map (apsnd map-of)
b)) x
     using assms(4) p(1) p(2) p(3) ran-distinct by fastforce
  }
  note 4 = this
```

```
  have wf-osig (translate-osig sa)
    using assms(4) wf-osig-iff-exe-wf-osig by simp
  thus ?thesis apply (cases sa)
    using 1 2 3 4 assms by auto
qed

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
    ∧ 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 . typ-ok-sig (map-of cto, map-of tao, translate-osig sa)
ty)
proof−
  {
    fix a b x y
    assume p: alist-conds tao
      exe-ars-conds b
      dom (map-of (map (apsnd map-of) b)) = dom (map-of tao)
      (x, y) ∈ set b

    hence x ∈ fst ‘ set tao
      by (metis domIff dom-map-of-conv-image-fst exe-ars-conds-def
          in-alist-imp-in-map-of option.distinct(1) translate-ars.simps)
  }
  note 1 = this
  {
    fix cl n ar and tcs :: (String.literal × (String.literal × String.literal set list)
list) list
    assume p: dom (map-of (map (apsnd map-of) tcs)) = dom (map-of tao)
      alist-conds tao
      (n, ar) ∈ set tao

    obtain mgd where translate-ars tcs n = Some mgd
      using p by (metis Some-eq-map-of-iff domI domIff option.exhaust-sel trans-
late-ars.simps)
    hence map-of (map (apsnd map-of) tcs) n = Some mgd
      by (simp add: tcsigs-translate exe-osig-conds-def p)
    hence n ∈ fst ‘ set (map (apsnd map-of) tcs)
      by (meson domI domIff map-of-eq-None-iff)
    then have n ∈ fst ‘ set tcs
      by force
  }
  note 2 = this
  {
    fix cl tcs n K c Ss
```

```
    assume p: (n, K) ∈ set tcs
      (c, Ss) ∈ set (the (lookup (λk. k = n) tcs))
      exe-ars-conds tcs
      dom (map-of (map (apsnd map-of) tcs)) = dom (map-of tao)
      ∀ type∈dom (map-of tao). ∀ ars∈ran (the (map-of (map (apsnd map-of) tcs)
type)).
          the (map-of tao type) = length ars

    have 1: translate-ars tcs n = Some (map-of K)
      using exe-ars-conds-def in-alist-imp-in-map-of p(1−3) by blast
    have 2: map-of K c = Some Ss
      using p(1−3)
      by (metis Some-eq-map-of-iff exe-ars-conds-def image-iff lookup-eq-map-of-ap
          option.sel snd-conv)
    have the (lookup (λk. k = n) tao) = length Ss
      using 1 2 p(4,5)
     by (metis domIff lookup-eq-map-of-ap option.distinct(1) option.sel ranI trans-
late-ars.simps)
  }
  note 3 = this

  have 1: wf-osig (translate-osig sa) dom (tcsigs (translate-osig sa)) = dom (map-of
tao)
    (∀ type ∈ dom (tcsigs (translate-osig sa)).
    (∀ ars ∈ ran (the (tcsigs (translate-osig sa) type)) . the ((map-of tao) type) =
length ars))
    (∀ ty ∈ Map.ran (map-of cto) . wf-type (map-of cto, map-of tao, translate-osig
sa) ty)
    using assms(4) by auto
  note pre = 1

  have exe-wf-osig sa
    using 1(1) wf-osig-iff-exe-wf-osig by blast
  moreover have fst ‘ set (snd sa) = fst ‘ set tao
  proof
    show fst ‘ set (snd sa) ⊆ fst ‘ set tao
      using assms(3−4)
    by (clarsimp simp add: dom-map-of-conv-image-fst exe-ars-conds-def exe-osig-conds-def)
     (metis tcsigs-translate assms(3) domIff in-alist-imp-in-map-of option.simps(3))
  next
    show fst ‘ set (snd sa) ⊇ fst ‘ set tao
      using 1(2) 2 assms(2−3) tcsigs-translate by auto
  qed
  moreover have (∀ type∈fst ‘ set (snd sa). ∀ ars∈snd ‘ set (the (lookup (λk. k =
type) (snd sa))).
       the (lookup (λk. k = type) tao) = length ars)
  proof (standard+, goal-cases)
    case (1 n Ss)
    obtain c where c: (c, Ss) ∈ set (the (lookup (λk. k = n) (snd sa)))
```

using *1(2)* **by** *force*
have *dom* (*map-of* (*map* (*apsnd map-of*) (*snd sa*))) = *dom* (*map-of tao*)
using *assms(3) pre(2) tcsigs-translate* **by** *fastforce*
show *?case*
using *assms(3) pre(2) c tcsigs-translate pre(2−3) domI*
by (*fastforce simp add*: *exe-osig-conds-def tcsigs-translate*[*OF assms(3)*]
1(1) key-present-imp-eq-lookup-finds-value lookup-present-eq-key′
split: *option.splits intro*!: *3*[*of - the* (*lookup* (λ*k*. *k* = *n*) (*snd sa*)) *snd sa c*])+
qed
moreover have (∀ *ty* ∈ *Map.ran* (*map-of cto*) . *wf-type* (*map-of cto*, *map-of tao*, *translate-osig sa*) *ty*)
using *1(4)* **by** *blast*
ultimately show *?thesis*
by (*simp add*: *assms(1) ran-distinct*)
qed

lemma *wf-sig-iff-exe-wf-sig-pre*: *alist-conds cto* ⟹ *alist-conds tao* ⟹ *exe-osig-conds sa*
⟹ *wf-sig* (*map-of cto*, *map-of tao*, *translate-osig 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*))
using *exe-wf-sig-imp-wf-sig wf-sig-imp-exe-wf-sig* **by** *meson*

lemma *wf-sig-iff-exe-wf-sig*: *alist-conds cto* ⟹ *alist-conds tao* ⟹ *exe-osig-conds sa*
⟹ *wf-sig* (*map-of cto*, *map-of tao*, *translate-osig sa*)
⟷ *exe-wf-sig* (*ExeSignature cto tao sa*)
unfolding *exe-wf-sig.simps*
using *wf-sig-iff-exe-wf-sig-pre* **by** *presburger*

fun *translate-signature* :: *exesignature* ⇒ *signature* **where**
translate-signature (*ExeSignature cto tao sa*)
= (*map-of cto*, *map-of tao*, *translate-osig sa*)

fun *exetyp-ok-sig* :: *exesignature* ⇒ *typ* ⇒ *bool* **where**
exetyp-ok-sig Σ (*Ty c Ts*) = (*case lookup* (λ*k*. *k*=*c*) (*exetyp-arity-of* Σ) *of*
None ⇒ *False*
| *Some ar* ⇒ *length Ts* = *ar* ∧ *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* ≡ *exesort-ex* (*execlasses* (*exesorts* Σ)) *S*
∧ *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)
by (*cases* Σ) (*simp add*: *lookup-eq-map-of-ap*)

lemma *typ-ok-sig-code*:
assumes *exe-osig-conds* (*exesorts* Σ)
shows *typ-ok-sig* (*translate-signature* Σ) *ty* = *exetyp-ok-sig* Σ *ty*
using *assms* **apply** (*induction ty*) **apply** *simp*
apply (*auto split*: *option.splits simp add*: *wf-sort-def list-all-iff typ-arity-lookup-code*)[]
using *wf-sort-code* **by** (*cases* Σ) (*simp add*: *exe-osig-conds-def classes-translate*)

fun *exe-wf-sig'* **where**
exe-wf-sig' (*ExeSignature cto tao sa*) = (*exe-wf-osig sa* $\wedge$
fst ' *set* (*exetcsigs sa*) = *fst* ' *set tao*
$\wedge$ ($\forall$ *type* $\in$ *fst* ' *set* (*exetcsigs sa*).
($\forall$ *ars* $\in$ *snd* ' *set* (*the* (*lookup* ($\lambda k.$ *k*=*type*) (*exetcsigs sa*))) .
the (*lookup* ($\lambda k.$ *k*=*type*) *tao*) = *length ars*))
$\wedge$ ($\forall$ *ty* $\in$ *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
using *typ-ok-sig-code* **by** (*cases* Σ, *simp*, *metis exesignature.sel*(*3*) *translate-signature.simps*)

fun *exeterm-ok'* :: *exesignature* $\Rightarrow$ *term* $\Rightarrow$ *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* ($\lambda k.$ *k*=*s*) (*execonst-type-of* Σ) *of*
None $\Rightarrow$ *False*
| *Some ty* $\Rightarrow$ *exetyp-ok-sig* Σ *T* $\wedge$ *tinstT T ty*)
| *exeterm-ok'* Σ (*t* $ *u*) $\longleftrightarrow$ *exeterm-ok'* Σ *t* $\wedge$ *exeterm-ok'* Σ *u*
| *exeterm-ok'* Σ (*Abs T t*) $\longleftrightarrow$ *exetyp-ok-sig* Σ *T* $\wedge$ *exeterm-ok'* Σ *t*

lemma *const-type-of-lookup-code*: *const-type* (*translate-signature* Σ) *n* = *lookup* ($\lambda k.$ *k* = *n*) (*execonst-type-of* Σ)
by (*cases* Σ) (*simp add*: *lookup-eq-map-of-ap*)

lemma *wt-term-code*:
assumes *exe-osig-conds* (*exesorts* Σ)
shows *term-ok'* (*translate-signature* Σ) *t* = *exeterm-ok'* Σ *t*
by (*induction t*) (*auto simp add*: *const-type-of-lookup-code assms typ-ok-sig-code split*: *option.splits*)

datatype *exetheory* = *ExeTheory* (*exesig*: *exesignature*) (*exeaxioms-of*: *term list*)

lemma *exetheory-full-exhaust*: ($\bigwedge$*const-type typ-arity sorts axioms*.
Θ = (*ExeTheory* (*ExeSignature const-type typ-arity sorts*) *axioms*) $\Longrightarrow$ *P*)
$\Longrightarrow$ *P*
apply (*cases* Θ) **subgoal for** Σ *axioms* **apply** (*cases* Σ) **by** *auto* **done**

definition *exe-sig-conds* $\Sigma \equiv$ *alist-conds* (*execonst-type-of* Σ) $\wedge$ *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
  by simp

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
   by (metis exe-sig-conds-def exesignature.exhaust-sel wf-sig-iff-exe-wf-sig trans-
late-signature.simps)

lemma wf-sig-imp-exe-wf-sig': exe-sig-conds Σ ⟹
    wf-sig (translate-signature Σ) ⟹
    exe-wf-sig Σ
   by (metis exe-sig-conds-def exesignature.exhaust-sel wf-sig-iff-exe-wf-sig trans-
late-signature.simps)

lemma exe-wf-sig-imp-wf-sig': exe-sig-conds Σ ⟹
    exe-wf-sig Σ
    ⟹ wf-sig (translate-signature Σ)
   by (metis exe-sig-conds-def exesignature.exhaust-sel wf-sig-iff-exe-wf-sig trans-
late-signature.simps)

lemma wf-theory-translate-imp-exe-wf-theory:
  assumes wf-theory (translate-theory a) shows exe-wf-theory a
proof–
  have exe-sig-conds (exesig a) using assms
   by (metis exetheory.collapse illformed-theory-not-wf-theory translate-theory.simps)
  moreover have wf-sig (translate-signature (exesig a))
    ⟷ exe-wf-sig (exesig a)
    by (simp add: calculation(1) wf-sig-iff-exe-wf-sig')
  ultimately show ?thesis using assms
    by (cases a rule: exe-wf-theory.cases) (fastforce simp add: image-iff eq-fst-iff)
qed
```

lemma *exe-wf-theory-translate-imp-wf-theory*:
assumes *exe-wf-theory a* **shows** *wf-theory* (*translate-theory a*)
proof−
have *exe-sig-conds* (*exesig a*) **using** *assms*
by (*metis* (*full-types*) *exe-wf-theory.simps exesignature.exhaust-sel exetheory.sel*(*1*) *translate-theory.cases*)
moreover hence
(∀ *ty* ∈ *Map.ran* (*map-of* (*execonst-type-of* (*exesig a*))) . *typ-ok-sig* (*translate-signature* (*exesig a*)) *ty*)
⟷ (∀ *ty* ∈ *snd* ‘ *set* (*execonst-type-of* (*exesig a*)) . *typ-ok-sig* (*translate-signature* (*exesig a*)) *ty*)
by (*simp add*: *exe-sig-conds-def ran-distinct*)
moreover have *wf-sig* (*translate-signature* (*exesig a*))
⟷ *exe-wf-sig* (*exesig a*)
by (*simp add*: *calculation*(*1*) *wf-sig-iff-exe-wf-sig*′)
ultimately show *?thesis*
using *assms* **by** (*cases a rule*: *exe-wf-theory.cases*) *auto*
qed

lemma *wf-theory-translate-iff-exe-wf-theory*:
wf-theory (*translate-theory a*) ⟷ *exe-wf-theory a*
using *exe-wf-theory-translate-imp-wf-theory wf-theory-translate-imp-exe-wf-theory*
by *blast*

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
by (*cases* Σ) (*auto simp add*: *lookup-eq-map-of-ap*)

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*)
using *wt-term-code*[*OF assms*] **by** *force*

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*)
using *assms* **by** (*auto simp add*: *wt-term-def split*: *if-splits*)

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*)
using *assms wf-term-imp-term-ok' exe-sig-conds-def wt-term-code*
by (*fastforce simp add*: *term-ok-translate-code-step wt-term-code wt-term-def*)

lemma *exe-wf-theory-code*[*code*]: *exe-wf-theory* Θ = *exe-wf-theory'* Θ
apply (*cases* Θ *rule*: *exetheory-full-exhaust*)
apply (*simp only*: *exe-wf-theory.simps exe-wf-theory'.simps*)
using *term-ok-theory-cond-code is-std-sig-code* **by** *meson*

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*
using *assms typ-ok-sig-code*
by (*metis exe-sig-conds-def exe-wf-theory.simps exe-wf-theory-code exesignature.exhaust exetheory.sel*(*1*) *sig.simps translate-theory.elims typ-ok-def wf-type-iff-typ-ok-sig*)

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*
by (*simp add*: *class-leq-def class-les-def member-def*)

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*
by (*simp add*: *execlass-leq-code exesort-leq-def sort-leq-def*)

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 (*induction T arbitrary*: *S*)
case (*Ty n Ts*)
obtain *sub' tcs'* **where** *sub'-tcs'*: *translate-osig* (*sub*, *tcs*) = (*sub'*, *tcs'*) **by** *fastforce*
obtain *mgd* **where** *mgd*: *tcs' n* = *Some mgd*
using *Ty.prems sub'-tcs'* **apply** (*simp split*: *option.splits*)
by (*metis assms exe-ars-conds-def exe-osig-conds-def in-alist-imp-in-map-of lookup-eq-map-of-ap*
map-of-SomeD snd-conv)
show *?case*
proof (*subst sub'-tcs'*, *rule has-sort-Ty*[*of tcs'*, *OF mgd*], *rule ballI*)
fix *c* **assume** *asm*: *c*∈*S*

have *l*: *lookup* (λ*k*. *k*=*n*) (*map* (*apsnd map-of*) *tcs*) = *Some mgd*
by (*metis assms lookup-eq-map-of-ap mgd snd-conv sub'-tcs' translate-ars.simps translate-osig.simps*)
hence ∃ *x*. (*lookup* (λ*k*. *k*=*n*) *tcs*) = *Some x*
by (*induction tcs*) *auto*
from *this* **obtain** *pre-mgd* **where** *pre-mgd*: (*lookup* (λ*k*. *k*=*n*) *tcs*) = *Some pre-mgd*
by *blast*
have *pre-mgd-mgd*: *map-of pre-mgd* = *mgd*
by (*metis l assms exe-ars-conds-def*
exe-osig-conds-def in-alist-imp-in-map-of lookup-eq-map-of-ap map-of-SomeD

option.sel pre-mgd snd-conv translate-ars.simps)

obtain *Ss* **where** *Ss*: *lookup* (λ*k*. *k*=*c*) *pre-mgd* = *Some Ss*
using *Ty.prems asm* **by** (*auto simp add*: *pre-mgd split*: *option.splits*)
hence *cond*: *list-all2* (*exehas-sort* (*sub*,*tcs*)) *Ts Ss*
using ‹*exehas-sort* (*sub*, *tcs*) (*Ty n Ts*) *S*›*asm pre-mgd* **by** (*auto split*:

option.splits)

from *Ss* **have** *mgd c = Some Ss*
by (*simp add*: *lookup-eq-map-of-ap pre-mgd-mgd*)
then show ∃ *Ss*. *mgd c = Some Ss* ∧ *list-all2* (*has-sort* (*sub′*, *tcs′*)) *Ts Ss*
using *cond Ty.IH list.rel-mono-strong sub′-tcs′* **by** *force*
qed
next
case (*Tv n S*)
then show *?case*
by (*metis assms exehas-sort.simps*(*1*) *exesort-les-code has-sort-Tv prod.collapse translate-osig.simps*)
qed

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 (*induction T arbitrary*: *S*)
case (*Ty n Ts*)
obtain *sub′ tcs′* **where** *sub′-tcs′*: *translate-osig* (*sub*, *tcs*) = (*sub′*, *tcs′*) **by** *fastforce*
obtain *mgd* **where** *mgd*: *tcs′ n = Some mgd*
using *Ty.prems sub′-tcs′ has-sort.simps* **by** (*auto split*: *option.splits*)
hence *lookup* (λ*k*. *k*=*n*) (*map* (*apsnd map-of*) *tcs*) = *Some mgd*
by (*metis assms lookup-eq-map-of-ap prod.inject sub′-tcs′ translate-ars.simps translate-osig.simps*)
have *l*: *lookup* (λ*k*. *k*=*n*) (*map* (*apsnd map-of*) *tcs*) = *Some mgd*
by (*metis assms lookup-eq-map-of-ap mgd snd-conv sub′-tcs′*
translate-ars.simps translate-osig.simps)
hence ∃ *x*. (*lookup* (λ*k*. *k*=*n*) *tcs*) = *Some x*
by (*induction tcs*) *auto*
from *this* **obtain** *pre-mgd* **where** *pre-mgd*: (*lookup* (λ*k*. *k*=*n*) *tcs*) = *Some pre-mgd*
by *blast*
have *pre-mgd-mgd*: *map-of pre-mgd = mgd*
by (*metis l assms exe-ars-conds-def*
exe-osig-conds-def in-alist-imp-in-map-of lookup-eq-map-of-ap map-of-SomeD option.sel
pre-mgd snd-conv translate-ars.simps)

{
fix *c* **assume** *asm*: *c*∈*S*

obtain *Ss* **where** *Ss*: *lookup* (λ*k*. *k*=*c*) *pre-mgd = Some Ss*
using ‹*c* ∈ *S*› ‹*map-of pre-mgd = mgd*› *sub′-tcs′ mgd assms Ty.prems has-sort.simps*
by (*auto simp add*: *dom-map-of-conv-image-fst domIff eq-fst-iff exe-ars-conds-def*

map-of-eq-None-iff classes-translate lookup-eq-map-of-ap split: *typ.splits*

```
        dest!: domD intro!: domI)
    have l: length Ts = length Ss
    using asm mgd pre-mgd Ty.prems assms sub'-tcs' Ss list-all2-lengthD pre-mgd-mgd
      by (fastforce simp add: has-sort.simps lookup-eq-map-of-ap)

    have 1: ∀ c ∈ S. ∃ Ss . mgd c = Some Ss ∧ list-all2 (has-sort (sub', tcs')) Ts Ss
      using mgd Ty.prems has-sort.simps sub'-tcs' by auto

    have cond: list-all2 (exehas-sort (sub,tcs)) Ts Ss
      apply (rule list-all2-all-nthI)
      using l apply simp
      subgoal premises p for m
        apply (rule Ty.IH)
        using p apply simp
        using p Ty.prems assms 1
          by (metis Ss asm list-all2-conv-all-nth lookup-eq-map-of-ap option.sel
pre-mgd-mgd sub'-tcs')
      done
    have (∀ C ∈ S.
    case lookup (λk. k=C) pre-mgd of
        None ⇒ False
      | Some Ss ⇒ list-all2 (exehas-sort (sub,tcs)) Ts Ss)
        by (metis 1 Ty.IH list-all2-conv-all-nth lookup-eq-map-of-ap nth-mem op-
tion.simps(5)
        pre-mgd-mgd sub'-tcs')
  }

  then show ?case
    using pre-mgd by simp
next
  case (Tv n S)
  then show ?case
    using assms exesort-les-code has-sort-Tv-imp-sort-leq by fastforce
qed

lemma has-sort-code:
  assumes exe-osig-conds oss
  shows has-sort (translate-osig oss) T S = exehas-sort oss T S
  by (metis assms exehas-sort-imp-has-sort has-sort-imp-exehas-sort prod.collapse)

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
  apply (cases Θ rule: exetheory-full-exhaust) using assms has-sort-code by auto

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*)
using *assms typ-ok-code* **by** (*auto simp add*: *list-all-iff*)

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*
using *has-sort-code' assms* **by** *auto*

lemma *inst-ok-code*:
assumes *exe-wf-theory'* Θ
shows *inst-ok* (*translate-theory* Θ) *insts* = *exeinst-ok* Θ *insts*
using *inst-ok-code1 inst-ok-code2 assms* **by** *auto*

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*
using *assms* **apply** (*cases* Θ *rule*: *exetheory-full-exhaust*)
by (*metis exe-sig-conds-def exe-wf-theory'.simps exeterm-ok-def exetheory.sel*(*1*) *sig.simps term-okD1 term-okD2 term-okI wt-term-code translate-theory.simps*)

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−
  have has-sort (osig (sig (translate-theory thy))) ty {c}
    = exehas-sort (exesorts (exesig thy)) ty {c}
    using has-sort-code' assms by simp
  moreover have typ-ok (translate-theory thy) ty = exetyp-ok thy ty
    using typ-ok-code assms by simp
  ultimately show ?thesis
    by auto
qed

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))
  by (metis assms const-type-of-lookup-code exe-wf-theory-code
    exe-wf-theory-translate-imp-wf-theory exetheory.sel(1) illformed-theory-not-wf-theory
```

sig.simps translate-theory.elims)

lemma *replay′-code*:
assumes *exe-wf-theory′ thy*
shows *replay′* (*translate-theory thy*) *vs ns Hs P* = *exereplay′ thy vs ns Hs P*
proof (*induction P arbitrary*: *vs ns Hs*)
case (*PAxm ax tys*)
have *wf*: *wf-theory* (*translate-theory thy*)
by (*simp add*: *assms exe-wf-theory-code exe-wf-theory-translate-imp-wf-theory*)
moreover have *inst*: *inst-ok* (*translate-theory thy*) *tys* ⟷ *exeinst-ok thy tys*
by (*simp add*: *assms inst-ok-code1 inst-ok-code2*)
moreover have *tok*: *term-ok* (*translate-theory thy*) *ax* ⟷ *exeterm-ok thy ax*
using *assms term-ok-code* **by** *blast*
moreover have *ax*: *ax* ∈ *axioms* (*translate-theory thy*) ⟷ *ax* ∈ *set* (*exeaxioms-of thy*)
by (*metis axioms.simps wf exetheory.sel*(*2*) *illformed-theory-not-wf-theory translate-theory.elims*)
ultimately show *?case*
by *simp*
qed (*use assms term-ok-code typ-ok-code of-class-code1 of-class-code2*
in ‹*auto simp only*: *replay′.simps exereplay′.simps split*: *if-splits*›)

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*
by (*simp add*: *assms replay′-code*)

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*
using *assms replay″-code term-ok-code* **by** *auto*

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*
using *check-proof-def exe-check-proof-def wf-theory-translate-iff-exe-wf-theory*
by (*metis exe-wf-theory-code replay-code*)

lemma *check-proof-sound*:

shows *exe-check-proof e P res* $\Longrightarrow$ *translate-theory e*, *set* (*hyps P*) $\vdash$ *res*
by (*simp add*: *check-proof-sound exe-check-proof-iff-check-proof*)

lemma *check-proof-really-sound*:
shows *exe-check-proof e P res* $\Longrightarrow$ *translate-theory e*, *set* (*hyps P*) $\Vdash$ *res*
by (*simp add*: *check-proof-really-sound exe-check-proof-iff-check-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.