

Isabelle’s Metalogic: Formalization and Proof Checker

Tobias Nipkow and Simon Roßkopf

December 14, 2021

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 Bruijn indices) to enable easy integration of generated code with the Isabelle system without a complicated translation layer.

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

Contents

1	Core Inference system	2
2	Preliminaries	8
3	Terms	17
4	Sorts	49
5	Wellformed Signature and Theory	57
6	More on Substitutions	61
7	Names	81
8	Beta Normalization	85
9	Eta Normalization	99
10	Logic	105
11	Derived rules on equality and normalization	141

12 Proof Terms and proof checker	175
13 Executable Sorts	187
14 Executable Instance Relations	195
15 Executable Signature and Theory	217
16 Code Generation	233

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-tyt S T ≡ Ty STR "fun" [S, T]
notation mk-fun-tyt (infixr  $\rightarrow$  100)
```

Collect variables in a term

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

$| \text{fv } (Bv \ -) = \{\}$
 $| \text{fv } (Abs \ \textit{body}) = \text{fv } \textit{body}$
 $| \text{fv } (t \ \$ \ u) = \text{fv } t \cup \text{fv } u$
definition *[simp]*: $FV \ S = (\bigcup_{s \in S} . \text{fv } s)$

Typ/term instantiations

fun *tsubstT* :: $\text{typ} \Rightarrow (\text{variable} \Rightarrow \text{sort} \Rightarrow \text{typ}) \Rightarrow \text{typ}$ **where**
 $\textit{tsubstT } (Tv \ a \ s) \ \varrho = \varrho \ a \ s$
 $| \textit{tsubstT } (Ty \ \kappa \ \sigma s) \ \varrho = Ty \ \kappa \ (\text{map } (\lambda \sigma. \textit{tsubstT } \sigma \ \varrho) \ \sigma s)$
definition *tinstantT* $T1 \ T2 \equiv \exists \varrho. \textit{tsubstT } T2 \ \varrho = T1$

fun *tsubst* :: $\text{term} \Rightarrow (\text{variable} \Rightarrow \text{sort} \Rightarrow \text{typ}) \Rightarrow \text{term}$ **where**
 $\textit{tsubst } (Ct \ s \ T) \ \varrho = Ct \ s \ (\textit{tsubstT } T \ \varrho)$
 $| \textit{tsubst } (Fv \ v \ T) \ \varrho = Fv \ v \ (\textit{tsubstT } T \ \varrho)$
 $| \textit{tsubst } (Bv \ i) \ - = Bv \ i$
 $| \textit{tsubst } (Abs \ T \ t) \ \varrho = Abs \ (\textit{tsubstT } T \ \varrho) \ (\textit{tsubst } t \ \varrho)$
 $| \textit{tsubst } (t \ \$ \ u) \ \varrho = \textit{tsubst } t \ \varrho \ \$ \ \textit{tsubst } u \ \varrho$

Typ of a term

inductive *has-typ1* :: $\text{typ list} \Rightarrow \text{term} \Rightarrow \text{typ} \Rightarrow \text{bool}$ ($\vdash_{\tau} \ - \ - \ -$ [51, 51, 51] 51)
where
 $\textit{has-typ1} \ - \ (Ct \ - \ T) \ T$
 $| \ i < \text{length } Ts \Longrightarrow \textit{has-typ1 } Ts \ (Bv \ i) \ (\text{nth } Ts \ i)$
 $| \textit{has-typ1} \ - \ (Fv \ - \ T) \ T$
 $| \textit{has-typ1} \ (T \# \ Ts) \ t \ T' \Longrightarrow \textit{has-typ1 } Ts \ (Abs \ T \ t) \ (T \rightarrow T')$
 $| \textit{has-typ1 } Ts \ u \ U \Longrightarrow \textit{has-typ1 } Ts \ t \ (U \rightarrow T) \Longrightarrow$
 $\textit{has-typ1 } Ts \ (t \ \$ \ u) \ T$
definition *has-typ* :: $\text{term} \Rightarrow \text{typ} \Rightarrow \text{bool}$ ($\vdash_{\tau} \ - \ - \ -$ [51, 51] 51) **where** $\textit{has-typ } t \ T$
 $= \textit{has-typ1} \ [] \ t \ T$

definition *typ-of* $t = (\text{if } \exists T . \textit{has-typ } t \ T \text{ then } \text{Some } (THE \ T . \textit{has-typ } t \ T) \text{ else } \text{None})$

More operations on terms

fun *lift* :: $\text{term} \Rightarrow \text{nat} \Rightarrow \text{term}$ **where**
 $\textit{lift } (Bv \ i) \ n = (\text{if } i \geq n \text{ then } Bv \ (i+1) \text{ else } Bv \ i)$
 $| \textit{lift } (Abs \ T \ \textit{body}) \ n = Abs \ T \ (\textit{lift } \textit{body} \ (n+1))$
 $| \textit{lift } (App \ f \ t) \ n = App \ (\textit{lift } f \ n) \ (\textit{lift } t \ n)$
 $| \textit{lift } u \ n = u$

fun *subst-bv2* :: $\text{term} \Rightarrow \text{nat} \Rightarrow \text{term} \Rightarrow \text{term}$ **where**
 $\textit{subst-bv2 } (Bv \ i) \ n \ u = (\text{if } i < n \text{ then } Bv \ i$
 $\text{ else if } i = n \text{ then } u$
 $\text{ else } (Bv \ (i - 1)))$
 $| \textit{subst-bv2 } (Abs \ T \ \textit{body}) \ n \ u = Abs \ T \ (\textit{subst-bv2 } \textit{body} \ (n + 1) \ (\textit{lift } u \ 0))$
 $| \textit{subst-bv2 } (f \ \$ \ t) \ n \ u = \textit{subst-bv2 } f \ n \ u \ \$ \ \textit{subst-bv2 } t \ n \ u$
 $| \textit{subst-bv2 } t \ - \ - = t$

definition *subst-bv* $u \ t = \textit{subst-bv2 } t \ 0 \ u$

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 *tsigs* :: *osig* ⇒ (*name* → (*class* → *sort list*)) **where** *tsigs* (-, *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* = (∀ *c2* ∈ *s2* . ∃ *c1* ∈ *s1*. *class-leq* *sub* *c1 c2*)

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]: $wf\text{-subclass } rel = (trans\ rel \wedge antisym\ rel \wedge Refl\ rel)$

definition $complete\text{-tcsigs } sub\ tcs \equiv (\forall ars \in ran\ tcs . \forall (c_1, c_2) \in sub . c_1 \in dom\ ars \longrightarrow c_2 \in dom\ ars)$

definition $coregular\text{-tcsigs } sub\ tcs \equiv (\forall ars \in ran\ tcs . \forall c_1 \in dom\ ars . \forall c_2 \in dom\ ars . (class\text{-}leq\ sub\ c_1\ c_2 \longrightarrow list\text{-}all2\ (sort\text{-}leq\ sub)\ (the\ (ars\ c_1))\ (the\ (ars\ c_2))))$

definition $consistent\text{-}length\text{-}tcsigs\ tcs \equiv (\forall ars \in ran\ tcs . \forall ss_1 \in ran\ ars . \forall ss_2 \in ran\ ars . length\ ss_1 = length\ ss_2)$

definition $all\text{-}normalized\text{-}and\text{-}ex\text{-}tcsigs\ sub\ tcs \equiv (\forall ars \in ran\ tcs . \forall ss \in ran\ ars . \forall s \in set\ ss . wf\text{-}sort\ sub\ s)$

definition [simp]: $wf\text{-}tcsigs\ sub\ tcs \longleftrightarrow coregular\text{-}tcsigs\ sub\ tcs \wedge complete\text{-}tcsigs\ sub\ tcs \wedge consistent\text{-}length\text{-}tcsigs\ tcs \wedge all\text{-}normalized\text{-}and\text{-}ex\text{-}tcsigs\ sub\ tcs$

fun $wf\text{-}osig$ **where** $wf\text{-}osig\ (sub, tcs) \longleftrightarrow wf\text{-}subclass\ sub \wedge wf\text{-}tcsigs\ sub\ tcs$

Embedding typs into terms/Encoding of type classes

definition $mk\text{-}type\ ty = Ct\ STR\ "Pure.type"\ (Core.\textit{itself}T\ ty)$

abbreviation $mk\text{-}suffix\ (str::name)\ suff \equiv String.\textit{implode}\ (String.\textit{explode}\ str\ @\ String.\textit{explode}\ suff)$

abbreviation $classN \equiv STR\ "-class"$

abbreviation $const\text{-}of\text{-}class\ name \equiv mk\text{-}suffix\ name\ classN$

definition $mk\text{-}of\text{-}class\ ty\ c = Ct\ (const\text{-}of\text{-}class\ c)\ (Core.\textit{itself}T\ ty \rightarrow propT)\ \$\ mk\text{-}type\ ty$

Checking if a typ belongs to a sort

inductive $has\text{-}sort :: osig \Rightarrow typ \Rightarrow sort \Rightarrow bool$ **where**
 $has\text{-}sort\text{-}Tv[\textit{intro}]: sort\text{-}leq\ sub\ S\ S' \Longrightarrow has\text{-}sort\ (sub, tcs)\ (Tv\ a\ S)\ S'$
 $| has\text{-}sort\text{-}Ty:$
 $tcs\ \kappa = Some\ dm \Longrightarrow \forall c \in S . \exists Ss . dm\ c = Some\ Ss \wedge list\text{-}all2\ (has\text{-}sort\ (sub, tcs))\ Ts\ Ss$
 $\Longrightarrow has\text{-}sort\ (sub, tcs)\ (Ty\ \kappa\ Ts)\ S$

Signatures

type-synonym $signature = (name \rightarrow typ) \times (name \rightarrow nat) \times osig$

fun $const\text{-}type :: signature \Rightarrow (name \rightarrow typ)$ **where** $const\text{-}type\ (ctf, -, -) = ctf$

fun $type\text{-}arity :: signature \Rightarrow (name \rightarrow nat)$ **where** $type\text{-}arity\ (-, arf, -) = arf$

fun $osig :: signature \Rightarrow osig$ **where** $osig\ (-, -, oss) = oss$

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

Wellformedness checks

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

inductive *wf-type* :: *signature* \Rightarrow *typ* \Rightarrow *bool* **where**
typ-ok-Ty: *type-arity* Σ κ = Some (length Ts) \Longrightarrow $\forall T \in \text{set } Ts . \text{wf-type } \Sigma T$
 \Longrightarrow *wf-type* Σ (Ty κ Ts)
| *typ-ok-Tv*[intro]: *wf-sort* (subclass (osig Σ)) S \Longrightarrow *wf-type* Σ (Tv a S)

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

definition *wt-term* Σ t \equiv *wf-term* Σ t \wedge ($\exists T . \text{has-tyt } t T$)

fun *wf-sig* :: *signature* \Rightarrow *bool* **where**
wf-sig (ctf, arf, oss) = (*wf-osig* oss
 \wedge dom (tcsigs oss) = dom arf
 \wedge ($\forall \text{type} \in \text{dom } (tcsigs \text{oss}) . (\forall \text{ars} \in \text{ran } (the (tcsigs \text{oss } \text{type})) . the (arf \text{type})$
= length ars))
 \wedge ($\forall \text{ty} \in \text{Map.ran } \text{ctf} . \text{wf-type } (\text{ctf}, \text{arf}, \text{oss}) \text{ty}$))

Theories

type-synonym *theory* = *signature* \times *term set*

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

Equality axioms, stated directly

abbreviation *tvariable* a \equiv (Tv (Var (a, 0)) full-sort)

abbreviation *variable* x T \equiv Fv (Var (x, 0)) T

abbreviation aT \equiv *tvariable* STR "'a'"

abbreviation $bT \equiv \text{tvariable STR } "b"$
abbreviation $x \equiv \text{variable STR } "x" aT$
abbreviation $y \equiv \text{variable STR } "y" aT$
abbreviation $z \equiv \text{variable STR } "z" aT$
abbreviation $f \equiv \text{variable STR } "f" (aT \rightarrow bT)$
abbreviation $g \equiv \text{variable STR } "g" (aT \rightarrow bT)$
abbreviation $P \equiv \text{variable STR } "P" (aT \rightarrow \text{propT})$
abbreviation $Q \equiv \text{variable STR } "Q" (aT \rightarrow \text{propT})$
abbreviation $A \equiv \text{variable STR } "A" \text{propT}$
abbreviation $B \equiv \text{variable STR } "B" \text{propT}$

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

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

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

Wellformedness of theories

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

Wellformedness of typ antiations

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

Inference system

inductive $\text{proves} :: \text{theory} \Rightarrow \text{term set} \Rightarrow \text{term} \Rightarrow \text{bool } ((-, -) \vdash (-) \ 50)$ **for** Θ **where**
 $\text{axiom}: \text{wf-theory } \Theta \Longrightarrow A \in \text{axioms } \Theta \Longrightarrow \text{wf-inst } \Theta \ \varrho$
 $\Longrightarrow \Theta, \Gamma \vdash \text{tsubst } A \ \varrho$
 $| \text{assume}: \text{wf-term } (\text{sig } \Theta) \ A \Longrightarrow \text{has-ty} \ A \ \text{propT} \Longrightarrow A \in \Gamma \Longrightarrow \Theta, \Gamma \vdash A$
 $| \text{forall-intro}: \text{wf-theory } \Theta \Longrightarrow \Theta, \Gamma \vdash B \Longrightarrow (x, \tau) \notin \text{FV } \Gamma \Longrightarrow \text{wf-type } (\text{sig } \Theta) \ \tau$
 $\Longrightarrow \Theta, \Gamma \vdash \text{mk-all } x \ \tau \ B$
 $| \text{forall-elim}: \Theta, \Gamma \vdash \text{Ct STR } "Pure.all" ((\tau \rightarrow \text{propT}) \rightarrow \text{propT}) \ \$ \ \text{Abs } \tau \ B$

```

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

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

```

Ensure no garbage in Θ, Γ

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

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 $\equiv \text{case } e1 \ \text{of } \text{None} \Rightarrow e2 \mid p \Rightarrow p$

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

by(*auto split: option.split*)

lemma *Or-None*: $(e1 \text{ OR } e2) = \text{None} \longleftrightarrow e1 = \text{None} \wedge e2 = \text{None}$
by(*auto split: option.split*)

fun *lift2-option* :: $('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow 'a \text{ option} \Rightarrow 'b \text{ option} \Rightarrow 'c \text{ option}$ **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*: $\text{lift2-option } f \ x \ y \neq \text{None} \longleftrightarrow (x \neq \text{None} \wedge y \neq \text{None})$

using *lift2-option.elims* **by** *blast*

lemma *lift2-option-None*: $\text{lift2-option } f \ x \ y = \text{None} \longleftrightarrow (x = \text{None} \vee y = \text{None})$
using *lift2-option.elims* **by** *blast*

Lookup functions for assoc lists

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

lemma *findD*:

find *f* *xs* = *Some* *p* $\implies \exists x \in \text{set } xs. f \ x = \text{Some } p$
 by(*induction* *xs* *arbitrary: p*) (*auto split: option.splits*)

lemma *find-None*:

find *f* *xs* = *None* $\longleftrightarrow (\forall x \in \text{set } xs. f \ x = \text{None})$
 by(*induction* *xs*) (*auto split: option.splits*)

lemma *find-ListFind*: $\text{find } f \ l = \text{Option.bind } (\text{List.find } (\lambda x. \text{case } f \ x \ \text{of } \text{None} \Rightarrow \text{False} \mid - \Rightarrow \text{True}) \ l) \ f$
by (*induction* *l*) (*auto split: option.split*)

lemma *List.find P l = Some p $\implies \exists p \in \text{set } l . P \ p$*
by (*induction* *l*) (*auto split: if-splits*)

lemma *find-the-pair*:

assumes *distinct* (*map fst pairs*)
 and $\bigwedge x \ y. x \in \text{set } (\text{map fst pairs}) \implies y \in \text{set } (\text{map fst pairs}) \implies P \ x \implies P \ y \implies x = y$
 and $(x,y) \in \text{set pairs}$ **and** $P \ x$
 shows $\text{List.find } (\lambda(x,-) . P \ x) \ \text{pairs} = \text{Some } (x,y)$
 using *assms*(1-3)
proof (*induction* *pairs*)
 case *Nil*
 then show *?case* **by** *simp*
next
 case (*Cons pair pairs*)
 thm *Cons.prem*

```

show ?case
proof(cases fst pair = x)
  case True
  then show ?thesis
    using eq-key-imp-eq-value[OF Cons.prem1] assms(4) by force
next
  case False
  hence (x,y) ∈ set pairs
    using Cons.prem2 by fastforce
  moreover have  $\bigwedge x y. x \in \text{set } (\text{map } \text{fst } \text{pairs}) \implies y \in \text{set } (\text{map } \text{fst } \text{pairs}) \implies$ 
   $P x \implies P y \implies x = y$ 
    using Cons.prem3 by (metis list.set-intros(2) list.simps(9))
  ultimately have I: List.find ( $\lambda(x,-) . P x$ ) pairs = Some (x,y)
    using Cons.prem1,3 by (auto intro!: Cons.IH)
  moreover have  $\bigwedge y. y \in \text{set } (\text{map } \text{fst } (\text{pair } \# \text{pairs})) \implies P y \implies x = y$ 
    using Cons.prem2,3 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  $\exists x' \in \text{set } xs . \text{cmp } x x'$  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) ←  $\neg(\exists x' \in \text{set } xs . \text{cmp } x x') \wedge \text{distinct-on cmp } xs$ 

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

lemma remdups-on-antimono:
  ( $\bigwedge x y . f x y \implies g x y$ ) ⇒ 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:  $(\bigwedge x y . f x y \implies f y x) \implies$ 
  distinct-on f xs  $\implies$  x $\in$ set xs  $\implies$  y $\in$ set xs  $\implies$  x $\neq$ y  $\implies$   $\neg$  f x y
by (induction xs) auto

```

```

fun lookup :: ('a  $\implies$  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)  $\implies$  (k, v)  $\in$  set al  $\longleftrightarrow$  lookup
  ( $\lambda x. x=k$ ) 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  $\implies$  p $\in$ set l  $\wedge$  P p
by (induction l) (auto split: if-splits)

```

```

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

```

```

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

```

```

lemma lookup-None-iff-find-None: lookup P xs = None  $\longleftrightarrow$  List.find ( $\lambda(k,-). P$ 
  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 ( $\lambda x. x=k$ ) pairs = lookup ( $\lambda x. x=k$ ) pairs'

```

```

proof (cases lookup ( $\lambda x. x=k$ ) 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 $el: (k,v) \in \text{set pairs}'$ **using** $\text{assms}(3)$ **by** blast
show $?thesis$ **using** $\text{lookup-present-eq-key}[OF \text{assms}(2)]$ $el \text{ Some}$ **by** simp
qed

lemma $\text{lookup-Some-append-back}$:

$\text{lookup } (\lambda x. x=k) \text{ insts} = \text{Some } v \implies \text{lookup } (\lambda x. x=k) (\text{insts}@[(k,v')]) = \text{Some } v$

by $(\text{induction insts arbitrary:})$ auto

lemma $\text{lookup-eq-key-not-present}$: $\text{key} \notin \text{set } (\text{map fst inst}) \implies \text{lookup } (\lambda x. x = \text{key}) \text{ inst} = \text{None}$

by (induction inst) auto

lemma $\text{lookup-in-empty}[simp]$: $\text{lookup } f [] = \text{None}$ **by** simp

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

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

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

by (induction al) $(\text{auto simp add: rev-image-eqI split: if-splits})$

lemma $\text{key-present-imp-eq-lookup-finds-value}$: $k \in \text{fst ' set al} \implies \exists v . \text{lookup } (\lambda x. x=k) \text{ al} = \text{Some } v$

by (induction al) $(\text{auto simp add: rev-image-eqI})$

lemma list-allI : $(\bigwedge x. x \in \text{set } l \implies P x) \implies \text{list-all } P l$

by $(\text{induction } l)$ auto

lemma map2-sym : $(\bigwedge x y . f x y = f y x) \implies \text{map2 } f \text{ xs ys} = \text{map2 } f \text{ ys xs}$

proof $(\text{induction xs arbitrary: ys})$

case Nil

then show $?case$ **by** simp

next

case $(\text{Cons } a \text{ xs})$

then show $?case$ **by** (induction ys) auto

qed

lemma idem-map2 : **assumes** $(\bigwedge x. f x x = x)$ **shows** $\text{map2 } f \text{ l l} = l$

proof–

have $\text{length } l = \text{length } l$ **by** simp

then show $\text{map2 } f \text{ l l} = l$ **by** $(\text{induction l l rule: list-induct2})$ $(\text{use assms in auto})$

qed

lemma $\text{rev-induct2}[consumes 1, case-names \text{Nil snoc}]$:

assumes $\text{length xs} = \text{length ys}$

assumes $P [] []$

assumes ($\bigwedge x xs y ys. \text{length } xs = \text{length } ys \implies P \text{ } xs \text{ } ys \implies P (xs @ [x]) (ys @ [y])$)

shows $P \text{ } xs \text{ } ys$

proof–

have $\text{length } (\text{rev } xs) = \text{length } (\text{rev } ys)$ **using** $\text{assms}(1)$ **by** simp

hence $P (\text{rev } (\text{rev } xs)) (\text{rev } (\text{rev } ys))$

using $\text{assms}(2-3)$ **by** ($\text{induction rule: list-induct2[of rev xs rev ys]$) simp-all

thus $?thesis$ **by** simp

qed

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

by simp

lemma $\text{distinct-fst-imp-distinct: distinct } (\text{map fst } l) \implies \text{distinct } l$

by ($\text{induction } l$) auto

lemma length-alist:

assumes $\text{distinct } (\text{map fst } al)$ **and** $\text{distinct } (\text{map fst } al')$ **and** $\text{set } al = \text{set } al'$

shows $\text{length } al = \text{length } al'$

using assms **by** ($\text{metis distinct-card length-map set-map}$)

lemma $\text{same-map-of-imp-same-length:}$

$\text{distinct } (\text{map fst } \text{ars1}) \implies \text{distinct } (\text{map fst } \text{ars2}) \implies \text{map-of } \text{ars1} = \text{map-of } \text{ars2}$

$\implies \text{length } \text{ars1} = \text{length } \text{ars2}$

using $\text{length-alist map-of-inject-set}$ **by** blast

lemma $\text{in-range-if-ex-key: } v \in \text{ran } m \longleftrightarrow (\exists k. m \text{ } k = \text{Some } v)$

by ($\text{auto simp add: ranI ran-def}$)

lemma $\text{set-AList-delete-bound: set } (\text{AList.delete } a \text{ } l) \subseteq \text{set } l$

by ($\text{induction } l$) auto

lemma $\text{list-all-clearjunk-cons:}$

$\text{list-all } P (x\#(\text{AList.clearjunk } l)) \implies \text{list-all } P (\text{AList.clearjunk } (x\#l))$

by ($\text{induction } l \text{ rule: AList.clearjunk.induct}$) ($\text{auto simp add: delete-twist}$)

lemma $\text{lookup-AList-delete: } k' \neq k \implies \text{lookup } (\lambda x. x = k) \text{ } al = \text{lookup } (\lambda x. x = k) (\text{AList.delete } k' \text{ } al)$

by ($\text{induction } al$) auto

lemma $\text{lookup-AList-clearjunk: lookup } (\lambda x. x = k) \text{ } al = \text{lookup } (\lambda x. x = k) (\text{AList.clearjunk } al)$

proof ($\text{induction } al$)

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* $xs\ ys \equiv \text{fold removeAll } ys\ xs$

lemma *diff-list-set[simp]*: $\text{set } (\text{diff-list } xs\ ys) = \text{set } xs - \text{set } ys$
unfolding *diff-list-def* **by** (*induction ys arbitrary: xs*) *auto*

lemma *diff-list-set-from-Nil[simp]*: $\text{diff-list } []\ ys = []$
using *last-in-set* **by** *fastforce*

lemma *diff-list-set-remove-Nil[simp]*: $\text{diff-list } xs\ [] = xs$
unfolding *diff-list-def* **by** (*induction xs*) *auto*

lemma *diff-list-rec*: $\text{diff-list } (x \# xs)\ ys = (\text{if } x \in \text{set } ys \text{ then } \text{diff-list } xs\ ys \text{ else } x \# \text{diff-list } xs\ ys)$

unfolding *diff-list-def* **by** (*induction ys arbitrary: x xs*) *auto*

lemma *diff-list-order-irr*: $\text{set } ys = \text{set } ys' \implies \text{diff-list } xs\ ys = \text{diff-list } xs\ ys'$

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*:
 $fold (\lambda new\ option . Option.bind\ option\ (f\ new))\ list\ start = Some\ res$
 $\implies start \neq None$
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*:
 $fold (\lambda new\ option . Option.bind\ option\ (f\ new))\ (l1@l2)\ start = Some\ res$
 $\implies fold (\lambda new\ option . Option.bind\ option\ (f\ new))\ (l1)\ start \neq None$
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'*:
 $fold (\lambda(x,y)\ option . Option.bind\ option\ (f\ x\ y))\ list\ start = Some\ res$
 $\implies start \neq None$
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*:
 $fold (\lambda(x,y)\ option . Option.bind\ option\ (f\ x\ y))\ list\ None = None$
by (*induction list*) (*auto split: option.splits if-splits prod.splits*)

lemma *fold-Option-bind-at-some-point-None-eq-None*:
 $fold (\lambda(x,y)\ option . Option.bind\ option\ (f\ x\ y))\ l1\ start = None \implies$
 $fold (\lambda(x,y)\ option . Option.bind\ option\ (f\ x\ y))\ (l1@l2)\ start = None$
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*:
 $fold (\lambda(x,y)\ option . Option.bind\ option\ (f\ x\ y))\ (l1@l2)\ start = Some\ res$
 $\implies (\exists\ point . fold (\lambda(x,y)\ option . Option.bind\ option\ (f\ x\ y))\ l1\ start = Some\ point$
 $\wedge fold (\lambda(x,y)\ option . Option.bind\ option\ (f\ x\ y))\ l2\ (Some\ point) = Some\ res)$
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* ($\lambda(x,y)$ *option* . *Option.bind option* (*f x y*)) (*xs@ys*) *start = Some res*
obtains *point* **where**
fold ($\lambda(x,y)$ *option* . *Option.bind option* (*f x y*)) *xs start = Some point* **and**
fold ($\lambda(x,y)$ *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 ($\lambda(x,y)$ *option* . *Option.bind option* (*f x y*)) (*l1@l2*) *start = Some res*
 \implies *fold* ($\lambda(x,y)$ *option* . *Option.bind option* (*f x y*)) (*l1*) *start \neq None*
using *fold-Option-bind-eq-Some-at-each-point-Some* **by** *fast*

lemma *fold-matches-first-step-not-None*:
assumes
fold ($\lambda(T, U)$ *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 ($\lambda(T, U)$ *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 ($\lambda(T, U)$ *subs* . *Option.bind subs* (*f T U*)) (*zip* (*xs@[x]*) (*ys@[y]*)) (*Some subs*) = *Some subs'*
obtains *point* **where**
fold ($\lambda(T, U)$ *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  $\Rightarrow$  (variable  $\times$  sort) set where
  tvsT (Tv v S) = {(v,S)}
| tvsT (Ty - Ts) =  $\bigcup$ (set (map tvsT Ts))
```

```
fun tvs :: term  $\Rightarrow$  (variable  $\times$  sort) set where
  tvs (Ct - T) = tvsT T
| tvs (Fv - T) = tvsT T
| tvs (Bv -) = {}
| tvs (Abs T t) = tvsT T  $\cup$  tvs t
| tvs (t $ u) = tvs t  $\cup$  tvs u
```

```
abbreviation tvs-set S  $\equiv$   $\bigcup$  t $\in$ S . tvs t
```

```
lemma tvsT-tsubstT: tvsT (tsubstT  $\sigma$   $\rho$ ) =  $\bigcup$  {tvsT ( $\rho$  a s) | a s. (a, s)  $\in$  tvsT  $\sigma$ }
by (induction  $\sigma$ ) fastforce+
```

```
lemma tsubstT-cong:
  ( $\forall$  (v,S)  $\in$  tvsT  $\sigma$ .  $\rho1$  v =  $\rho2$  v)  $\implies$  tsubstT  $\sigma$   $\rho1$  = tsubstT  $\sigma$   $\rho2$ 
by (induction  $\sigma$ ) fastforce+
```

```
lemma tsubstT-ith: i < length Ts  $\implies$  map ( $\lambda$ T . tsubstT T  $\rho$ ) Ts ! i = tsubstT
(Ts ! i)  $\rho$ 
by simp
```

```
lemma tsubstT-fun-typ-dist: tsubstT (T  $\rightarrow$  T1)  $\rho$  = tsubstT T  $\rho$   $\rightarrow$  tsubstT T1  $\rho$ 
by simp
```

```
fun subst :: term  $\Rightarrow$  (variable  $\Rightarrow$  typ  $\Rightarrow$  term)  $\Rightarrow$  term where
  subst (Ct s T)  $\rho$  = Ct s T
| subst (Fv v T)  $\rho$  =  $\rho$  v T
| subst (Bv i) - = Bv i
| subst (Abs T t)  $\rho$  = Abs T (subst t  $\rho$ )
| subst (t $ u)  $\rho$  = subst t  $\rho$  $ subst u  $\rho$ 
```

```
definition tinst t1 t2  $\equiv$   $\exists$   $\rho$ . tsubst t2  $\rho$  = t1
```

```
definition inst t1 t2  $\equiv$   $\exists$   $\rho$ . subst t2  $\rho$  = t1
```

fun *SortsT* :: *typ* \Rightarrow *sort set* **where**
 SortsT (*Tv* - *S*) = {*S*}
 | *SortsT* (*Ty* - *Ts*) = (\bigcup *T* \in *set* *Ts* . *SortsT* *T*)

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

fun *Types* :: *term* \Rightarrow *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* \cup *Types* *u*

abbreviation *tvS-Set* *S* \equiv \bigcup *s* \in *S* . *tvS* *s*
abbreviation *tvST-Set* *S* \equiv \bigcup *s* \in *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* = {} \implies *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* \implies *finite* (*FV* *S*)

by (*induction* *S* *rule: finite-induct*) *auto*

lemma *finite-tvS-Set*: *finite* *S* \implies *finite* (*tvS-Set* *S*)

by (*induction* *S* *rule: finite-induct*) *auto*

lemma *finite-tvST-Set*: *finite* *S* \implies *finite* (*tvST-Set* *S*)

by (*induction* *S* *rule: finite-induct*) *auto*

lemma *no-tvS-imp-tsubst-unchanged*: *tvS* *t* = {} \implies *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* = {} \implies *subst* *t* ϱ = *t*

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

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

```

fun typ-of1 :: typ list  $\Rightarrow$  term  $\Rightarrow$  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) ( $\lambda x$ . Some (T  $\rightarrow$ 
x))
| typ-of1 Ts (t $ u) = Option.bind (typ-of1 Ts u) ( $\lambda U$ . Option.bind (typ-of1 Ts t)
( $\lambda T$ .
  case T of
    Ty fun [T1,T2]  $\Rightarrow$  if fun = STR "fun" then
      if T1=U then Some T2 else None
      else None
    | -  $\Rightarrow$  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 \implies 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 \implies 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 \implies 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 \iff 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 \iff 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 \implies has-typ t ty*

by *simp*

lemma *typ-of1-weaken-Ts*: *typ-of1 Ts t = Some ty \implies 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 \implies has-typ1 (map ($\lambda T. tsubstT T \rho$) 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 \equiv 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 \Rightarrow nat \Rightarrow bool* **where**
loose-bvar (Bv i) k \longleftrightarrow i \geq k
| *loose-bvar (t \$ u) k \longleftrightarrow loose-bvar t k \vee loose-bvar u k*
| *loose-bvar (Abs - t) k = loose-bvar t (k+1)*
| *loose-bvar - - = False*

fun *loose-bvar1* :: *term \Rightarrow nat \Rightarrow bool* **where**
loose-bvar1 (Bv i) k \longleftrightarrow i = k
| *loose-bvar1 (t \$ u) k \longleftrightarrow loose-bvar1 t k \vee 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 \implies loose-bvar t n*
by (*induction t arbitrary: n*) *auto*

lemma *not-loose-bvar-imp-not-loose-bvar1*: \neg *loose-bvar t n \implies \neg loose-bvar1 t n
by (*induction t arbitrary: n*) *auto**

lemma *loose-bvar-iff-exist-loose-bvar1*: $\text{loose-bvar } t \text{ lev} \longleftrightarrow (\exists \text{lev}' \geq \text{lev}. \text{loose-bvar1 } t \text{ lev}')$

by (*induction t arbitrary: lev*) (*auto dest: Suc-le-D*)

definition *is-open* $t \equiv \text{loose-bvar } t \ 0$

abbreviation *is-closed* $t \equiv \neg \text{is-open } t$

definition *is-dependent* $t \equiv \text{loose-bvar1 } t \ 0$

lemma *loose-bvar-Suc*: $\text{loose-bvar } t \ (\text{Suc } k) \Longrightarrow \text{loose-bvar } t \ k$

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

lemma *loose-bvar-leq*: $k \geq p \Longrightarrow \text{loose-bvar } t \ k \Longrightarrow \text{loose-bvar } t \ p$

by (*induction rule: inc-induct*) (*use loose-bvar-Suc in auto*)

lemma *has-typ1-imp-no-loose-bvar*: $\text{has-typ1 } Ts \ t \ ty \Longrightarrow \neg \text{loose-bvar } t \ (\text{length } Ts)$

by (*induction rule: has-typ1.induct*) *auto*

corollary *has-typ-imp-closed*: $\text{has-typ } t \ ty \Longrightarrow \neg \text{is-open } t$

unfolding *is-open-def has-typ-def using has-typ1-imp-no-loose-bvar by fastforce*

corollary *typ-of-imp-closed*: $\text{typ-of } t = \text{Some } ty \Longrightarrow \neg \text{is-open } t$

by (*simp add: has-typ-imp-closed*)

Subterms

fun *exists-subterm* :: $(\text{term} \Rightarrow \text{bool}) \Rightarrow \text{term} \Rightarrow \text{bool}$ **where**

exists-subterm $P \ t \longleftrightarrow P \ t \vee (\text{case } t \text{ of}$
 $(t \ \$ \ u) \Rightarrow \text{exists-subterm } P \ t \vee \text{exists-subterm } P \ u$
 $| \text{Abs } ty \ \text{body} \Rightarrow \text{exists-subterm } P \ \text{body}$
 $| - \Rightarrow \text{False})$

fun *exists-subterm'* :: $(\text{term} \Rightarrow \text{bool}) \Rightarrow \text{term} \Rightarrow \text{bool}$ **where**

exists-subterm' $P \ (t \ \$ \ u) \longleftrightarrow P \ (t \ \$ \ u) \vee \text{exists-subterm}' \ P \ t \vee \text{exists-subterm}'$
 $P \ u$
 $| \text{exists-subterm}' \ P \ (\text{Abs } ty \ \text{body}) \longleftrightarrow P \ (\text{Abs } ty \ \text{body}) \vee \text{exists-subterm}' \ P \ \text{body}$
 $| \text{exists-subterm}' \ P \ t \longleftrightarrow P \ t$

lemma *exists-subterm-iff-exists-subterm'*: $\text{exists-subterm } P \ t \longleftrightarrow \text{exists-subterm}' \ P \ t$

by (*induction t*) *auto*

lemma *exists-subterm* $(\lambda t. t = \text{Fv } idx \ T) \ t \longleftrightarrow (idx, T) \in \text{fv } t$

by (*induction t*) *auto*

abbreviation *occs* $t \ u \equiv \text{exists-subterm } (\lambda s. t = s) \ u$

lemma *occs-Fv-eq-elem-fv*: $\text{occs } (\text{Fv } v \ S) \ t \longleftrightarrow (v, S) \in \text{fv } t$

by (*induction t*) *auto*

lemma *bind-fv2-unchanged*:

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

lemma bind-fv2-changed:
 $\text{bind-fv2 } v \text{ lev } tm \neq tm \implies v \in \text{fv } tm$
by (induction v lev tm rule: bind-fv2.induct) (auto split: if-splits)

lemma bind-fv2-changed':
 $\text{bind-fv2 } v \text{ lev } tm \neq tm \implies \text{occs } (\text{case-prod } Fv \ v) \ tm$
by (induction v lev tm rule: bind-fv2.induct) (auto split: if-splits)

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

corollary bind-fv-unchanged: $(x, \tau) \notin \text{fv } t \implies \text{bind-fv } (x, \tau) \ t = t$
using bind-fv-changed **by** auto

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

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

corollary typ-of-arg: $\text{typ-of } (t\$u) = \text{Some } R \implies \text{typ-of } u = \text{Some } T \implies \text{typ-of } t = \text{Some } (T \rightarrow R)$
by (metis typ-of1-arg-typ typ-of-def)

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

corollary typ-of-fun: $\text{typ-of } (t\$u) = \text{Some } R \implies \text{typ-of } t = \text{Some } (U \rightarrow R) \implies \text{typ-of } u = \text{Some } U$
by (metis typ-of1-fun-typ typ-of-def)

lemma *typ-of-eta-expand*: $\text{typ-of } f = \text{Some } (\tau \rightarrow \tau') \implies \text{typ-of } (\text{Abs } \tau (f \ \$ \ \text{Bv } 0)) = \text{Some } (\tau \rightarrow \tau')$

using *typ-of1-weaken* **by** (*fastforce simp add: bind-eq-Some-conv typ-of-def*)

lemma *bind-fv2-preserves-type*:

assumes *typ-of1* $Ts \ t = \text{Some } ty$

shows *typ-of1* $(Ts@[T]) \ (\text{bind-fv2 } (v, T) \ (\text{length } Ts) \ t) = \text{Some } ty$

using *assms* **by** (*induction* $(v, T) \ \text{length } Ts \ t$ *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* $A = \text{Some } ty$

shows *typ-of* $(\text{Abs } bT \ (\text{bind-fv } (v, bT) \ A)) = \text{Some } (bT \rightarrow ty)$

using *bind-fv2-preserves-type bind-fv-def assms typ-of-def* **by** *fastforce*

corollary *typ-of-Abs-fv*:

assumes *typ-of* $A = \text{Some } ty$

shows *typ-of* $(\text{Abs-fv } v \ bT \ A) = \text{Some } (bT \rightarrow ty)$

using *assms typ-of-Abs-bind-fv typ-of-def* **by** *simp*

lemma *typ-of-mk-all*:

assumes *typ-of* $A = \text{Some } \text{prop}T$

shows *typ-of* $(\text{mk-all } x \ ty \ A) = \text{Some } \text{prop}T$

using *typ-of-Abs-bind-fv[OF assms, of ty]* **by** (*auto simp add: typ-of-def*)

fun *incr-bv* :: $\text{nat} \Rightarrow \text{nat} \Rightarrow \text{term} \Rightarrow \text{term}$ **where**

incr-bv *inc* $n \ (\text{Bv } i) = (\text{if } i \geq n \ \text{then } \text{Bv } (i+\text{inc}) \ \text{else } \text{Bv } i)$

| *incr-bv* *inc* $n \ (\text{Abs } T \ \text{body}) = \text{Abs } T \ (\text{incr-bv } \text{inc} \ (n+1) \ \text{body})$

| *incr-bv* *inc* $n \ (\text{App } f \ t) = \text{App } (\text{incr-bv } \text{inc} \ n \ f) \ (\text{incr-bv } \text{inc} \ n \ t)$

| *incr-bv* - - $u = u$

lemma *lift-def*: $\text{lift } t \ n = \text{incr-bv } 1 \ n \ t$

by (*induction t n rule: lift.induct*) *auto*

declare *lift.simps[simp del]*

declare *lift-def[simp]*

definition *incr-boundvars* $\text{inc } t = \text{incr-bv } \text{inc} \ 0 \ t$

fun *decr* :: $\text{nat} \Rightarrow \text{term} \Rightarrow \text{term}$ **where**

decr *lev* $(\text{Bv } i) = (\text{if } i \geq \text{lev} \ \text{then } \text{Bv } (i - 1) \ \text{else } \text{Bv } i)$

| *decr* *lev* $(\text{Abs } T \ t) = \text{Abs } T \ (\text{decr } (\text{lev} + 1) \ t)$

| *decr* *lev* $(t \ \$ \ u) = (\text{decr } \text{lev} \ t \ \$ \ \text{decr } \text{lev} \ u)$

| *decr* - $t = t$

lemma *incr-bv-0[simp]*: $\text{incr-bv } 0 \ \text{lev} \ t = t$

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

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

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

lemma *is-close-no-incr-boundvars[simp]*: $is-closed\ t \implies incr-boundvars\ inc\ t = t$
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*: $\neg\ loose-bvar\ t\ k \implies \neg\ loose-bvar\ (decr\ k\ t)\ k$
by (*induction t k rule: loose-bvar.induct*) *auto*

lemma *loose-bvar-decr-unchanged[simp]*: $\neg\ loose-bvar\ t\ k \implies decr\ k\ t = t$
by (*induction t k rule: loose-bvar.induct*) *auto*

lemma *is-closed-decr-unchanged[simp]*: $is-closed\ t \implies decr\ 0\ t = t$
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]*: $i < j \implies subst-bv2\ (Bv\ j)\ i\ u = Bv\ (j - 1)$
by (*simp add:*)

lemma *subst-bv2-subst-lt* [*simp*]: $j < i \implies \text{subst-bv2 } (Bv\ j)\ i\ u = Bv\ j$
by (*simp add*);

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

lemma *lift-subst* [*simp*]:
 $j < i + 1 \implies \text{lift } (\text{subst-bv2 } t\ j\ s)\ i = \text{subst-bv2 } (\text{lift } t\ (i + 1))\ j\ (\text{lift } s\ i)$
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*:
 $i < j + 1 \implies \text{lift } (\text{subst-bv2 } t\ j\ s)\ i = \text{subst-bv2 } (\text{lift } t\ i)\ (j + 1)\ (\text{lift } s\ i)$
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*]:
 $\text{subst-bv2 } (\text{lift } t\ k)\ k\ s = t$
by (*induct t arbitrary: k s simp-all*)

lemma *subst-bv2-subst-bv2*:
 $i < j + 1 \implies \text{subst-bv2 } (\text{subst-bv2 } t\ (\text{Suc } j)\ (\text{lift } v\ i))\ i\ (\text{subst-bv2 } u\ j\ v)$
 $= \text{subst-bv2 } (\text{subst-bv2 } t\ i\ u)\ j\ v$
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*: $\text{subst-bv } u\ t \equiv \text{subst-bv1 } t\ 0\ u$
by (*simp add: substn-subst-0' substn-subst-n*)

fun *subst-bvs1* :: *term* \Rightarrow *nat* \Rightarrow *term list* \Rightarrow *term* **where**
 $\text{subst-bvs1 } (Bv\ n)\ lev\ args = (\text{if } n < lev$
 $\text{then } Bv\ n$
 $\text{else if } n - lev < \text{length } args$

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

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

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

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

lemma $\text{no-loose-bvar-imp-no-subst-bv1}$: $\neg \text{loose-bvar t lev} \implies \text{subst-bv1 t lev u} = t$
by (*induction t arbitrary: lev*) *auto*

lemma $\text{no-loose-bvar-imp-no-subst-bvs1}$: $\neg \text{loose-bvar t lev} \implies \text{subst-bvs1 t lev us} = t$
by (*induction t arbitrary: lev*) *auto*

lemma subst-bvs1-step :

assumes $\neg \text{loose-bvar t lev}$

shows $\text{subst-bvs1 t lev (args@[u])} = \text{subst-bv1 (subst-bvs1 t lev args) lev u}$

using *assms* **by** (*induction t arbitrary: lev args u*) *auto*

corollary $\text{closed-subst-bv-no-change}$: $\text{is-closed t} \implies \text{subst-bv u t} = t$

unfolding $\text{is-open-def subst-bv-def no-loose-bvar-imp-no-subst-bv1}$ **by** *simp*

lemma $\text{is-variable-imp-incr-bv-unchanged}$: $\text{incr-bv inc lev (Fv v T)} = (\text{Fv v T})$
by *simp*

lemma $\text{is-variable-imp-incr-boundvars-unchanged}$: $\text{incr-boundvars inc (Fv v T)} = (\text{Fv v T})$

using $\text{is-variable-imp-incr-bv-unchanged incr-boundvars-def}$ **by** *simp*

lemma $\text{loose-bvar-subst-bv1}$:

$\neg \text{loose-bvar (subst-bv1 t lev u) lev} \implies \neg \text{loose-bvar t (Suc lev)}$

by (*induction t lev u rule: subst-bv1.induct*) *auto*

lemma $\text{is-closed-subst-bv}$: $\text{is-closed (subst-bv u t)} \implies \neg \text{loose-bvar t 1}$

by (*simp add: is-open-def loose-bvar-subst-bv1 subst-bv-def*)

lemma $\text{subst-bv1-bind-fv2}$:

assumes $\neg \text{loose-bvar t lev}$

shows $\text{subst-bv1 (bind-fv2 (v, T) lev t) lev (Fv v T)} = t$

using *assms* **by** (*induction t arbitrary: lev*) (*use is-variable-imp-incr-boundvars-unchanged 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  $\Rightarrow$  term  $\Rightarrow$  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  $\implies$   $\neg$  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  $\implies$  ( $\exists$  uty . typ-of1 Ts t = Some (uty  $\rightarrow$  ty)  $\wedge$ 
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  $\rightarrow$  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 $lev \leq length\ Ts$
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** $Ts' \neq []$
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.prem1 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*:
 $\neg \text{exists-subterm } (\lambda x . \text{case } x \text{ of } Bv - \Rightarrow \text{True} \mid - \Rightarrow \text{False}) t$
 $\Rightarrow \text{typ-of1 } Ts t = \text{typ-of1 } Ts' t$
by (*induction t arbitrary: Ts Ts' (simp-all, metis+)*)

lemma *typ-of1-drop-extra-bounds*:
 $\neg \text{loose-bvar } t (\text{length } Ts)$
 $\Rightarrow \text{typ-of1 } (Ts@rest) t = \text{typ-of1 } Ts t$
by (*induction Ts t arbitrary: rest rule: typ-of1.induct (fastforce simp add: nth-append)+*)

lemma *typ-of-betapply*:
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:  $\neg$  eta-reducible (Abs T t)  $\implies$   $\neg$  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  $\implies$  lev < lev'  $\implies$  loose-bvar1 (decr
lev' t) lev
by (induction t arbitrary: lev lev') auto
lemma loose-bvar1-decr''': loose-bvar1 t (Suc lev)  $\implies$  lev'  $\leq$  lev  $\implies$  loose-bvar1
(decr lev' t) lev
by (induction t arbitrary: lev lev') auto

lemma loose-bvar1-decr'''':  $\neg$  loose-bvar1 t lev'  $\implies$  lev'  $\leq$  lev  $\implies$   $\neg$  loose-bvar1
t (Suc lev)
 $\implies$   $\neg$  loose-bvar1 (decr lev' t) lev
by (induction lev t arbitrary: lev' rule: decr.induct) auto

lemma not-eta-reducible-decr:
 $\neg$  eta-reducible t  $\implies$   $\neg$  loose-bvar1 t lev  $\implies$   $\neg$  eta-reducible (decr lev t)
proof (induction lev t arbitrary: rule: decr.induct)
case (2 lev T body)
hence  $\neg$  eta-reducible body using eta-reducible-Abs by blast
hence I:  $\neg$  eta-reducible (decr (lev + 1) body) using 2.IH
using 2.prem(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  $\neg$  eta-reducible f
using 0 2.prem(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, domintrors) eta-norm :: term  $\Rightarrow$  term where
  eta-norm (Abs T t) = (case eta-norm t of
    f $ Bv 0  $\Rightarrow$  (if is-dependent f then Abs T (f $ Bv 0) else decr 0 (eta-norm f))
  | body  $\Rightarrow$  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  $\Longrightarrow$  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.domintrors 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  $\neg$  is-dependent t eta-norm-dom body
  have depth t < depth (t $ Bv 0) by auto
  moreover have depth (eta-norm body)  $\leq$  depth body using asms eta-norm-reduces-depth
by blast
  ultimately show (t, Abs T body)  $\in$  measure depth using asms by (auto simp add: eta-norm.psimps)
qed simp-all

```

```

lemma loose-bvar1-eta-norm: loose-bvar1 t lev  $\Longrightarrow$  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':  $\neg$  loose-bvar1 t lev  $\Longrightarrow$   $\neg$  loose-bvar1 (eta-norm t) lev
proof (induction t arbitrary: lev rule: eta-norm.induct)
  case (1 T body)
  hence  $\neg$  loose-bvar1 body (Suc lev) by simp
  hence I:  $\neg$  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:  $\neg$  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  $\neg$  eta-reducible f using 1 App u by simp
    hence  $\neg$  eta-reducible (eta-norm f)
      by (simp add: 1.IH(2) App False u)
    have  $\neg$  loose-bvar1 f 0
      using False is-dependent-def by blast
    hence  $\neg$  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  $\langle \neg$ 
loose-bvar1 (eta-norm f) 0  $\rangle$ 
      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*: $\neg \text{eta-reducible } t \implies \text{eta-norm } t = t$
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*: $\text{eta-norm } (\text{eta-norm } t) = \text{eta-norm } t$
using *not-eta-reducible-imp-eta-norm-no-change not-eta-reducible-eta-norm* **by**
blast

lemma *typ-of1-decr*: $\text{typ-of1 } (Ts@[T]@Ts') t = \text{Some } ty \implies \neg \text{loose-bvar1 } t (\text{length } Ts)$
 $\implies \text{typ-of1 } (Ts@Ts') (\text{decr } (\text{length } Ts) t) = \text{Some } ty$
proof (*induction t arbitrary: Ts T Ts' ty*)
case (*Abs b 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*)

lemma *typ-of1-decr-gen*: $\text{typ-of1 } (Ts@[T]@Ts') t = \text{tyo} \implies \neg \text{loose-bvar1 } t (\text{length } Ts)$
 $\implies \text{typ-of1 } (Ts@Ts') (\text{decr } (\text{length } Ts) t) = \text{tyo}$
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'*: $\text{typ-of1 } (Ts@Ts') (\text{decr } (\text{length } Ts) t) = \text{tyo} \implies \neg \text{loose-bvar1 } t (\text{length } Ts)$
 $\implies \text{typ-of1 } (Ts@[T]@Ts') t = \text{tyo}$
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*: $\text{typ-of1 } Ts t = \text{Some } ty \implies \text{typ-of1 } Ts (\text{eta-norm } t) = \text{Some } ty$
proof (*induction Ts t arbitrary: ty rule: typ-of1.induct*)

```

case ( $\lambda$   $Ts$   $T$   $body$ )
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 ( $auto$   $simp$   $add$ :  $App$   $0$   $\lambda$ . $IH$   $Bv$   $bind$ -eq- $Some$ -conv  $is$ -dependent-def
split:  $nat$ . $splits$ )
    then show  $?thesis$ 
    using  $\lambda$  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 ( $auto$   $simp$   $add$ :  $App$   $0$   $\lambda$ . $IH$   $Bv$   $bind$ -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  $\lambda$ . $prems$  by  $fastforce$ 
  hence  $typ$ -of1 ( $T$   $\#$   $Ts$ ) ( $eta$ -norm  $body$ ) =  $Some$   $bT$ 
    using  $\lambda$ . $IH$  by  $blast$ 
  moreover have  $T \rightarrow bT = ty$ 
    using  $\lambda$ . $prems$   $bT$  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*: $\text{typ-of } t = \text{Some } ty \implies \text{typ-of } (\text{eta-norm } t) = \text{Some } ty$
 using *typ-of1-eta-norm typ-of-def* by *simp*

lemma *typ-of-Abs-body-ty*: $\text{typ-of1 } Ts (\text{Abs } T t) = \text{Some } ty \implies \exists rty. ty = (T \rightarrow rty)$

by (*metis (no-types, lifting) bind-eq-Some-conv option.sel typ-of1.simps(4)*)

lemma *typ-of-Abs-body-ty'*: $\text{typ-of1 } Ts (\text{Abs } T t) = \text{Some } ty$

$\implies \exists rty. ty = (T \rightarrow rty) \wedge \text{typ-of1 } (T \# Ts) t = \text{Some } rty$

by (*metis (no-types, lifting) bind-eq-Some-conv option.sel typ-of1.simps(4)*)

lemma *typ-of-beta-redex-arg*: $\text{typ-of } (\text{Abs } T s \$ t) \neq \text{None} \implies \text{typ-of } t = \text{Some } T$

by (*metis list.inject not-Some-eq typ.inject(1) typ-of1-split-App typ-of-Abs-body-ty' typ-of-def*)

lemma [*partial-function-mono*]: *option.mono-body*

($\lambda \text{beta-norm. map-option } (\text{Abs } T) (\text{beta-norm } t)$)

by (*smt flat-ord-def fun-ord-def map-option-is-None monotone-def*)

lemma [*partial-function-mono*]: *option.mono-body*

($\lambda \text{beta-norm.}$

case beta-norm x of None \implies None

| *Some (Ct list typ) \implies*

map-option (($\$$) (Ct list typ)) (beta-norm u)

| *Some (Fv p typ) \implies*

map-option (($\$$) (Fv p typ)) (beta-norm u)

| *Some (Bv n) \implies*

map-option (($\$$) (Bv n)) (beta-norm u)

| *Some (Abs T body) \implies*

beta-norm (subst-bv u body)

| *Some (term1 $\$$ term2) \implies*

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  $\Rightarrow$  term option where
  beta-norm t = (case t of
    (Abs T body)  $\Rightarrow$  map-option (Abs T) (beta-norm body)
  | (Abs T body $ u)  $\Rightarrow$  beta-norm (subst-bv u body)
  | (f $ u)  $\Rightarrow$  (case beta-norm f of
    Some (Abs T body)  $\Rightarrow$  beta-norm (subst-bv u body)
  | Some f'  $\Rightarrow$  map-option (App f') (beta-norm u)
  | None  $\Rightarrow$  None)
  | t  $\Rightarrow$  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:  $\neg$  beta-reducible t  $\Longrightarrow$  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:  $\neg$  beta-reducible t  $\Longrightarrow$   $\neg$  beta-reducible (decr n t)
  by (induction t arbitrary: n rule: beta-reducible.induct) auto

```

```

lemma  $\neg$  beta-reducible t  $\Longrightarrow$  eta-norm t = t'  $\Longrightarrow$   $\neg$  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.prem(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.prem(2) ⟨¬ beta-reducible u⟩ ⟨¬ beta-reducible f'⟩ ⟨eta-norm
f = f'⟩ 2(3)
    by (auto elim: beta-reducible.elims(3))
  qed (use 2.prem(2) in auto)

```

qed *auto*

fun *is-variable* :: *term* \Rightarrow *bool* **where**
 is-variable (*Fv* -) = *True*
| *is-variable* - = *False*

lemma *fv-occs*: $(x, \tau) \in \text{fv } t \implies \text{occs } (\text{Fv } x \ \tau) \ t$
by (*induction* *t*) *auto*

lemma *fv-iff-occs*: $(x, \tau) \in \text{fv } t \longleftrightarrow \text{occs } (\text{Fv } x \ \tau) \ t$
by (*induction* *t*) *auto*

fun *strip-abs* :: *term* \Rightarrow *typ list* * *term* **where**
 strip-abs (*Abs* *T* *t*) = (*let* (*a'*, *t'*) = *strip-abs* *t* *in* (*T* # *a'*, *t'*)
| *strip-abs* *t* = (\square , *t*)

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

fun *strip-abs-vars* :: *term* \Rightarrow *typ list* **where**
 strip-abs-vars (*Abs* *T* *t*) = *T* # *strip-abs-vars* *t*
| *strip-abs-vars* *u* = \square

fun *strip-qnt-body* :: *name* \Rightarrow *term* \Rightarrow *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* \Rightarrow *term* \Rightarrow *typ list* **where**
 strip-qnt-vars *qnt* (*Ct* *c* - \$ *Abs* *T* *t*) = (*if* *c=qnt* *then* *T* # *strip-qnt-vars* *qnt* *t*
 else \square)
| *strip-qnt-vars* *qnt* *t* = \square

definition *list-comb* :: *term* * *term list* \Rightarrow *term* **where** *list-comb* = *case-prod* (*foldl* (\$))

definition *list-comb'* :: *term* \Rightarrow *term list* \Rightarrow *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 \Rightarrow term * term list **where**
strip-comb u = *strip-comb-imp* (u, [])

fun *head-of* :: term \Rightarrow 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 \Rightarrow bool **where**
is-app (- \$ -) = True
| *is-app* - = False

lemma *not-is-app-imp-strip-com-imp-unchanged*: \neg *is-app* t \Longrightarrow *strip-comb-imp*
(t,ts) = (t,ts)
by (*cases* t) *simp-all*
corollary *not-is-app-imp-strip-com-unchanged*: \neg *is-app* t \Longrightarrow *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 \Rightarrow int \Rightarrow 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 \Rightarrow int \Rightarrow 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 \Rightarrow typ) \Rightarrow term \Rightarrow 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 $add-tvar-namesT' = fold-atyps (\lambda x l . case\ x\ of\ Tv\ xi\ - => List.insert\ xi\ l\ | - => l)$

definition $add-tvar-names' = fold-types\ add-tvar-namesT'$

definition $add-tvarsT' = fold-atyps (\lambda x l . case\ x\ of\ Tv\ idn\ s => List.insert\ (idn,s)\ l\ | - => l)$

definition $add-tvars' = fold-types\ add-tvarsT'$

definition $add-vars' = fold-aterms (\lambda x l . case\ x\ of\ Fv\ idn\ s => List.insert\ (idn,s)\ l\ | - => l)$

definition $add-var-names' = fold-aterms (\lambda x l . case\ x\ of\ Fv\ xi\ - => List.insert\ xi\ l\ | - => l)$

definition $add-const-names' = fold-aterms (\lambda x l . case\ x\ of\ Ct\ c\ - => List.insert\ c\ l\ | - => l)$

definition $add-consts' = fold-aterms (\lambda x l . case\ x\ of\ Ct\ n\ s => List.insert\ (n,s)\ l\ | - => l)$

definition $add-tvar-namesT = fold-atyps (\lambda x . case\ x\ of\ Tv\ xi\ - => insert\ xi\ | - => id)$

definition $add-tvar-names = fold-types\ add-tvar-namesT$

definition $add-tvarsT = fold-atyps (\lambda x . case\ x\ of\ Tv\ idn\ s => insert\ (idn,s)\ | - => id)$

definition $add-tvars = fold-types\ add-tvarsT$

definition $add-var-names = fold-aterms (\lambda x . case\ x\ of\ Fv\ xi\ - => insert\ xi\ | - => id)$

definition $add-vars = fold-aterms (\lambda x . case\ x\ of\ Fv\ idn\ s => insert\ (idn,s)\ | - => id)$

definition $add-const-names = fold-aterms (\lambda x . case\ x\ of\ Ct\ c\ - => insert\ c\ | - => id)$

definition $add-consts = fold-aterms (\lambda x . case\ x\ of\ Ct\ n\ s => insert\ (n,s)\ | - => id)$

```

lemma add-tvarsT'-tvsT-pre[simp]:  $set (add-tvarsT' T acc) = set acc \cup tvsT T$ 
  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]:  $set (add-tvars' t acc) = set acc \cup tvs t$ 
  by (induction t arbitrary: acc) (auto simp add: add-tvars'-def fold-types-def)

```

```

lemma add-tvarsT T acc = acc \cup 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:  $set (add-vars' t acc) = set acc \cup fv t$ 
  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  $\Rightarrow$  term where
  strip-all-body (Ct all S $ Abs T t) = (if all= STR "Pure.all" \wedge S=(T \to propT) \to propT
    then strip-all-body t else (Ct all S $ Abs T t))
| strip-all-body t = t

```

```

fun strip-all-vars :: term  $\Rightarrow$  typ list where
  strip-all-vars (Ct all S $ Abs T t) = (if all= STR "Pure.all" \wedge S=(T \to propT) \to propT
    then T # strip-all-vars t else [])
| strip-all-vars t = []

```

```

fun strip-all-single-body :: term  $\Rightarrow$  term where
  strip-all-single-body (Ct all S $ Abs T t) = (if all= STR "Pure.all" \wedge S=(T \to propT) \to propT
    then t else (Ct all S $ Abs T t))
| strip-all-single-body t = t

```

```

fun strip-all-single-var :: term  $\Rightarrow$  typ option where
  strip-all-single-var (Ct all S $ Abs T t) = (if all= STR "Pure.all" \wedge S=(T \to propT) \to 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 \implies 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 \longleftrightarrow strip-all-vars t \neq []*
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*:
 \neg *is-all t \implies 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 = (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 $\dots =$ *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 $\dots =$ *ty # rev (map snd vs)*
by (*simp add: I*)
also have $\dots =$ *rev (map snd (vs @ [v]))*
using *v* **by** *simp*
finally show *?case .*
qed

lemma *subst-bv-no-loose-unchanged*:
assumes $\bigwedge x . x \geq lev \implies \neg$ *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  $\neg\ occs\ (case-prod\ Fv\ v)\ t$ 
  shows  $(bind-fv2\ v\ lev\ t) = t$ 
  using assms by (induction t arbitrary: lev) auto

```

```

lemma bind-fv2-subst-bv1-cancel:
  assumes  $\bigwedge x . x > lev \implies \neg\ loose-bvar1\ t\ x$ 
  assumes  $\neg\ occs\ (case-prod\ Fv\ v)\ t$ 
  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-unchanged)
  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.premis 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.prem(1-2) False assms le-neq-implies-less loose-bvar1.simp(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.prem(2) bind-fv2-no-occs-unchanged
  using App.prem(2) bind-fv2-changed' exists-subterm'.simp(1)
  exists-subterm-iff-exists-subterm' by blast
ultimately show ?thesis by simp
qed
then show ?thesis using I1 App.prem is-variable.elim(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.prem(1-2) False le-neq-implies-less loose-bvar1.simp(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.prem(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.prem is-variable.elim(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.prem(1-2) False assms le-neq-implies-less loose-bvar1.simp(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.prem(2) bind-fv2-no-occs-unchanged by auto
ultimately show ?thesis by simp
qed
then show ?thesis using I1 App.prem is-variable.elim(2) by auto
qed
qed
qed auto

```

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

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

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

lemma $\text{mk-all'-subst-bv-strip-all-single-body-cancel}$:
assumes $\text{strip-all-single-var } t = \text{Some } T$
assumes $\text{is-closed } t$
assumes $(\text{name}, T) \notin \text{fv } t$
shows $\text{mk-all } \text{name } T \ (\text{subst-bv } (Fv \ \text{name } T) \ (\text{strip-all-single-body } t)) = t$
proof–
from $\text{assms}(1)$ **obtain** t' **where** $t': (Ct \ STR \ "Pure.all" \ ((T \rightarrow \text{prop } T) \rightarrow \text{prop } T))$
 $\$ \text{Abs } T \ t') = t$
by $(\text{auto } \text{elim!}: \text{strip-all-single-var.elims}$
 $\text{simp } \text{add: } \text{bind-eq-Some-conv } \text{typ-of-def } \text{split: } \text{if-splits } \text{option.splits } \text{if-splits})$

hence $s: \text{strip-all-single-body } t = t'$ **by** auto

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

have $0 < x \implies \neg \text{loose-bvar1 } t' \ x$ **for** x
using $\text{assms}(2)$ **by** $(\text{auto } \text{simp } \text{add: } \text{is-open-def } t'[\text{symmetric}] \ \text{loose-bvar-iff-exist-loose-bvar1}$
 $\text{gr0-conv-Suc})$

have $\text{occs } t' \ t$ **by** $(\text{simp } \text{add: } t'[\text{symmetric}])$

have $\text{bind-fv } (\text{name}, T) \ (\text{subst-bv } (Fv \ \text{name } T) \ (\text{strip-all-single-body } t)) =$
 $(\text{strip-all-single-body } t)$
using $\text{assms}(2-3)$ $\text{bind-fv-subst-bv-cancel } \text{gr0-conv-Suc}$
by $(\text{force } \text{simp } \text{add: } s \ \text{is-open-def } t'[\text{symmetric}]$
 $\text{loose-bvar-iff-exist-loose-bvar1 } \text{fv-iff-occs } \text{intro!}: \text{bind-fv-subst-bv-cancel})$

then show $?thesis$ **using** assms **by** $(\text{auto } \text{simp } \text{add: } s \ \text{typ-of-def } t')$

qed

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

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

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

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

corollary *mk-all-fv-unchanged*: $fv (mk\text{-}all\ x\ \tau\ B) = fv\ B - \{(x,\tau)\}$
using *bind-fv2-Fv-fv bind-fv-def* **by** *auto*

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

case (*snoc x xs*)
have $s: mk\text{-}all\text{-}list\ (xs@[x])\ B = case\text{-}prod\ mk\text{-}all\ x\ (mk\text{-}all\text{-}list\ xs\ B)$
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* $t\ Hs \equiv mk\text{-}all\text{-}list$
(*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* $cs\ s1\ s2 = (sort\text{-}leq\ cs\ s1\ s2 \wedge \neg\ sort\text{-}leq\ cs\ s2\ s1)$

definition *sort-eqv* $cs\ s1\ s2 = (sort\text{-}leq\ cs\ s1\ s2 \wedge sort\text{-}leq\ cs\ s2\ s1)$

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*: $sort\text{-}ex\ cs\ S \equiv \forall c \in S. class\text{-}ex\ cs\ c$
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*: $\neg\ class\text{-}les\ cs\ c\ c$
using *wf* **by** (*simp add: class-les-def*)

lemma *class-les-trans*: $class\text{-}les\ cs\ x\ y \implies class\text{-}les\ cs\ y\ z \implies class\text{-}les\ cs\ x\ z$

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

lemma *class-leq-refl*[*iff*]: *class-ex cs c* \implies *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* \implies *class-leq cs y z* \implies *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* \implies *class-leq cs c2 c1* \implies *c1=c2*
using *wf* **by** (*auto intro: antisymD simp: trans-def class-leq-def*)

lemma *sort-leq-refl*[*iff*]: *sort-ex cs s* \implies *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* \implies *sort-leq cs y z* \implies *sort-leq cs x z*
by (*meson class-leq-trans sort-leq-def*)

lemma *sort-leq-ex*: *sort-leq cs s1 s2* \implies *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 \implies $\exists s1'. (\forall c1 \in s1'. \exists c2 \in s2. \text{class-leq cs } c1 c2) \wedge \text{sort-leq cs } s1' s2$
by (*meson class-leq-refl sort-ex-class-ex sort-leq-ex sort-leq-refl*)

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

lemma *superset-imp-sort-leq*: *sort-ex cs s2* \implies *s1* \supseteq *s2* \implies *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* \implies *sort-leq cs s full-sort*
by (*simp add: sort-leq-def*)

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

lemma *sort-equivI*: *sort-leq cs s1 s2* \implies *sort-leq cs s2 s1* \implies *sort-equiv cs s1 s2*
by (*simp add: sort-equiv-def*)

lemma *sort-equiv-refl*: *sort-ex cs s* \implies *sort-equiv cs s s*
using *sort-leq-refl* **by** (*auto simp add: sort-equiv-def*)

lemma *sort-equiv-trans*: *sort-equiv cs x y* \implies *sort-equiv cs y z* \implies *sort-equiv cs x z*
using *sort-equiv-def sort-leq-trans* **by** *blast*

lemma *sort-equiv-sym*: *sort-equiv cs x y* \implies *sort-equiv cs y x*
by (*auto simp add: sort-equiv-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*: $\text{sort-ex } cs \ s \implies \text{sort-ex } cs \ (\text{normalize-sort } cs \ s)$
by (simp add: normalize-sort-def sort-ex-class-ex)

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

lemma *normalize-sort-removed-elem-irrelevant'*:
assumes *sort-ex cs (insert c s)*
assumes $c \notin (\text{normalize-sort } cs \ (\text{insert } c \ s))$
shows $\text{normalize-sort } cs \ (\text{insert } c \ s) = \text{normalize-sort } cs \ s$
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' ∈ s*
using *class-les-irrefl* **by** (auto simp add: normalize-sort-def)
thus *?thesis*
using $\langle \text{class-ex } cs \ c \rangle$ *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 $c \notin (\text{normalize-sort } cs \ (\text{insert } c \ s))$
shows $\text{normalize-sort } cs \ (\text{insert } c \ s) = \text{normalize-sort } cs \ s$
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 ≠ full-sort*
assumes *sort-ex cs s*
shows $\text{normalize-sort } cs \ s \neq \text{full-sort}$
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 $\text{normalize-sort } cs \ (\text{insert } c \ s) = \{c\}$
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 $\text{normalize-sort } cs \ s \neq \text{full-sort}$
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.premis(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 insert.premis(2)
insert.premis(3) insert-iff insert-subset normalize-sort-removed-elem-irrelevant'
sort-ex-def)
  next
    case False
    hence c ∈ s using insert.premis 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.premis(2) insert.premis(3) ⟨c ∈ s⟩
normalize-sort-removed-elem-irrelevant sort-ex-def insert-subset)
qed
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-equiv-normalize-sort:
  assumes finite s
  assumes sort-ex cs s
  shows sort-equiv cs s (normalize-sort cs s)
proof (intro sort-equivI)
  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  $c' \in \text{normalize-sort } cs \ s$  and  $\text{class-les } cs \ c' \ c2$ 
      using  $\langle c2 \in s \rangle \text{ normalize-ex-bound } \text{assms}$  by metis
    then show ?thesis using  $\text{class-les-def}$  by metis
  qed
qed
qed

lemma normalize-sort-eq-imp-sort-eqv:  $\text{sort-ex } cs \ s1 \implies \text{sort-ex } cs \ s2 \implies \text{finite } s1 \implies \text{finite } s2$ 
 $\implies \text{normalize-sort } cs \ s1 = \text{normalize-sort } cs \ s2$ 
 $\implies \text{sort-eqv } cs \ s1 \ s2$ 
by (metis sort-eqv-sym sort-eqv-trans wf-subclass-loc.sort-eqv-normalize-sort wf-subclass-loc-axioms)

lemma class-leq  $cs \ c1 \ c2 \longleftrightarrow \text{class-les } cs \ c1 \ c2 \vee (c1=c2 \wedge \text{class-ex } cs \ c1)$ 
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  $\text{sort-ex } cs \ s1 \ \text{sort-ex } cs \ s2 \ \text{sort-eqv } cs \ s1 \ s2$ 
  shows  $\text{normalize-sort } cs \ s1 = \text{normalize-sort } cs \ s2$ 
proof (rule ccontr)
  have  $\text{sort-leq } cs \ s1 \ s2 \ \text{sort-leq } cs \ s2 \ s1$ 
    using assms(3) by (auto simp add: sort-eqv-def)

  assume  $\text{normalize-sort } cs \ s1 \neq \text{normalize-sort } cs \ s2$ 
  hence  $\neg \text{normalize-sort } cs \ s1 \subseteq \text{normalize-sort } cs \ s2 \vee$ 
     $\neg \text{normalize-sort } cs \ s2 \subseteq \text{normalize-sort } cs \ s1$ 
    by simp
  from this consider  $\neg \text{normalize-sort } cs \ s1 \subseteq \text{normalize-sort } cs \ s2$ 
    |  $\text{normalize-sort } cs \ s1 \subseteq \text{normalize-sort } cs \ s2$ 
    |  $\neg \text{normalize-sort } cs \ s2 \subseteq \text{normalize-sort } cs \ s1$ 
    by blast
  thus False
proof cases
  case 1
    from this obtain c where  $c: c \in \text{normalize-sort } cs \ s1 \ c \notin \text{normalize-sort } cs \ s2$ 
      by blast
    from this obtain c' where  $c': c' \in \text{normalize-sort } cs \ s2 \ \text{class-les } cs \ c' \ c$ 
      by (smt  $\langle \text{sort-leq } cs \ s1 \ s2 \rangle \langle \text{sort-leq } cs \ s2 \ s1 \rangle \text{class-les-def mem-Collect-eq}$ 
        normalize-sort-def
         $\text{sort-leq-def wf-subclass-loc.class-leq-antisym wf-subclass-loc.class-leq-trans}$ 
        wf-subclass-loc-axioms)
    then show ?thesis
  proof (cases  $c' \in \text{normalize-sort } cs \ s1$ )
    case True
      hence  $c \notin \text{normalize-sort } cs \ s1$ 
        using  $c \ c'$  by (auto simp add: normalize-sort-def)
      then show ?thesis using  $c(1)$  by simp
    next
  end
next

```

case *False*
from *False* *c'* **obtain** *c''* **where** *c''*: $c'' \in \text{normalize-sort } cs \ s1 \ \text{class-les } cs \ c''$
c'
by (*smt* $\langle \text{sort-leq } cs \ s1 \ s2 \rangle \ \langle \text{sort-leq } cs \ s2 \ s1 \rangle \ \text{class-les-def } \text{mem-Collect-eq}$
normalize-sort-def
 $\text{sort-leq-def } \text{wf-subclass-loc.class-leq-antisym } \text{wf-subclass-loc.class-leq-trans}$
wf-subclass-loc-axioms)
hence $\text{class-les } cs \ c'' \ c$
using *c'*(2) *class-les-trans* **by** *blast*
hence $c \notin \text{normalize-sort } cs \ s1$
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*: $c \in \text{normalize-sort } cs \ s2 \ c \notin \text{normalize-sort } cs \ s1$
by *blast*
from *this* **obtain** *c'* **where** *c'*: $c' \in \text{normalize-sort } cs \ s1 \ \text{class-les } cs \ c' \ c$
by (*smt* $\langle \text{sort-leq } cs \ s1 \ s2 \rangle \ \langle \text{sort-leq } cs \ s2 \ s1 \rangle \ \text{class-les-def } \text{mem-Collect-eq}$
normalize-sort-def
 $\text{sort-leq-def } \text{wf-subclass-loc.class-leq-antisym } \text{wf-subclass-loc.class-leq-trans}$
wf-subclass-loc-axioms)
then show *?thesis*
proof(*cases* $c' \in \text{normalize-sort } cs \ s2$)
case *True*
hence $c \notin \text{normalize-sort } cs \ s2$
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''*: $c'' \in \text{normalize-sort } cs \ s2 \ \text{class-les } cs \ c'' \ c'$
by (*smt* $\langle \text{sort-leq } cs \ s1 \ s2 \rangle \ \langle \text{sort-leq } cs \ s2 \ s1 \rangle \ \text{class-les-def } \text{mem-Collect-eq}$
normalize-sort-def
 $\text{sort-leq-def } \text{wf-subclass-loc.class-leq-antisym } \text{wf-subclass-loc.class-leq-trans}$
wf-subclass-loc-axioms)
hence $\text{class-les } cs \ c'' \ c$
using *c'*(2) *class-les-trans* **by** *blast*
hence $c \notin \text{normalize-sort } cs \ s2$
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 $\text{sort-eqv } cs \ s1 \ s2 \longleftrightarrow \text{normalize-sort } cs \ s1 = \text{normalize-sort } cs \ s2$

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

end

lemma *tcsigs-sorts-defined*: $wf\text{-osig } oss \implies$
 $(\forall ars \in \text{ran } (tcsigs \text{ } oss) . \forall ss \in \text{ran } ars . \forall s \in \text{set } ss . \text{sort-ex } (subclass \text{ } oss) \text{ } s)$
by (*cases oss*) (*simp add: wf-sort-def all-normalized-and-ex-tcsigs-def*)

lemma *osig-subclass-loc*: $wf\text{-osig } oss \implies wf\text{-subclass-loc } (subclass \text{ } oss)$
using *wf-subclass-loc.intro* **by** (*cases oss*) *simp*

lemma *wf-osig-imp-wf-subclass-loc*: $wf\text{-osig } oss \implies wf\text{-subclass-loc } (subclass \text{ } oss)$
by (*cases oss*) (*simp add: wf-subclass-loc-def*)

lemma *has-sort-Tv-imp-sort-leq*: $has\text{-sort } oss \text{ } (Tv \text{ } idn \text{ } S) \text{ } S' \implies \text{sort-leq } (subclass \text{ } oss) \text{ } S \text{ } S'$
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* :: $term \Rightarrow typ \text{ } option$ **where**
 $dest\text{-type } (Ct \text{ } nc \text{ } (Ty \text{ } nt \text{ } [ty])) =$
 $(if \text{ } nc = STR \text{ } "Pure.type" \wedge nt = STR \text{ } "Pure.type" \text{ then } Some \text{ } ty \text{ else } None)$
 $| \text{ } dest\text{-type } t = None$

definition *type-map* $f \text{ } t = \text{map-option } (\lambda ty . \text{mk-type } (f \text{ } ty)) \text{ } (dest\text{-type } t)$

consts *unsuffix* :: $name \Rightarrow name \Rightarrow name \text{ } option$

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

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

definition *mk-of-sort* $ty \text{ } S == \text{map } (\lambda c . \text{mk-of-class } ty \text{ } c) \text{ } S$

end

5 Wellformed Signature and Theory

theory *Theory*

imports *Term Sorts SortConstants*

begin

fun *typ-ok-sig* :: *signature* \Rightarrow *typ* \Rightarrow *bool* **where**
 typ-ok-sig Σ (*Ty* *c* *Ts*) = (*case type-arity* Σ *c* of
 None \Rightarrow *False*
 | *Some ar* \Rightarrow *length* *Ts* = *ar* \wedge *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* \Longrightarrow *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* \Longrightarrow *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* \Rightarrow *term* \Rightarrow *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 \Rightarrow *False*
 | *Some ty* \Rightarrow *typ-ok-sig* Σ *T* \wedge *tinstT* *T* *ty*)
| *term-ok'* Σ (*t* \$ *u*) \longleftrightarrow *term-ok'* Σ *t* \wedge *term-ok'* Σ *u*
| *term-ok'* Σ (*Abs* *T* *t*) \longleftrightarrow *typ-ok-sig* Σ *T* \wedge *term-ok'* Σ *t*

lemma *term-ok'-imp-wf-term*: *term-ok'* Σ *t* \Longrightarrow *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* \Longrightarrow *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 \Longrightarrow *wf-term* Σ *t* \Longrightarrow *list-all* (*typ-ok-sig* Σ) *Ts*

\Longrightarrow *typ-of1* *Ts* *t* = *Some ty* \Longrightarrow *typ-ok-sig* Σ *ty*

```

proof (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.prem1 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.prem1(4) by (meson typ-of1-arg-ty)
    ultimately have typ-ok-sig Σ (U → T)
      using 5.IH(2) 5.prem1(1) 5.prem1(2) 5.prem1(3) term-ok'.sims(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.*

$\Theta = ((cto, tao, sorts), axioms) \implies P$
 $\implies P$

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 \langle cases Θ rule: theory-full-exhaust \rangle)
(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* $\Theta \implies$ *is-std-sig* (sig Θ)

by (cases Θ rule: theory-full-exhaust) simp

lemma *wf-theory-imp-wf-sig*: *wf-theory* $\Theta \implies$ *wf-sig* (sig Θ)

by (cases Θ rule: theory-full-exhaust) simp

lemma

term-ok-imp-typ-ok:

wf-theory thy \implies *term-ok* thy t \implies *typ-of* t = Some ty \implies *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 \implies $A \in$ *axioms* thy \implies *term-ok* thy A

using *wt-term-def* **by** (cases thy rule: theory-full-exhaust) simp

lemma *axioms-typ-of-propT*: *wf-theory* thy \implies $A \in$ *axioms* thy \implies *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* $\Theta \implies$ *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*: $\text{term-ok } \Theta \text{ (mk-eq } s \ t) \implies \text{term-ok } \Theta \ s \wedge \text{term-ok } \Theta \ t$
using *term-ok'.simps(4) wt-term-def typ-of-def* **by** (*auto simp add: bind-eq-Some-conv*)
lemma *term-ok-app-eqD*: $\text{term-ok } \Theta \ (s \ \$ \ t) \implies \text{term-ok } \Theta \ s \wedge \text{term-ok } \Theta \ t$
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*:
 $\text{wf-sig } \Sigma \implies \text{wf-type } \Sigma \text{ (Ty } n \ Ts) \implies \text{tcsigs (osig } \Sigma) \ n \neq \text{None}$
by (*cases } \Sigma) \text{ (auto split: option.splits)*

lemma *term-ok-eta-expand*:
assumes *wf-theory } \Theta \ \text{term-ok } \Theta \ f \ \text{typ-of } f = \text{Some } (\tau \rightarrow \tau') \ \text{typ-ok } \Theta \ \tau*
shows $\text{term-ok } \Theta \ (\text{Abs } \tau \ (f \ \$ \ Bv \ 0))$
using *assms typ-of-eta-expand* **by** (*auto simp add: wt-term-def*)

lemma *term-ok'-incr-bv*: $\text{term-ok}' \Sigma \ t \implies \text{term-ok}' \Sigma \ (\text{incr-bv } \text{inc } \text{lev } t)$
by (*induction inc lev t rule: incr-bv.induct*) *auto*

lemma *term-ok'-subst-bv2*: $\text{term-ok}' \Sigma \ s \implies \text{term-ok}' \Sigma \ u \implies \text{term-ok}' \Sigma \ (\text{subst-bv2 } s \ \text{lev } u)$
by (*induction s lev u rule: subst-bv2.induct*) (*auto simp add: term-ok'-incr-bv*)

lemma *term-ok'-subst-bv*: $\text{term-ok}' \Sigma \ (\text{Abs } T \ t) \implies \text{term-ok}' \Sigma \ (\text{subst-bv } (\text{Fv } x \ T) \ t)$
by (*simp add: substn-subst-0' term-ok'-subst-bv2*)

lemma *term-ok-subst-bv*: $\text{term-ok } \Theta \ (\text{Abs } T \ t) \implies \text{term-ok } \Theta \ (\text{subst-bv } (\text{Fv } x \ T) \ t)$
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*: $\text{term-ok } \Theta \ (\text{Abs } T \ t) \implies \text{term-ok } \Theta \ (\text{subst-bv2 } t \ 0 \ (\text{Fv } x \ T))$
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 } \Sigma \ \text{wf-type } \Sigma \ T*
shows $\text{has-sort (osig } \Sigma) \ T \ \text{full-sort}$
proof(*cases } T)*
case (*Ty } n \ Ts)*
obtain *cl tcs* **where** $\text{cltcs: osig } \Sigma = (cl, \ tcs)$
by *fastforce*
obtain *mgd* **where** $\text{mgd: tcsigs (osig } \Sigma) \ n = \text{Some } \text{mgd}$
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  $\Sigma$ ) (auto simp add: sort-leq-def split: prod.splits)
qed

```

```

lemma typ-Fv-of-full-sort[simp]:
  wf-theory  $\Theta \implies$  term-ok  $\Theta$  (Fv  $v$   $T$ )  $\implies$  has-sort (osig (sig  $\Theta$ ))  $T$  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  $\times$  sort)  $\times$  typ) list  $\Rightarrow$  typ  $\Rightarrow$  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 ( $\lambda x . x = (idn, S)$ ) 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  $l \implies$  tvsT  $T \subseteq$  set  $l$ 
   $\implies$  tsubstT  $T$   $\varrho =$  subst-typ (map ( $\lambda(x,y).((x,y), \varrho x y)$ )  $l$ )  $T$ 

```

```

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 ( $\lambda(x,y).((x,y), \varrho x y)$ )  $l$ ))

```

```

    by (simp add: case-prod-beta map-idI)

```

```

    hence  $el$ :  $((idn,S), \varrho idn S) \in$  set (map ( $\lambda a.$  case a of  $(x, y) \Rightarrow ((x, y), \varrho x y)$ ))

```

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

lemma *subst-typ-simulates-tsubstT-gen*: $tsubstT\ T\ \varrho$
 $=\ subst\text{-}typ\ (map\ (\lambda(x,y).((x,y),\ \varrho\ x\ y))\ (SOME\ l.\ distinct\ l\ \wedge\ tvsT\ T\ \subseteq\ set\ l))$
 T

proof(*rule someI2-ex*)

show $\exists a.\ distinct\ a\ \wedge\ tvsT\ T\ \subseteq\ set\ a$

using *finite-tvsT finite-distinct-list*

by (*metis order-refl*)

next

fix l **assume** $l:\ distinct\ l\ \wedge\ tvsT\ T\ \subseteq\ set\ l$

then show $tsubstT\ T\ \varrho = subst\text{-}typ\ (map\ (\lambda a.\ case\ a\ of\ (x,\ y)\ \Rightarrow\ ((x,\ y),\ \varrho\ x\ y))\ l)\ T$

using *subst-typ-simulates-tsubstT-gen'* **by** *blast*

qed

corollary *subst-typ-simulates-tsubstT*: $tsubstT\ T\ \varrho$
 $=\ subst\text{-}typ\ (map\ (\lambda(x,y).((x,y),\ \varrho\ x\ y))\ (SOME\ l.\ distinct\ l\ \wedge\ set\ l = tvsT\ T))$
 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*: $subst\text{-}typ\ insts\ T$
 $=\ tsubstT\ T\ (\lambda idn\ S.\ the\text{-}default\ (Tv\ idn\ S)\ (lookup\ (\lambda x.\ x=(idn,\ S))\ insts))$
by (*induction T*) *auto*

lemma *subst-typ-comp*:

$subst\text{-}typ\ inst1\ (subst\text{-}typ\ inst2\ T) = subst\text{-}typ\ (map\ (apsnd\ (subst\text{-}typ\ inst1))$
 $inst2\ @\ inst1)\ T$

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*: $subst\text{-}typ\ insts\ T = subst\text{-}typ\ (AList.clearjunk$
 $insts)\ T$

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.prem(1-3)])
      using lookup-eq-order-irrelevant by (metis Fv.prem(4) Fv.prem(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* \subseteq *set lty*
assumes *lt-assms*: *distinct lt fv (tsubst t ρ ty)* \subseteq *set lt*
shows *subst (tsubst t ρ ty) ρ t*
 $=$ *subst-type-term (map ($\lambda(x,y).(x,y), \rho$ ty x y)) lty) (map ($\lambda(x,y).(x,y), \rho$ t x y)) lt) t*
proof–
let *?lty* $=$ *map ($\lambda(x,y).(x,y), \rho$ ty x y) lty*

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

let *?lt* $=$ *map ($\lambda(x,y).(x,y), \rho$ t 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 ρ ty*
have *el*: (*(idn, ?T), ρ t idn ?T*) \in *set (map ($\lambda(x,y).(x,y), \rho$ t x y)) lt*
using *Fv* **by** *auto*
have *d*: *distinct (map fst (map ($\lambda(x,y).(x,y), \rho$ t x y) lt))*
using *Fv* **by** (*simp add: case-prod-beta map-idI*)
show *?case* **using** *Fv.prem*s *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 ρ ty) ρ t*
 $=$ *subst-type-term (map ($\lambda(x,y).(x,y), \rho$ ty x y)) (SOME lty . distinct lty \wedge tvs t = set lty)*
(map ($\lambda(x,y).(x,y), \rho$ t x y)) (SOME lt . distinct lt \wedge fv (tsubst t ρ ty) = 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* \equiv *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* \implies *tvs t* \subseteq *set pairs*

\implies $tsubst\ t\ \varrho = subst\text{-}typ'\ (map\ (\lambda(x,y).((x,y),\ \varrho\ x\ y))\ pairs)\ t$
by (*induction t arbitrary: pairs ϱ*)
(auto simp add: subst-typ-simulates-tsubstT-gen')

lemma *subst-typ'-simulates-tsubst-gen: tsubst t ϱ*
 $= subst\text{-}typ'\ (map\ (\lambda(x,y).((x,y),\ \varrho\ x\ y))\ (SOME\ l.\ distinct\ l \wedge tvs\ t \subseteq set\ l))\ t$
proof(*rule someI2-ex*)
show $\exists a.\ distinct\ a \wedge tvs\ t \subseteq set\ a$
using *finite-tvs finite-distinct-list*
by (*metis order-refl*)
next
fix l **assume** $l:\ distinct\ l \wedge tvs\ t \subseteq set\ l$

then show $tsubst\ t\ \varrho = subst\text{-}typ'\ (map\ (\lambda a.\ case\ a\ of\ (x,\ y) \Rightarrow ((x,\ y),\ \varrho\ x\ y))\ l)\ t$
using *subst-typ'-simulates-tsubst-gen'* **by** *blast*
qed

lemma *tsubst-simulates-subst-typ': subst-typ' insts T*
 $= tsubst\ T\ (\lambda idn\ S.\ the\ default\ (Tv\ idn\ S)\ (lookup\ (\lambda x.\ x=(idn,\ S))\ insts))$
by (*induction T*) (*auto simp add: tsubstT-simulates-subst-typ*)

lemma *subst-type-add-degenerate-instance:*
 $(idx,s) \notin set\ (map\ fst\ insts) \implies subst\text{-}typ\ insts\ T = subst\text{-}typ\ (((idx,s),\ Tv\ idx\ s)\#insts)\ T$
by (*induction T*) (*auto simp add: lookup-eq-key-not-present*)

lemma *subst-typ'-add-degenerate-instance:*
 $(idx,s) \notin set\ (map\ fst\ insts) \implies subst\text{-}typ'\ insts\ t = subst\text{-}typ'\ (((idx,s),\ Tv\ idx\ s)\#insts)\ t$
by (*induction t*) (*auto simp add: subst-type-add-degenerate-instance*)

lemma *subst-typ'-comp:*
 $subst\text{-}typ'\ inst1\ (subst\text{-}typ'\ inst2\ t) = subst\text{-}typ'\ (map\ (apsnd\ (subst\text{-}typ\ inst1))\ inst2\ @\ inst1)\ t$
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* :: $((variable\ * typ) * term)\ list \Rightarrow term \Rightarrow term$ **where**
 $subst\text{-}term\ insts\ (Ct\ c\ T) = Ct\ c\ T$
 $| subst\text{-}term\ insts\ (Fv\ idn\ T) = the\ default\ (Fv\ idn\ T)\ (lookup\ (\lambda x.\ x=(idn,\ T))\ insts)$
 $| subst\text{-}term\ -\ (Bv\ n) = Bv\ n$

| $\text{subst-term insts (Abs } T \ t) = \text{Abs } T \ (\text{subst-term insts } t)$
| $\text{subst-term insts } (t \ \$ \ u) = \text{subst-term insts } t \ \$ \ \text{subst-term insts } u$

lemma *subst-term-empty-no-change*[simp]: $\text{subst-term } [] \ t = t$
by (*induction t*) *auto*

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

lemma *subst-type-term-split-levels*:
 $\text{subst-type-term instT insts } t = \text{subst-term insts } (\text{subst-typr' instT } t)$
by (*induction t*) (*auto simp add: Let-def*)

lemma *subst-typr-stepwise*:
assumes *distinct* (*map fst instT*)
assumes $\bigwedge x . x \in (\bigcup t \in \text{snd } ' \text{set instT} . \text{tvsT } t) \implies x \notin \text{fst } ' \text{set instT}$
shows $\text{subst-typr instT } T = \text{fold } (\lambda \text{single } \text{acc} . \text{subst-typr } [\text{single}] \ \text{acc}) \ \text{instT } T$
using *assms* **proof** (*induction instT T* *rule: subst-typr.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 $\text{subst-typr inst } (\text{Ty } a \ Ts) = \text{fold } (\lambda \text{single} . \text{subst-typr } [\text{single}]) \ \text{inst } (\text{Ty } a \ Ts)$
by *simp*
moreover have $\text{subst-typr inst } T = \text{fold } (\lambda \text{single} . \text{subst-typr } [\text{single}]) \ \text{inst } T$
using *Cons 1* **by** *simp*
moreover have $\text{fold } (\lambda \text{single} . \text{subst-typr } [\text{single}]) \ \text{inst } (\text{Ty } a \ (T \ # \ Ts))$
 $= \text{Ty } a \ (\text{map } (\text{fold } (\lambda \text{single} . \text{subst-typr } [\text{single}]) \ \text{inst}) \ (T \ # \ Ts))$
proof (*induction inst* *rule: rev-induct*)
case *Nil*
then show *?case* **by** *simp*
next
case (*snoc x xs*)
hence $\text{fold } (\lambda \text{single} . \text{subst-typr } [\text{single}]) \ (xs \ @ \ [x]) \ (\text{Ty } a \ (T \ # \ Ts)) =$
 $\text{Ty } a \ (\text{map } (\text{subst-typr } [x]) \ (\text{map } (\text{fold } (\lambda \text{single} . \text{subst-typr } [\text{single}]) \ xs) \ (T \ # \ Ts)))$
by *simp*
then show *?case* **by** *simp*
qed
ultimately show *?case*
using *Cons.premis(1)* *Cons.premis(2)* *local.Cons(4)* **by** *auto*

qed
next
case (2 *inst idn S*)
then show ?*case*
proof (*cases lookup* ($\lambda x . x = (idn, S)$) (*inst*))
case *None*
hence *fst p* $\neq (idn, S)$ **if** $p \in set\ inst$ **for** *p* **using** *that* **by** (*auto simp add: lookup-None-iff*)
hence *subst-typ* [*p*] (*Tv idn S*) = *Tv idn S* **if** $p \in set\ inst$ **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*: $((idn, S), a) \in set\ inst$ **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 $(idn, S) \notin set\ (map\ fst\ fs)$ **and** $(idn, S) \notin set\ (map\ fst\ bs)$ **using** 2 **by** *simp-all*

hence *fst p* $\neq (idn, S)$ **if** $p \in set\ fs$ **for** *p*
using *that* **by** *force*
hence *id-subst-fs*: *subst-typ* [*p*] (*Tv idn S*) = *Tv idn S* **if** $p \in set\ fs$ **for** *p*
using *that* **by** (*cases p*) *fastforce*
hence *fs-step*: *fold* ($\lambda single. subst-typ\ [single]$) *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*: $set\ bs \subseteq set\ inst$ **using** *split* **by** *auto*
hence $x \notin fst\ 'set\ bs$
if $x \in \bigcup (tvsT\ 'snd\ 'set\ bs)$ **for** *x*
using 2 *that* *split* **by** (*auto simp add: image-iff*)

have $v \notin fst\ 'set\ bs$ **if** $v \in tvsT\ a$ **for** *v*
using *that* 2 *elem bs-sub* **by** (*fastforce simp add: image-iff*)

hence *id-subst-bs*: *subst-typ* [*p*] *a* = *a* **if** $p \in set\ bs$ **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* ($\lambda single. subst-typ\ [single]$) *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 $\bigwedge y . y \in (\bigcup t \in \text{snd} \text{ ` } \text{set} (x\#xs) . \text{tvs}T t) \implies y \notin \text{fst} \text{ ` } (\text{set} (x\#xs))$
 shows *subst-typ* (x#xs) T = *subst-typ* xs (*subst-typ* [x] T)
proof–
 have *subst-typ* (x#xs) T = fold ($\lambda \text{single} . \text{subst-typ} [\text{single}]$) (x#xs) T
 using *assms* *subst-typ-stepwise* by blast
 also have ... = fold ($\lambda \text{single} . \text{subst-typ} [\text{single}]$) 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 $\bigwedge y . y \in (\bigcup t \in \text{snd} \text{ ` } (\text{set} (xs @ [x])) . \text{tvs}T t) \implies y \notin \text{fst} \text{ ` } (\text{set} (xs @ [x]))$
 shows *subst-typ* (xs @ [x]) T = *subst-typ* [x] (*subst-typ* xs T)
proof–
 have *subst-typ* (xs @ [x]) T = fold ($\lambda \text{single} . \text{subst-typ} [\text{single}]$) (xs@[x]) T
 using *assms* *subst-typ-stepwise* by blast
 also have ... = *subst-typ* [x] (fold ($\lambda \text{single} . \text{subst-typ} [\text{single}]$) 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 $\bigwedge x . x \in (\bigcup t \in \text{snd} \text{ ` } (\text{set} \text{inst}T) . \text{tvs}T t) \implies x \notin \text{fst} \text{ ` } (\text{set} \text{inst}T)$
 shows *subst-typ'* *instT* t = fold ($\lambda \text{single} \text{acc} . \text{subst-typ}' [\text{single}] \text{acc}$) *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 $\bigwedge x . x \in (\bigcup t \in \text{snd} \text{ ` } (set \text{ } insts) . \text{fv } t) \implies x \notin \text{fst} \text{ ` } (set \text{ } insts)$
 shows *subst-term* *insts* *t* = fold ($\lambda \text{single } acc . \text{subst-term} [\text{single}] acc$) *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*: $x \notin \text{fst} \text{ ` } set \text{ } insts$ **if** $x \in \bigcup (\text{fv} \text{ ` } \text{snd} \text{ ` } set \text{ } insts)$ **for** *x*
using *insts-def snoc* **that** **by** *blast*
from *Fv* **show** *?case*

proof (*cases lookup* ($\lambda x . x = (idn, T)$) *insts*)
 case *None*
 hence *fst p* $\neq (idn, T)$ **if** $p \in set \text{ } insts$ **for** *p* **using** *that* **by** (*auto simp add: lookup-None-iff*)
 hence *subst-term* [*p*] (*Fv idn T*) = *Fv idn T* **if** $p \in set \text{ } insts$ **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*: $((idn, T), a) \in set \text{ } insts$ **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 $(idn, T) \notin set \text{ } (map \text{ } fst \text{ } fs)$ **and** $(idn, T) \notin set \text{ } (map \text{ } fst \text{ } bs)$ **using**
insts-thm1 **by** *simp-all*

hence *fst p* $\sim = (idn, T)$ **if** $p \in set \text{ } fs$ **for** *p*
using *that* **by** *force*
 hence *id-subst-fs*: *subst-term* [*p*] (*Fv idn T*) = *Fv idn T* **if** $p \in set \text{ } fs$ **for** *p*
using *that* **by** (*cases p*) *fastforce*
 hence *fs-step*: fold ($\lambda \text{single} . \text{subst-term} [\text{single}]$) *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* $bs \subseteq$ *set insts* **using** *split* **by** *auto*
hence $x \notin fst\ 'set\ bs$
if $x \in \bigcup (fv\ 'snd\ 'set\ bs)$ **for** x
using *insts-thm2* **that** *split* **by** $(auto\ simp\ add:\ image-iff)$

have $v \notin fst\ 'set\ bs$ **if** $v \in fv\ a$ **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** $p \in set\ bs$ **for** p
using *that* **by** $(cases\ p,\ induction\ a)\ force+$
hence *bs-step*: *fold* $(\lambda single.\ subst-term\ [single])\ bs\ a = a$
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 $\bigwedge y . y \in (\bigcup t \in snd\ ' (set\ (xs\ @\ [x])) . fv\ t) \implies y \notin fst\ ' (set\ (xs\ @\ [x]))$
shows *subst-term* $(xs\ @\ [x])\ t = subst-term\ [x]\ (subst-term\ xs\ t)$
proof-
have *subst-term* $(xs\ @\ [x])\ t = fold\ (\lambda single . subst-term\ [single])\ (xs@[x])\ t$
using *assms subst-term-stepwise* **by** *blast*
also **have** $\dots = subst-term\ [x]\ (fold\ (\lambda single . subst-term\ [single])\ xs\ t)$
by *simp*
also **have** $\dots = subst-term\ [x]\ (subst-term\ xs\ t)$
using *assms subst-term-stepwise* **by** *simp*
finally **show** *?thesis* .
qed

corollary *subst-type-term-stepwise*:
assumes *distinct* $(map\ fst\ instT)$
assumes $\bigwedge x . x \in (\bigcup T \in snd\ ' (set\ instT) . tvsT\ T) \implies x \notin fst\ ' (set\ instT)$
assumes *distinct* $(map\ fst\ insts)$
assumes $\bigwedge x . x \in (\bigcup t \in snd\ ' (set\ insts) . fv\ t) \implies x \notin fst\ ' (set\ insts)$
shows *subst-type-term* $instT\ insts\ t$
 $= fold\ (\lambda single . subst-term\ [single])\ insts\ (fold\ (\lambda single . subst-typ'\ [single])\ instT\ t)$
using *assms subst-typ'-stepwise subst-term-stepwise subst-type-term-split-levels*
by *auto*

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

l) *auto*
lemma *distinct-kv-list*: *distinct l* \implies *distinct (map ($\lambda x. (x, f x)$) l)* **by** (*induction l*) *auto*

lemma *subst-subst-term*:

assumes *distinct l* **and** *fv t \subseteq set l*
shows *subst t ϱ = subst-term (map ($\lambda x. (x, case-prod \varrho x)$) l) t*
using *assms* **proof** (*induction t arbitrary: l*)
case (*Fv idn T*)
then show *?case*
proof (*cases (idn, T) \in set l*)
case *True*
hence *((idn, T), ϱ idn T) \in set (map ($\lambda x. (x, case-prod \varrho x)$) l)* **by** *auto*
moreover have *distinct (map fst (map ($\lambda x. (x, case-prod \varrho x)$) l))*
using *Fv(1)* **by** (*induction l*) *auto*
ultimately have *(lookup ($\lambda x. x = (idn, T)$) (map ($\lambda x. (x, case x of (x, xa) \Rightarrow \varrho x xa$)) 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 ($\lambda((idn, T), t) f x y. if x=idn \wedge y=T$ then t else f x y) l Fv)*
using *assms* **proof** (*induction t*)
case (*Fv idn T*)
then show *?case*
proof (*cases lookup ($\lambda x. x = (idn, T)$) l*)
case *None*
hence *(idn, T) \notin set (map fst l)*
by (*metis (full-types) lookup-None-iff*)

hence *(fold ($\lambda((idn, T), t) f x y. if x=idn \wedge y=T$ 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) \in 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 $(idn, T) \notin \text{set } (\text{map fst fs})$ and *not-in-bs*: $(idn, T) \notin \text{set } (\text{map fst bs})$
 using *Fv* by *simp-all*

hence $\text{fst } p \sim = (idn, T)$ if $p \in \text{set } fs$ for p
 using that by *force*
 hence *fs-step*: $(\text{fold } (\lambda((idn, T), t) f x y. \text{if } x=idn \wedge y=T \text{ then } t \text{ else } f x y) fs$
Fv) $idn T = Fv idn T$
 by (*induction fs rule: rev-induct*) (*fastforce split: if-splits prod.splits*) $+$

have *bs-sub*: $\text{set } bs \subseteq \text{set } l$ using *split* by *auto*

have $\text{fst } p \sim = (idn, T)$ if $p \in \text{set } bs$ for p
 using that *not-in-bs* by *force*
 hence *bs-step*: $(\text{fold } (\lambda((idn, T), t) f x y. \text{if } x=idn \wedge y=T \text{ then } t \text{ else } f x y) bs$
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* $\notin \text{fst } ' \text{tvsT } \tau$
shows *subst-typ* $[[(\text{fresh-idn}, S), \tau 2)] (\text{subst-typ } [((idn, S), Tv \text{fresh-idn } S)] \tau)$
 $= \text{subst-typ } [((idn, S), \tau 2)] \tau$
 using *assms* by (*induction* τ) *auto*

lemma *subst-typ-combine*:

assumes $\text{length } \text{fresh-idns} = \text{length } \text{insts}$
assumes *distinct fresh-idns*
assumes *distinct (map fst insts)*
assumes $\forall idn \in \text{set } \text{fresh-idns} . idn \notin \text{fst } ' (\text{tvsT } \tau \cup (\bigcup ty \in \text{snd } ' \text{set } \text{insts} .$
 $(\text{tvsT } ty))$
 $\cup (\text{fst } ' \text{set } \text{insts}))$
shows *subst-typ insts* τ
 $= \text{subst-typ } (\text{zip } (\text{zip } \text{fresh-idns } (\text{map } \text{snd } (\text{map } \text{fst } \text{insts}))) (\text{map } \text{snd } \text{insts}))$
 $(\text{subst-typ } (\text{zip } (\text{map } \text{fst } \text{insts}) (\text{map2 } Tv \text{ fresh-idns } (\text{map } \text{snd } (\text{map } \text{fst } \text{insts}))))$
 $\tau)$
 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* $(\lambda x. x = (idn, S)) \text{inst}$)
case *None*
 hence $((idn, S)) \notin \text{fst } ' \text{set } \text{inst}$

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

have (*idn, S*) \notin *set* (*zip fresh-idns* (*map* (*snd* \circ *fst*) *inst*))
 using 2 *set- zip-leftD* by *fastforce*
 hence (*lookup* ($\lambda x. x = (idn, S)$)
 (*zip* (*zip fresh-idns* (*map* (*snd* \circ *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 ($\bigwedge idx. as ! idx = ((idn, S), ty) \implies idx < length as \implies thesis$)
 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* \circ *fst*) *inst*))) ! *idx*
 = (*idn, S*), *Tv fresh-idn S*)
 by *auto*
 from *this idx(2)* have (*(idn, S), Tv fresh-idn S*) \in *set*
 (*zip* (*map fst inst*) (*map2 Tv fresh-idns* (*map* (*snd* \circ *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* ($\lambda x. x = (idn, S)$)
 (*zip* (*map fst inst*) (*map2 Tv fresh-idns* (*map* (*snd* \circ *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*)
 \in *set* (*zip* (*zip fresh-idns* (*map* (*snd* \circ *fst*) *inst*)) (*map snd inst*))
 using *in-set-conv-nth* by *fastforce*
 hence 2: (*lookup* ($\lambda x. x = (fresh-idn, S)$)
 (*zip* (*zip fresh-idns* (*map* (*snd* \circ *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 $\forall idn \in \text{set fresh-idns} . idn \notin \text{fst} \text{ ` } (tvsT \ \tau \cup (\bigcup ty \in \text{snd} \text{ ` } \text{set insts} .$
 $(tvsT \ ty))$
 $\cup (\text{fst} \text{ ` } \text{set insts}))$

shows *subst-typ insts* τ

$= \text{fold} (\lambda \text{single acc} . \text{subst-typ} [\text{single}] \text{ acc}) (\text{zip} (\text{zip fresh-idns} (\text{map snd} (\text{map}$
 $\text{fst insts}))) (\text{map snd insts}))$

$(\text{fold} (\lambda \text{single acc} . \text{subst-typ} [\text{single}] \text{ acc}) (\text{zip} (\text{map fst insts}) (\text{map2 Tv}$
 $\text{fresh-idns} (\text{map snd} (\text{map fst insts})))) \ \tau$

proof–

have *s1*: $\text{fst} \text{ ` } \text{set} (\text{zip} (\text{map fst insts}) (\text{map2 Tv fresh-idns} (\text{map snd} (\text{map fst}$
 $\text{insts}))))$

$= \text{fst} \text{ ` } \text{set insts}$

proof–

have $\text{fst} \text{ ` } \text{set} (\text{zip} (\text{map fst insts}) (\text{map2 Tv fresh-idns} (\text{map snd} (\text{map fst}$
 $\text{insts}))))$

$= \text{set} (\text{map fst} (\text{zip} (\text{map fst insts}) (\text{map2 Tv fresh-idns} (\text{map snd} (\text{map fst}$
 $\text{insts}))))))$

by *auto*

also have $\dots = \text{set} (\text{map fst insts})$ **using** *map-fst-zip assms(1)* **by** *auto*

also have $\dots = \text{fst} \text{ ` } \text{set insts}$ **by** *simp*

finally show *?thesis* .

qed

have $\text{snd} \text{ ` } \text{set} (\text{zip} (\text{map fst insts}) (\text{map2 Tv fresh-idns} (\text{map snd} (\text{map fst insts}))))$
 $= \text{set} (\text{map2 Tv fresh-idns} (\text{map snd} (\text{map fst insts})))$ **using** *map-snd-zip*
assms(1)

by (*metis (no-types, lifting) image-set length-map*)

hence $(\bigcup (tvsT \ \text{snd} \text{ ` } \text{set} (\text{zip} (\text{map fst insts}) (\text{map2 Tv fresh-idns} (\text{map snd}$
 $(\text{map fst insts}))))))$

$= (\bigcup (tvsT \ \text{set} (\text{map2 Tv fresh-idns} (\text{map snd} (\text{map fst insts}))))))$

by *simp*

from *assms(1)* **this have** *s2*:

$(\bigcup (tvsT \ \text{snd} \text{ ` } \text{set} (\text{zip} (\text{map fst insts}) (\text{map2 Tv fresh-idns} (\text{map snd} (\text{map}$
 $\text{fst insts}))))))$

$= (\text{set} (\text{zip fresh-idns} (\text{map snd} (\text{map fst insts}))))$

using *assms(1)* **by** (*induction fresh-idns insts rule: list-induct2*) *auto*

hence *s3*: $\bigcup (tvsT \ \text{snd} \text{ ` } \text{set} (\text{zip} (\text{map fst insts}$

$(\text{map2 Tv fresh-idns} (\text{map} (\text{snd} \circ \text{fst}) \text{ insts}))))$

$= \text{set} (\text{zip fresh-idns} (\text{map snd} (\text{map fst insts})))$ **by** *simp*

have $idn \notin \text{fst} \text{ ` } \text{fst} \text{ ` } \text{set insts}$ **if** $idn \in \text{set fresh-idns}$ **for** idn

using *that assms* **by** *auto*

hence $I: (idn, S) \notin fst \text{ ' set insts}$ **if** $idn \in set \text{ fresh-idns}$ **for** $idn S$
using *that assms* **by** (*metis fst-conv image-eqI*)

have $u1: (subst\text{-}typ (zip (map \text{fst insts}) (map2 \text{Tv fresh-idns} (map \text{snd} (map \text{fst insts})))) \tau)$
 $= fold (\lambda single \text{acc} . subst\text{-}typ [single] \text{acc}) (zip (map \text{fst insts}) (map2 \text{Tv fresh-idns} (map \text{snd} (map \text{fst insts})))) \tau$
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 $u2: subst\text{-}typ (zip (zip \text{fresh-idns} (map \text{snd} (map \text{fst insts}))) (map \text{snd insts}))$
 $(subst\text{-}typ (zip (map \text{fst insts}) (map2 \text{Tv fresh-idns} (map \text{snd} (map \text{fst insts})))) \tau)$
 $= fold (\lambda single \text{acc} . subst\text{-}typ [single] \text{acc}) (zip (zip \text{fresh-idns} (map \text{snd} (map \text{fst insts}))) (map \text{snd insts})) (map \text{snd insts})$
 $(subst\text{-}typ (zip (map \text{fst insts}) (map2 \text{Tv fresh-idns} (map \text{snd} (map \text{fst insts})))) \tau)$
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 $unfold: subst\text{-}typ (zip (zip \text{fresh-idns} (map \text{snd} (map \text{fst insts}))) (map \text{snd insts}))$
 $(subst\text{-}typ (zip (map \text{fst insts}) (map2 \text{Tv fresh-idns} (map \text{snd} (map \text{fst insts})))) \tau)$
 $= fold (\lambda single \text{acc} . subst\text{-}typ [single] \text{acc}) (zip (zip \text{fresh-idns} (map \text{snd} (map \text{fst insts}))) (map \text{snd insts})) (map \text{snd insts})$
 $(fold (\lambda single \text{acc} . subst\text{-}typ [single] \text{acc}) (zip (map \text{fst insts}) (map2 \text{Tv fresh-idns} (map \text{snd} (map \text{fst insts})))) \tau)$
by *simp*
show *?thesis* **using** *assms subst-typ-combine unfold* **by** *auto*
qed

lemma *subst-typ'-combine:*
assumes $length \text{ fresh-idns} = length \text{ insts}$
assumes $distinct \text{ fresh-idns}$
assumes $distinct (map \text{fst insts})$
assumes $\forall idn \in set \text{ fresh-idns} . idn \notin fst \text{ ' } (tvs \text{ } t \cup (\bigcup ty \in snd \text{ ' set insts} . (tvs \text{ } T ty)))$
 $\cup (fst \text{ ' set insts})$
shows $subst\text{-}typ' \text{ insts } t$
 $= subst\text{-}typ' (zip (zip \text{fresh-idns} (map \text{snd} (map \text{fst insts}))) (map \text{snd insts})) (map \text{snd insts})$
 $(subst\text{-}typ' (zip (map \text{fst insts}) (map2 \text{Tv fresh-idns} (map \text{snd} (map \text{fst insts})))) \tau)$
using *assms* **proof** (*induction t arbitrary: fresh-idns insts*)

```

case (Abs T t)
moreover have  $tvs\ t \subseteq tvs\ (Abs\ T\ t)$  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.premis by fastforce
ultimately show ?case by simp
next
case (App t1 t2)
moreover have  $tvs\ t1 \subseteq tvs\ (t1\ \$\ t2)$   $tvs\ t2 \subseteq tvs\ (t1\ \$\ t2)$  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  $\forall idn \in set\ fresh-idns . idn \notin fst\ ' (fv\ t \cup (\bigcup t \in snd\ ' set\ insts . (fv\ t))$ 
   $\cup (fst\ ' set\ insts))$ 
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  $((idn, ty) \notin fst\ ' set\ insts$ 
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 (λx. 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.prem1 in-set-conv-nth insert-iff list.set2)
  then show ?case
  by (meson Cons.prem1 Cons.prem2 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.prem1 length-map map-fst-zip
map-map map-snd-zip nth-mem)
from this have 1: (lookup (λx. x = (idn, ty))
  (zip (map fst insts) (map2 Fv fresh-idns (map (snd ∘ fst) insts)))) = Some
(Fv fresh-idn ty)
by (simp add: Fv.prem1 Fv.prem3 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 (λx. x = (fresh-idn, ty))
  (zip (zip fresh-idns (map (snd ∘ fst) insts)) (map snd insts))) = Some u
by (simp add: Fv.prem1 Fv.prem2 distinct-zipI1 lookup-present-eq-key'')
then show ?thesis using Some 1 2 by simp
qed

```

next
case (*App t1 t2*)
moreover have $fv\ t1 \subseteq fv\ (t1\ \$\ t2)\ fv\ t2 \subseteq fv\ (t1\ \$\ t2)$ **by** *simp-all*
ultimately have *subst-term insts t1* =
 $subst\text{-}term\ (zip\ (zip\ fresh\text{-}idns\ (map\ snd\ (map\ fst\ insts)))\ (map\ snd\ insts))$
 $(subst\text{-}term\ (zip\ (map\ fst\ insts)\ (map2\ Fv\ fresh\text{-}idns\ (map\ snd\ (map\ fst\ insts))))$
t1)
and *subst-term insts t2* =
 $subst\text{-}term\ (zip\ (zip\ fresh\text{-}idns\ (map\ snd\ (map\ fst\ insts)))\ (map\ snd\ insts))$
 $(subst\text{-}term\ (zip\ (map\ fst\ insts)\ (map2\ Fv\ fresh\text{-}idns\ (map\ snd\ (map\ fst\ insts))))$
t2)
by *blast+*
then show *?case* **by** *simp*
qed *auto*

corollary *subst-term-combine'*:
assumes $length\ fresh\text{-}idns = length\ insts$
assumes *distinct fresh-idns*
assumes *distinct (map fst insts)*
assumes $\forall idn \in set\ fresh\text{-}idns . idn \notin fst\ ' (fv\ t \cup (\bigcup t \in snd\ ' set\ insts . (fv\ t) \cup (fst\ ' set\ insts)))$
shows *subst-term insts t*
 $= fold\ (\lambda single\ acc . subst\text{-}term\ [single]\ acc)\ (zip\ (zip\ fresh\text{-}idns\ (map\ snd\ (map\ fst\ insts)))\ (map\ snd\ insts))$
 $(fold\ (\lambda single\ acc . subst\text{-}term\ [single]\ acc)\ (zip\ (map\ fst\ insts)\ (map2\ Fv\ fresh\text{-}idns\ (map\ snd\ (map\ fst\ insts))))\ t)$
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))))*
 $= set\ (map\ fst\ (zip\ (map\ fst\ insts)\ (map2\ Fv\ fresh\text{-}idns\ (map\ snd\ (map\ fst\ insts))))$
by *auto*
also have $\dots = set\ (map\ fst\ insts)$ **using** *map-fst-zip assms(1)* **by** *auto*
also have $\dots = fst\ ' set\ insts$ **by** *simp*
finally show *?thesis* .
qed

have *snd ' set (zip (map fst insts) (map2 Fv fresh-idns (map snd (map fst insts))))*
 $= set\ (map2\ Fv\ fresh\text{-}idns\ (map\ snd\ (map\ fst\ insts)))$ **using** *map-snd-zip assms(1)*
by (*metis (no-types, lifting) image-set length-map*)
hence $(\bigcup (fv\ ' snd\ ' set\ (zip\ (map\ fst\ insts)\ (map2\ Fv\ fresh\text{-}idns\ (map\ snd\ (map\ fst\ insts))))))$
 $= (\bigcup (fv\ ' set\ (map2\ Fv\ fresh\text{-}idns\ (map\ snd\ (map\ fst\ insts)))))$
by *simp*

from *assms(1)* **this have** *s2*:
 $(\bigcup (fv \text{ ' } snd \text{ ' } set (zip (map fst insts) (map2 Fv fresh-idns (map snd (map fst insts))))))$
 $= (set (zip fresh-idns (map snd (map fst insts))))$
using *assms(1)* **by** (*induction fresh-idns insts rule: list-induct2*) *auto*
hence *s3*: $\bigcup (fv \text{ ' } snd \text{ ' } set (zip (map fst insts) (map2 Fv fresh-idns (map (snd \circ fst) insts))))$
 $= set (zip fresh-idns (map snd (map fst insts)))$ **by** *simp*
have $idn \notin fst \text{ ' } fst \text{ ' } set insts$ **if** $idn \in set fresh-idns$ **for** *idn*
using *that assms* **by** *auto*
hence *I*: $(idn, T) \notin fst \text{ ' } set insts$ **if** $idn \in set fresh-idns$ **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*)
 $= fold (\lambda single acc . subst-term [single] acc) (zip (map fst insts) (map2 Fv fresh-idns (map snd (map fst insts)))) t$
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* ($zip (zip fresh-idns (map snd (map fst insts))) (map snd insts)$)
 $(subst-term (zip (map fst insts) (map2 Fv fresh-idns (map snd (map fst insts))))$
 $t)$
 $= fold (\lambda single acc . subst-term [single] acc) (zip (zip fresh-idns (map snd (map fst insts))) (map snd insts))$
 $(subst-term (zip (map fst insts) (map2 Fv fresh-idns (map snd (map fst insts))))$
 $t)$
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* ($zip (zip fresh-idns (map snd (map fst insts))) (map snd insts)$)
 $(subst-term (zip (map fst insts) (map2 Fv fresh-idns (map snd (map fst insts))))$
 $t)$
 $= fold (\lambda single acc . subst-term [single] acc) (zip (zip fresh-idns (map snd (map fst insts))) (map snd insts))$
 $(fold (\lambda single acc . subst-term [single] acc) (zip (map fst insts) (map2 Fv fresh-idns (map snd (map fst insts)))) t)$
by *simp*
show *?thesis* **using** *assms subst-term-combine unfold* **by** *auto*
qed

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

assumes \neg loose-bvar t n is-closed b
shows \neg 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 \neg 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: \neg loose-bvar B lev
shows subst-bv1 (bind-fv2 (x , τ) lev B) lev a = subst-term $[(x, \tau), a]$ 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 $[(x, \tau), a]$ 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) \cdot a = subst-term $[(x, \tau), a]$ B
using *assms* instantiate-var-same-typ' **by** simp

lemma instantiate-vars-same-typ:
assumes *typs*: list-all $(\lambda(idx, ty), t) .$ typ-of t = Some ty *insts*
assumes closed-B: \neg loose-bvar B lev
shows fold $(\lambda(idx, ty), t) B .$ subst-bv1 (bind-fv2 (idx , ty) lev B) lev t *insts* B
= fold $(\lambda single .$ 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 tys: list-all ( $\lambda((idx, ty), t) . \text{typ-of } t = \text{Some } ty$ ) xs using Cons.prems by
simp
have not-loose:  $\neg \text{loose-bvar}$  (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 ( $\lambda((idx, ty), t) B . \text{subst-bv1}$  (bind-fv2 (idx, ty) lev B) lev t) (x # xs)
B
= fold ( $\lambda((idx, ty), t) B . \text{subst-bv1}$  (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 ( $\lambda((idx, ty), t) B . \text{subst-bv1}$  (bind-fv2 (idx, ty) lev B) lev t)
xs
(subst-term [((idn, ty), t)] B)
using single by simp
also have ... = fold ( $\lambda \text{single} . \text{subst-term}$  [single]) xs (subst-term [((idn, ty), t)]
B)
using Cons.IH[where B = subst-term [((idn, ty), t)] B, OF tys not-loose]
Cons.prems by blast
also have ... = fold ( $\lambda \text{single} . \text{subst-term}$  [single]) (x # xs) B
by (simp add: x)
finally show ?case .
qed

```

```

corollary instantiate-vars-same-typ':
assumes tys: list-all ( $\lambda((idx, ty), t) . \text{typ-of } t = \text{Some } ty$ ) insts
assumes closed-B:  $\neg \text{loose-bvar}$  B lev
assumes distinct: distinct (map fst insts)
assumes no-overlap:  $\bigwedge x . x \in (\bigcup t \in \text{snd} \text{ ` } (\text{set } \text{insts}) . \text{fv } t) \implies x \notin \text{fst} \text{ ` } (\text{set } \text{insts})$ 
shows fold ( $\lambda((idx, ty), t) B . \text{subst-bv1}$  (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  $\Rightarrow$  string where
fresh-name S = (if S=empty then "a" else replicate (Max (length ` S) + 1) (CHR
""))

```

```

lemma fresh-name-fresh:
  assumes finite S
  shows fresh-name S  $\notin$  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  $\forall s \in S. \text{length } (fresh\text{-name } S) > \text{length } s$  using assms by (simp add:
le-imp-less-Suc)
  thus fresh-name S  $\notin$  S by blast
qed

```

```

context
  includes String.literal.lifting
begin
lift-definition fresh-name' :: String.literal set  $\Rightarrow$  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  $\notin$  S
  by (metis assms finite-imageI fresh-name'.rep-eq fresh-name-fresh rev-image-eqI)
end

```

```

fun variant-name :: name  $\Rightarrow$  name set  $\Rightarrow$  (name  $\times$  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)  $\notin$  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  $\Rightarrow$  name where
  name (variable.Free n) = n
| name (Var (n,-)) = n

```

```

fun variant-variable :: variable  $\Rightarrow$  variable set  $\Rightarrow$  (variable  $\times$  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)  $\notin$  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  $\Rightarrow$  variable  $\Rightarrow$  variable set  $\Rightarrow$  (variable list  $\times$  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  $\forall s \in \text{set } (\text{fst } (\text{variant-variables } n \ s \ S)) . s \notin S$ 
  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'  $\notin$  S
  by (metis Suc.prem1 fst-conv variant-variable-fresh)
  moreover have I:  $\forall s \in \text{set } (\text{fst } (\text{variant-variables } n \ s' \ S')) . s \notin S'$ 
  by (metis Suc.IH Suc.prem1 s'S' finite.insertI snd-conv variant-variable-adds)
  moreover have S  $\subseteq$  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.prem1 fst-conv variant-variable-fresh*)
moreover have *I: distinct (fst (variant-variables n s' S'))*
 by (*metis Suc.IH Suc.prem1 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.prem1 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* \equiv *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* \equiv *fresh-rename-ns n B insts*

lemma *map-Pair-zip-replicate-conv:* *map (λx. 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) .*

$nm \notin (fst \text{ ` } (fv B \cup (\bigcup t \in snd \text{ ` } set\ insts . (fv t)) \cup (fst \text{ ` } set\ insts)) \cup G)$
by (*metis* (*no-types*, *lifting*) *List.finite-set add-vars'-fv finite-UN finite-Un fi-*
nite-imageI variant-names-fresh)

lemma *length-fresh-rename-ns*: $finite\ G \implies length\ (fresh\ rename\ ns\ n\ B\ insts\ G) = n$

by (*metis* (*no-types*, *lifting*) *List.finite-set add-vars'-fv finite-UN finite-Un fi-*
nite-imageI variant-names-amount)

lemma *distinct-fresh-rename-idns*: $finite\ G \implies distinct\ (fresh\ rename\ idns\ n\ B\ insts\ G)$

using *distinct-fresh-rename-ns* **by** (*metis*)

lemma *fresh-fresh-rename-idns*: $finite\ G \implies \forall nm \in set\ (fresh\ rename\ idns\ n\ B\ insts\ G) .$

$nm \notin (fst \text{ ` } (fv B \cup (\bigcup t \in snd \text{ ` } set\ insts . (fv t)) \cup (fst \text{ ` } set\ insts)) \cup G)$

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*: $finite\ G \implies length\ (fresh\ rename\ idns\ n\ B\ insts\ G) = n$

by (*metis length-fresh-rename-ns*)

end

8 Beta Normalization

theory *BetaNorm*

imports *Term*

begin

inductive *beta* :: *term* \Rightarrow *term* \Rightarrow *bool* (**infixl** \rightarrow_β 50)

where

$beta\ [simp,\ intro!]: Abs\ T\ s\ \$\ t \rightarrow_\beta\ subst\ bv2\ s\ 0\ t$
 $|\ appL\ [simp,\ intro!]: s \rightarrow_\beta t \implies s\ \$\ u \rightarrow_\beta t\ \$\ u$
 $||\ appR\ [simp,\ intro!]: s \rightarrow_\beta t \implies u\ \$\ s \rightarrow_\beta u\ \$\ t$
 $||\ abs\ [simp,\ intro!]: s \rightarrow_\beta t \implies Abs\ T\ s \rightarrow_\beta Abs\ T\ t$

abbreviation

beta-reds :: *term* \Rightarrow *term* \Rightarrow *bool* (**infixl** \rightarrow_β^* 50) **where**
 $s \rightarrow_\beta^* t == beta^{**}\ s\ t$

inductive-cases *beta-cases* [*elim!*]:

$Bv\ i \rightarrow_\beta t$
 $Fv\ idn\ S \rightarrow_\beta t$
 $Abs\ T\ r \rightarrow_\beta s$
 $s\ \$\ t \rightarrow_\beta u$

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

lemma *rtrancl-beta-Abs* [intro]:

$$s \rightarrow_{\beta^*} s' \implies \text{Abs } T \ s \rightarrow_{\beta^*} \text{Abs } T \ s'$$

by (induct set: *rtrancl*) (blast intro: *rtranclp.rtrancl-into-rtrancl*)+

lemma *rtrancl-beta-AppL*:

$$s \rightarrow_{\beta^*} s' \implies s \ \$ \ t \rightarrow_{\beta^*} s' \ \$ \ t$$

by (induct set: *rtrancl*) (blast intro: *rtranclp.rtrancl-into-rtrancl*)+

lemma *rtrancl-beta-AppR*:

$$t \rightarrow_{\beta^*} t' \implies s \ \$ \ t \rightarrow_{\beta^*} s \ \$ \ t'$$

by (induct set: *rtrancl*) (blast intro: *rtranclp.rtrancl-into-rtrancl*)+

lemma *rtrancl-beta-App* [intro]:

$$s \rightarrow_{\beta^*} s' \implies t \rightarrow_{\beta^*} t' \implies s \ \$ \ t \rightarrow_{\beta^*} s' \ \$ \ t'$$

by (blast intro!: *rtrancl-beta-AppL* *rtrancl-beta-AppR* intro: *rtranclp-trans*)

theorem *subst-bv2-preserves-beta* [simp]:

$$r \rightarrow_{\beta} s \implies \text{subst-bv2 } r \ k \ u \rightarrow_{\beta} \text{subst-bv2 } s \ k \ u$$

by (induct arbitrary: *k u* set: *beta*) (simp-all add: *subst-bv2-subst-bv2[symmetric]*)

theorem *subst-bv2-preserves-beta'*: $r \rightarrow_{\beta^*} s \implies \text{subst-bv2 } r \ i \ t \rightarrow_{\beta^*} \text{subst-bv2 } s \ i \ t$

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]:

$$r \rightarrow_{\beta} s \implies \text{lift } r \ i \rightarrow_{\beta} \text{lift } s \ i$$

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'*: $r \rightarrow_{\beta^*} s \implies \text{lift } r \ i \rightarrow_{\beta^*} \text{lift } s \ i$

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]: $r \rightarrow_{\beta} s \implies \text{subst-bv2 } t \ i \ r \rightarrow_{\beta^*} \text{subst-bv2 } t \ i \ s$

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'*: $r \rightarrow_{\beta}^* s \implies \text{subst-bv2 } t \text{ i } r \rightarrow_{\beta}^* \text{subst-bv2 } t \text{ i } s$

apply (*induct set*: *rtranclp*)
apply (*auto elim*: *rtranclp-trans subst-bv2-preserves-beta2*)
done

lemma *beta-preserves-typ-of1*: $\text{typ-of1 } Ts \text{ } r = \text{Some } T \implies r \rightarrow_{\beta} s \implies \text{typ-of1 } Ts \text{ } s = \text{Some } T$

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*: $\text{typ-of1 } Ts \text{ } u = \text{Some } argT$ **and** $\text{typ-of1 } Ts \text{ } f = \text{Some } (argT \rightarrow T)$

by (*meson typ-of1-split-App-obtains*)

from *5* **show** *?case* **apply** $-$

apply (*ind-cases* *f \$ u \rightarrow_{\beta} s* **for** *f u s*)

using $\langle \text{typ-of1 } Ts \text{ } f = \text{Some } (argT \rightarrow T) \rangle$ *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*: $\text{typ-of } r = \text{Some } T \implies r \rightarrow_{\beta} s \implies \text{typ-of } s = \text{Some } T$

by (*metis beta-preserves-typ-of1 typ-of-def*)

lemma *beta-star-preserves-typ-of1*: $r \rightarrow_{\beta}^* s \implies \text{typ-of1 } Ts \text{ } r = \text{Some } T \implies \text{typ-of1 } Ts \text{ } s = \text{Some } T$

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*: $\text{beta-reducible } t \implies \exists t'. t \rightarrow_{\beta} t'$

proof (*induction* *t*)

case (*App t1 t2*)

then show *?case* **using** *App* **by** (*cases t1*) *auto*

qed *auto*

lemma *beta-step-imp-beta-reducible*: $t \rightarrow_{\beta} t' \implies \text{beta-reducible } t$

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.prem1 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.prem1 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.prem1 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.prem1 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.prem1 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  $\exists$  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  $\exists$  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  $\exists$  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  $\exists$  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  $\exists$  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  $\exists$  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 $\text{beta-norm } t = \text{Some } t' \implies \text{typ-of1 } Ts \ t = \text{Some } T \implies \text{typ-of1 } Ts \ t' = \text{Some } T$

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*
 $t = \text{Some } t'$

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.prem*s **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.prem*s **by** *auto*

next

case (*abs s t T*)

then show *?case* **by** *auto*

qed

lemma *beta-subst-bv1*: $s \rightarrow_{\beta} t \implies$ *subst-bv1 s lev x* \rightarrow_{β} *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*: $s \rightarrow_{\beta} t \implies$ *subst-bv x s* \rightarrow_{β} *subst-bv x t*

by (*simp add: substn-subst-0'*)

lemma *subst-bv1-beta*:

$\text{subst-bv1 } s \text{ (length (T\#Ts)) } x \rightarrow_{\beta} \text{subst-bv1 } t \text{ (length (T\#Ts)) } x$

$\implies \text{typ-of1 } Ts \text{ } s = \text{Some } ty$

$\implies \text{typ-of1 } Ts \text{ } t = \text{Some } ty$

$\implies s \rightarrow_{\beta} t$

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.prem1 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.prem1 loose-bvar-Suc no-loose-bvar-imp-no-subst-bv1
        typ-of1-imp-no-loose-bvar
        by (metis length-Cons)
      ultimately have s'  $\rightarrow_\beta$  t'
        by (metis abs.hyps(1) abs.hyps(3) abs.hyps(4) abs.prem1 abs.prem2 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  $\Rightarrow$  nat  $\Rightarrow$  term list  $\Rightarrow$  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 ( $\lambda$ 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]: $\text{subst-bvs1}' t \text{ lev } [] = t$
 by (induction t lev []::term list rule: subst-bvs1.induct) auto

lemma *subst-bvs1'-eq* [simp]: $\text{args} \neq [] \implies \text{subst-bvs1}' (Bv k) k \text{ args} = \text{args} ! 0$
 by simp

lemma *subst-bvs1'-eq'* [simp]: $i < \text{length args} \implies \text{subst-bvs1}' (Bv (k+i)) k \text{ args} =$

args ! i
by auto

lemma subst-bvs1'-gt [simp]:
 $i + \text{length args} < j \implies \text{subst-bvs1}' (Bv j) i \text{ args} = Bv (j - \text{length args})$
by auto

lemma subst-bv2-lt [simp]: $j < i \implies \text{subst-bvs1}' (Bv j) i u = Bv j$
by simp

lemma subst-bvs1'-App[simp]: $\text{subst-bvs1}' (s\$t) k \text{ args}$
 $= \text{subst-bvs1}' s k \text{ args} \$ \text{subst-bvs1}' t k \text{ args}$
by simp

lemma incr-bv-incr-bv:
 $i < k + 1 \implies \text{incr-bv inc2} (k + \text{inc1}) (\text{incr-bv inc1 } i t) = \text{incr-bv inc1 } i (\text{incr-bv inc2 } k t)$
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': $\text{subst-bvs1 } t n s = \text{subst-bvs1}' t n (\text{map} (\text{incr-bv } n) s)$
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: $\text{subst-bvs1 } t 0 s = \text{subst-bvs1}' t 0 s$
proof–
have $\text{subst-bvs1 } t 0 s = \text{subst-bvs1}' t 0 (\text{map} (\text{incr-bv } 0) s)$
using *subst-bvs1-subst-bvs1'* **by** *blast*
moreover have $\text{map} (\text{incr-bv } 0) s = s$
by (*induction s*) *auto*
ultimately show *?thesis*
by *simp*
qed

corollary subst-bvs-subst-bvs1': $\text{subst-bvs } s t = \text{subst-bvs1}' t 0 s$
using *subst-bvs-def subst-bvs1-subst-bvs1'-0* **by** *simp*

lemma no-loose-bvar-subst-bvs1'-unchanged: $\neg \text{loose-bvar } t \text{ lev} \implies \text{subst-bvs1}' t \text{ lev args} = t$
by (*induction t lev args rule: subst-bvs1'.induct*) *auto*

lemma *subst-bvs1'-step*: $\forall x \in \text{set } (a\#\text{args}) . \text{is-closed } x \implies$
 $\text{subst-bvs1}' t \text{ lev } (a\#\text{args}) = \text{subst-bvs1}' (\text{subst-bv2 } t \text{ lev } a) \text{ lev args}$
proof (*induction* $t \text{ lev args}$ *rule*: *subst-bvs1'.induct*)
case ($1 i \text{ lev args}$)
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*: $\neg \text{loose-bvar } a \text{ lev} \implies \neg \text{loose-bvar } (\text{incr-bv } \text{inc } \text{lev } a) (\text{lev}+\text{inc})$
by (*induction* $a \text{ lev}$ *rule*: *loose-bvar.induct*) *auto*

lemma *not-loose-bvar-incr-bv-less*:
 $i < j \implies \neg \text{loose-bvar } (\text{incr-bv } \text{inc } i \ a) (\text{lev}+\text{inc}) \implies \neg \text{loose-bvar } (\text{incr-bv } \text{inc } j \ a) (\text{lev}+\text{inc})$
proof (*induction* $\text{inc } i \ a$ *arbitrary*: $\text{lev } j$ *rule*: *incr-bv.induct*)
case ($2 \text{ inc } n \ T \ \text{body}$)
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*: $\forall x \in \text{set } \text{args} . \text{is-closed } x \implies \neg \text{loose-bvar } (\text{subst-bv2 } t \text{ lev } a) \text{ lev} \implies$
 $\text{subst-bvs1}' t \text{ lev } (a\#\text{args}) = \text{subst-bvs1}' (\text{subst-bv2 } t \text{ lev } a) \text{ lev args}$
proof (*induction* $t \text{ lev args}$ *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 ($2 \ T \ \text{body} \ \text{lev } \text{args}$)
then show *?case* **using** *no-loose-bvar-subst-bvs1'-unchanged*
by (*auto simp add: is-open-def*)
next
case ($3 \ f \ t \ \text{lev } u$)
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*: $is-closed\ t \implies subst-bv2\ t\ n\ u = t$
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*: $\forall x \in set\ (a\#\ args) . is-closed\ x \implies$
 $subst-bv2\ (subst-bvs1'\ t\ (Suc\ lev)\ args)\ lev\ a$
 $= subst-bvs1'\ t\ lev\ (a\#\ args)$

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 $i - Suc\ lev < length\ args \neg i < Suc\ lev$

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 $\dots = args\ !\ (i - Suc\ lev)$

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 $i \geq Suc\ lev\ i - lev \geq length\ args \neg i < Suc\ lev$

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*:

$\forall x \in set\ (a\#\ args) . is-closed\ x \implies$

$subst-bv\ a\ (subst-bvs1'\ t\ 1\ args) = subst-bvs\ (a\#\ args)\ t$

using *subst-bvs1'-step-extend-lower-level*

by (*simp add: subst-bvs-subst-bvs1' substn-subst-0'*)

lemma *subst-bvs1'-preserves-beta*:

$\forall x \in set\ u . is-closed\ x \implies r \rightarrow_\beta s \implies subst-bvs1'\ r\ k\ u \rightarrow_\beta subst-bvs1'\ s\ k\ u$

proof (*induction u arbitrary: r s*)

case *Nil*

then show *?case by auto*

next

case (*Cons a u*)

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

lemma *subst-bvs1'-fold*: $\forall x \in \text{set args} . \text{is-closed } x \implies$
 $\text{subst-bvs1}' \ t \ \text{lev args} = \text{fold } (\lambda \text{arg } t . \text{subst-bv2 } t \ \text{lev arg}) \ \text{args } t$
by (*induction args arbitrary: t*) (*simp-all add: subst-bvs1'-step*)

lemma *subst-bvs1'-Abs[simp]*: $\forall x \in \text{set args} . \text{is-closed } x \implies$
 $\text{subst-bvs1}' (\text{Abs } T \ t) \ \text{lev args} = \text{Abs } T (\text{subst-bvs1}' \ t \ (\text{Suc lev}) \ \text{args})$
by (*simp add: is-open-def map-idI*)

lemma *subst-bvs-Abs[simp]*: $\forall x \in \text{set args} . \text{is-closed } x \implies$
 $\text{subst-bvs } \text{args} (\text{Abs } T \ t) = \text{Abs } T (\text{subst-bvs1}' \ t \ 1 \ \text{args})$
using *subst-bvs1'-Abs subst-bvs-subst-bvs1'* **by** *auto*

lemma *subst-bvs1'-incr-bv [simp]*:
 $\text{subst-bvs1}' (\text{incr-bv } (\text{length } ss) \ k \ t) \ k \ ss = t$
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]*:
 $j < i + 1 \implies \text{lift } (\text{subst-bvs1}' \ t \ j \ ss) \ i$
 $= \text{subst-bvs1}' (\text{lift } t \ (i + \text{length } ss)) \ j \ (\text{map } (\lambda s . \text{lift } s \ i) \ ss)$
proof (*induct t arbitrary: i j ss*)
case (*Abs T t*)
hence *I*: $\text{lift } (\text{subst-bvs1}' \ t \ (\text{Suc } j) \ (\text{map } (\lambda t . \text{lift } t \ 0) \ ss)) \ (\text{Suc } i) =$
 $\text{subst-bvs1}' (\text{lift } t \ (\text{Suc } i + \text{length } (\text{map } (\lambda t . \text{lift } t \ 0) \ ss))) \ (\text{Suc } j) \ (\text{map } (\lambda a . \text{lift}$
 $a \ (\text{Suc } i)) \ (\text{map } (\lambda t . \text{lift } t \ 0) \ ss))$
by *auto*

have $\text{lift } (\text{subst-bvs1}' (\text{Abs } T \ t) \ j \ ss) \ i$
 $= \text{Abs } T (\text{lift } (\text{subst-bvs1}' \ t \ (\text{Suc } j) \ (\text{map } (\lambda t . \text{lift } t \ 0) \ ss)) \ (\text{Suc } i))$
by *simp*
also have $\dots = \text{Abs } T$
 $(\text{subst-bvs1}' (\text{lift } t \ (\text{Suc } i + \text{length } (\text{map } (\text{incr-bv } 1 \ 0) \ ss))) \ (\text{Suc } j)$
 $(\text{map } (\text{incr-bv } 1 \ (\text{Suc } i)) \ (\text{map } (\text{incr-bv } 1 \ 0) \ ss)))$
using *I* **by** *auto*
also have $\dots = \text{Abs } T$
 $(\text{subst-bvs1}' (\text{lift } t \ (\text{Suc } i + \text{length } (\text{map } (\text{incr-bv } 1 \ 0) \ ss))) \ (\text{Suc } j)$
 $(\text{map } (\lambda t . \text{lift } t \ 0) \ (\text{map } (\lambda t . \text{lift } t \ i) \ ss)))$
proof–

have $\text{map } (\lambda t . \text{lift } t \text{ (Suc } i)) \text{ (map } (\lambda t . \text{lift } t \ 0) \text{ ss)} = \text{map } (\lambda t . \text{lift } t \ 0) \text{ (map } (\lambda t . \text{lift } t \ i) \text{ ss)}$
using *lift-lift by auto*
thus *?thesis unfolding lift-def*
by *argo*
qed
also have $\dots = \text{subst-bvs1'} \text{ (Abs } T \text{ (lift } t \text{ (Suc } i + \text{length (map (incr-bv 1 0) ss)))) } j$
 $\text{(map } (\lambda t . \text{lift } t \ i) \text{ ss)}$
by *auto*
finally show *?case*
by *simp*
qed *(auto simp add: diff-Suc lift-lift split: nat.split)*

lemma *lift-subst-bvs1'-lt:*
 $i < j + 1 \implies \text{lift (subst-bvs1' } t \ j \text{ ss)} \ i$
 $= \text{subst-bvs1'} \text{ (lift } t \ i) \text{ (} j + 1 \text{) (map } (\lambda s . \text{lift } s \ i) \text{ ss)}$
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:*
 $i < j + 1 \implies$
 $\text{subst-bv2(subst-bvs1' } t \text{ (Suc } j) \text{ (map } (\lambda v . \text{lift } v \ i) \text{ vs)}) } \ i \text{ (subst-bvs1' } u \ j \text{ vs)}$
 $= \text{subst-bvs1'} \text{ (subst-bv2 } t \ i \ u) \ j \text{ vs}$
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:* $\text{fv (subst-bv2 } t \ \text{lev } u) \subseteq \text{fv } t \cup \text{fv } u$
by *(induction t lev u rule: subst-bv2.induct) auto*

lemma *beta-fv:* $s \rightarrow_{\beta} t \implies \text{fv } t \subseteq \text{fv } s$
by *(induction rule: beta.induct) (use fv-subst-bv2-upper-bound in auto)*

lemma *loose-bvar1-subst-bvs1'-closed:* $\neg \text{loose-bvar1 } t \ \text{lev} \implies \text{lev} < k \implies \forall x \in \text{set } us . \text{is-closed } x$
 $\implies \neg \text{loose-bvar1 (subst-bvs1' } t \ k \text{ us) lev}$
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'-closed:* $\neg \text{is-dependent } t \implies \forall x \in \text{set } us . \text{is-closed } x$
 $\implies \neg \text{is-dependent (subst-bvs1' } t \text{ (Suc } k) \text{ us)}$
by *(simp add: is-dependent-def loose-bvar1-subst-bvs1'-closed)*

```

end
Facts about beta normalization involving theories
theory BetaNormProof
  imports BetaNorm Theory
begin

lemma beta-preserves-term-ok':  $term-ok' \Sigma r \implies r \rightarrow_{\beta} s \implies term-ok' \Sigma s$ 
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:  $term-ok \Theta r \implies r \rightarrow_{\beta} s \implies term-ok \Theta s$ 
proof –
  assume a1: term-ok \Theta 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':  $r \rightarrow_{\beta^*} s \implies term-ok' \Sigma r \implies term-ok' \Sigma s$ 

```

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

corollary *beta-star-preserves-term-ok*: $r \rightarrow_{\beta}^* s \implies \text{term-ok } \text{thy } r \implies \text{term-ok } \text{thy } s$
 using *beta-star-preserves-term-ok'* *beta-star-preserves-typ-of1* *wt-term-def* *typ-of-def*
 by auto

corollary *term-ok-beta-norm*: $\text{term-ok } \text{thy } t \implies \text{beta-norm } t = \text{Some } t' \implies \text{term-ok } \text{thy } t'$
 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 \rightarrow_{η} 50)

where

eta [*simp*, *intro*]: $\neg \text{is-dependent } s \implies \text{Abs } T (s \$ \text{Bv } 0) \rightarrow_{\eta} \text{decr } 0 s$
 | *appL* [*simp*, *intro*]: $s \rightarrow_{\eta} t \implies s \$ u \rightarrow_{\eta} t \$ u$
 | *appR* [*simp*, *intro*]: $s \rightarrow_{\eta} t \implies u \$ s \rightarrow_{\eta} u \$ t$
 | *abs* [*simp*, *intro*]: $s \rightarrow_{\eta} t \implies \text{Abs } T s \rightarrow_{\eta} \text{Abs } T t$

abbreviation

eta-reds :: *term* \Rightarrow *term* \Rightarrow *bool* (infixl \rightarrow_{η}^* 50) where
 $s \rightarrow_{\eta}^* t \equiv \text{eta}^{**} s t$

abbreviation

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

inductive-cases *eta-cases* [*elim!*]:

Abs $T s \rightarrow_{\eta} z$
 $s \$ t \rightarrow_{\eta} u$
 $\text{Bv } i \rightarrow_{\eta} t$

lemma *subst-bv2-not-free* [*simp*]: $\neg \text{loose-bvar1 } s i \implies \text{subst-bv2 } s i t = \text{subst-bv2 } s i u$

by (induction *s* arbitrary: *i t u*) (*simp-all* add:)

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

$\text{loose-bvar1 } (\text{lift } t k) i = (i < k \wedge \text{loose-bvar1 } t i \vee k < i \wedge \text{loose-bvar1 } t (i - 1))$

by (induct *t* arbitrary: *i k*) (auto cong: *conj-cong*)

lemma *free-subst-bv2* [simp]:
 $\text{loose-bvar1 } (\text{subst-bv2 } s \ k \ t) \ i =$
 $(\text{loose-bvar1 } s \ k \wedge \text{loose-bvar1 } t \ i \vee \text{loose-bvar1 } s \ (\text{if } i < k \text{ then } i \text{ else } i + 1))$
apply (*induct s arbitrary: i k t*)
using *free-lift apply (simp-all add: diff-Suc split: nat.split)*
by *blast*

lemma *free-eta*: $s \rightarrow_{\eta} t \implies \text{loose-bvar1 } t \ i = \text{loose-bvar1 } s \ i$
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*:
 $s \rightarrow_{\eta} t \implies \neg \text{loose-bvar1 } s \ i \implies \neg \text{loose-bvar1 } t \ i$
by (*simp add: free-eta*)

lemma *no-loose-bvar1-subst-bv2-decr*: $\neg \text{loose-bvar1 } t \ i \implies \text{subst-bv2 } t \ i \ x = \text{decr}$
 $i \ t$
by (*induction t i x rule: subst-bv2.induct*) *auto*

lemma *eta-subst-bv2* [simp]:
 $s \rightarrow_{\eta} t \implies \text{subst-bv2 } s \ i \ u \rightarrow_{\eta} \text{subst-bv2 } t \ i \ u$
proof (*induction s t arbitrary: u i rule: eta.induct*)
case (*eta s T*)
hence *1*: $\neg \text{loose-bvar1 } s \ 0$
using *is-dependent-def* **by** *simp*
have $\text{decr } 0 \ s = \text{subst-bv2 } s \ 0 \ \text{dummy}$ **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*: $\neg \text{loose-bvar } s \ i \implies \text{lift } (\text{decr } i \ s) \ i = s$
by (*induct s arbitrary: i*) *simp-all*

lemma *decr-is-closed*[simp]: $\text{is-closed } t \implies \text{decr lev } t = t$
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*: $\text{eta-reducible } t \implies \exists t'. t \rightarrow_{\eta} t'$
by (*induction t rule: eta-reducible.induct*) *auto*

lemma *eta-step-imp-eta-reducible*: $t \rightarrow_{\eta} t' \implies \text{eta-reducible } t$
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.prem(1) 4.prem(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.prem(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  $\tau = T \rightarrow rty$ 
    by (metis 1(1) 4.prem(2) calculation option.inject typ-of-Abs-body-typ')
    ultimately have typ-of1 (T#Ts) s = Some  $\tau$ 

```

```

      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  $\tau$ 
        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.prem(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.prem(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 (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*: $t \rightarrow_{\eta} t' \implies \text{typ-of } t = \text{Some } \tau \implies \text{typ-of } t' = \text{Some } \tau$

```

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

lemma eta-star-preserves-typ-of1:  $r \rightarrow_{\eta}^* s \implies \text{typ-of1 } Ts \ r = \text{Some } T \implies \text{typ-of1 } Ts \ s = \text{Some } T$ 
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:  $r \rightarrow_{\eta}^* s \implies \text{typ-of } r = \text{Some } T \implies \text{typ-of } s = \text{Some } T$ 
using eta-star-preserves-typ-of1 typ-of-def by simp

lemma subst-bvs1'-decr:  $\forall x \in \text{set } us. \text{is-closed } x \implies \neg \text{loose-bvar1 } t \ k \implies \text{subst-bvs1'} (decr \ k \ t) \ k \ us = decr \ k (subst-bvs1' \ t (Suc \ k) \ us)$ 
by (induction  $k \ t$  arbitrary:  $us$  rule: decr.induct) (auto simp add: is-open-def)

lemma subst-bvs-decr:  $\forall x \in \text{set } us. \text{is-closed } x \implies \neg \text{is-dependent } t \implies \text{subst-bvs } us (decr \ 0 \ t) = decr \ 0 (subst-bvs1' \ t \ 1 \ us)$ 
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:  $\text{term-ok}' \Sigma \ t \implies \text{term-ok}' \Sigma (decr \ i \ t)$ 
by (induction  $i \ t$  rule: decr.induct) auto

lemma eta-preserves-term-ok':  $\text{term-ok}' \Sigma \ r \implies r \rightarrow_{\eta} s \implies \text{term-ok}' \Sigma \ s$ 
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*: $\text{term-ok } \Theta r \implies r \rightarrow_{\eta} s \implies \text{term-ok } \Theta s$

proof –

```

  assume a1: term-ok  $\Theta$  r
  assume a2:  $r \rightarrow_{\eta} s$ 
  then have None  $\neq$  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'*: $r \rightarrow_{\eta}^* s \implies \text{term-ok}' \Sigma r \implies \text{term-ok}' \Sigma s$
 by (*induction rule*: rtranclp.induct) (*auto simp add*: eta-preserves-term-ok')

corollary *eta-star-preserves-term-ok*: $r \rightarrow_{\eta}^* s \implies \text{term-ok } \text{thy } r \implies \text{term-ok } \text{thy } s$

using eta-star-preserves-term-ok' eta-star-preserves-typ-of1 wt-term-def typ-of-def
 by auto

corollary *term-ok-eta-norm*: $\text{term-ok } \text{thy } t \implies \text{eta-norm } t = t' \implies \text{term-ok } \text{thy } t'$
 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* \equiv

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

lemma *inst-ok-imp-wf-inst*:

inst-ok Θ *insts* \implies *wf-inst* Θ ($\lambda idn S . the-default (Tv idn S) (lookup (\lambda x. x=(idn, S)) insts)$)
by (*induction insts*) (*auto split: if-splits prod.splits*)

lemma *term-ok'-eta-norm*: *term-ok'* $\Sigma t \implies$ *term-ok'* $\Sigma (eta-norm t)$

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* \implies *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* \equiv *map-option eta-norm (beta-norm t)*

lemma *beta-eta-norm t = Some t' \implies \neg eta-reducible t'*

using *not-eta-reducible-eta-norm* **by** *auto*

lemma *term-ok-beta-eta-norm*: *term-ok* *thy t* \implies *beta-eta-norm t = Some t' \implies 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 \implies beta-eta-norm t = Some t' \implies 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* \in *axioms* Θ *inst-ok* Θ *insts*

shows $\Theta, \Gamma \vdash$ *subst-typ' insts A*

proof—

have *wf-inst* Θ ($\lambda idn S . the-default (Tv idn S) (lookup (\lambda x. x=(idn, S)) insts)$)

using *inst-ok-imp-wf-inst assms(3)* **by** *blast*

moreover have *subst-typ' insts A*

$=$ *tsubst A* ($\lambda idn S . the-default (Tv idn S) (lookup (\lambda x. x=(idn, S)) insts)$)

by (*simp add: tsubst-simulates-subst-typ'*)

ultimately show *?thesis*

using *assms axiom* **by** *simp*

qed

corollary *axiom'*: *wf-theory* $\Theta \implies A \in$ *axioms* $\Theta \implies \Theta, \Gamma \vdash A$

apply (*subst subst-typ'-nil[symmetric]*)

```

using axiom-subst-typ' inst-ok-nil by metis

lemma has-sort-Tv-refl: wf-osig oss  $\implies$  sort-ex (subclass oss) S  $\implies$  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  $\Theta \implies$  typ-ok  $\Theta$  (Tv v S)  $\implies$  has-sort (osig (sig  $\Theta$ )) (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  $\Theta \implies$  distinct l  $\implies$   $\forall (v, S) \in$  set l . typ-ok  $\Theta$  (Tv v S)  $\implies$  wf-inst
 $\Theta$   $\varrho$ 
 $\implies$  inst-ok  $\Theta$  (map ( $\lambda(v, S) . ((v, S), \varrho v S)$ ) l)
proof (induction l)
case Nil
then show ?case by simp
next
case (Cons a l)
have I: inst-ok  $\Theta$  (map ( $\lambda(v, S) . ((v, S), \varrho v S)$ ) l)
using Cons by fastforce

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

moreover have wf-type (sig  $\Theta$ ) (case-prod  $\varrho$  a)
using Cons.prem(3-4) by auto (metis typ-ok-Tv wf-type-imp-typ-ok-sig)
moreover hence typ-ok  $\Theta$  (case-prod  $\varrho$  a)
by simp
moreover hence has-sort (osig (sig  $\Theta$ )) (case-prod  $\varrho$  a) (snd a)
using Cons.prem 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  $\subseteq$  Types t
by (induction t) auto
lemma osig-tvsT-subset-SortsT: snd ' tvsT T  $\subseteq$  SortsT T
by (induction T) auto

```

lemma *osig-tvs-subset-Sorts*: $snd \text{ ` tvs } t \subseteq \text{Sorts } t$
by (*induction t*) (*use osig-tvsT-subset-SortsT in <auto simp add: image-subset-iff>*)

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

lemma *term-ok-Types-typ-ok*: $wf\text{-theory } \Theta \implies term\text{-ok } \Theta t \implies \tau \in \text{Types } t \implies typ\text{-ok } \Theta \tau$
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*:
 $is\text{-std}\text{-sig } \Sigma \implies term\text{-ok}' \Sigma t \implies (x, \tau) \in fv t \implies typ\text{-ok}\text{-sig } \Sigma \tau$
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*: $wf\text{-theory } \Theta \implies term\text{-ok } \Theta t \implies (x, \tau) \in fv t \implies typ\text{-ok } \Theta \tau$
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*:
 $is\text{-std}\text{-sig } \Sigma \implies typ\text{-ok}\text{-sig } \Sigma T \implies (x, S) \in tvsT T \implies wf\text{-sort } (subclass (osig \Sigma)) S$
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*:
 $is\text{-std}\text{-sig } \Sigma \implies term\text{-ok}' \Sigma t \implies (x, S) \in tvs t \implies wf\text{-sort } (subclass (osig \Sigma)) S$
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  $\Sigma \implies$  typ-ok-sig  $\Sigma T \implies (x,S) \in$  tvsT  $T \implies$  wf-sort (subclass (osig
 $\Sigma$ ))  $S$ 
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  $\Theta$  term-ok  $\Theta t (x, S) \in$  tvs  $t$ 
  shows wf-sort (subclass (osig (sig  $\Theta$ )))  $S$ 
proof-
  have term-ok' (sig  $\Theta$ )  $t$ 
  using assms(2) by (simp add: wt-term-def)
  moreover have is-std-sig (sig  $\Theta$ )
  using assms by (cases  $\Theta$  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'  $\Sigma t$ 
  shows term-ok'  $\Sigma$  (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'  $\Sigma t$ 
  shows term-ok'  $\Sigma$  (bind-fv ( $v,\tau$ )  $t$ )
  using term-ok'-bind-fv2 bind-fv-def assms by metis

lemma term-ok'-Abs-fv:
  assumes term-ok'  $\Sigma t$  typ-ok-sig  $\Sigma \tau$ 
  shows term-ok'  $\Sigma$  (Abs  $\tau$  (bind-fv ( $v,\tau$ )  $t$ ))
  using term-ok'-bind-fv assms by simp

lemma term-ok'-mk-all:
  assumes wf-theory  $\Theta$  and term-ok' (sig  $\Theta$ )  $B$  and typ-of  $B =$  Some propT
  and typ-ok  $\Theta \tau$ 
  shows term-ok' (sig  $\Theta$ ) (mk-all  $x \tau B$ )
  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* \implies *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* \cdot *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 \rightarrow tty)* *typ-of* *u* = *Some uty*
shows *term-ok* Θ (*f* \cdot *u*)
using *assms term-ok'-betapply wt-term-def typ-of-betapply assms* **by** *auto*

lemma *typ-ok-sig-subst-ty*:

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

hence $\forall ty \in set (map (subst-ty inst) Ts)$. *typ-ok-sig* Σ *ty*
by *simp*

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

corollary $subst\text{-}typ\text{-}tinstT: tinstT (subst\text{-}typ insts ty) ty$
unfolding $tinstT\text{-}def$ **using** $tsubstT\text{-}simulates\text{-}subst\text{-}typ$ **by** $fastforce$

lemma $tsubstT\text{-}trans: tsubstT ty \rho1 = ty1 \implies tsubstT ty1 \rho2 = ty2$
 $\implies tsubstT ty (\lambda idx s . case \rho1 idx s of Tv idx' s' \Rightarrow \rho2 idx' s'$
 $| Ty s Ts \Rightarrow Ty s (map (\lambda T. tsubstT T \rho2) Ts)) = ty2$
unfolding $tinstT\text{-}def$ **proof** $(induction ty arbitrary: ty1 ty2)$
case $(Tv idx s)$
then show $?case$ **by** $(cases \rho1 idx s) auto$
qed $auto$

corollary $tinstT\text{-}trans: tinstT ty1 ty \implies tinstT ty2 ty1 \implies tinstT ty2 ty$
unfolding $tinstT\text{-}def$ **using** $tsubstT\text{-}trans$ **by** $blast$

lemma $term\text{-}ok'\text{-}subst\text{-}typ'$:
assumes $is\text{-}std\text{-}sig \Sigma$ **and** $term\text{-}ok' \Sigma t$ **and** $distinct (map fst insts)$
and $list\text{-}all (typ\text{-}ok\text{-}sig \Sigma) (map snd insts)$
shows $term\text{-}ok' \Sigma (subst\text{-}typ' insts t)$
using $assms$ **by** $(induction t)$
 $(use typ\text{-}ok\text{-}sig\text{-}subst\text{-}typ subst\text{-}typ\text{-}tinstT tinstT\text{-}trans$ **in** $\langle auto split: option.splits \rangle)$

lemma
 $term\text{-}ok'\text{-}occs:$
 $is\text{-}std\text{-}sig \Sigma \implies term\text{-}ok' \Sigma t \implies occs u t \implies term\text{-}ok' \Sigma u$

by (*induction t*) *auto*

lemma *typ-of1-tsubst*:

$\text{typ-of1 } Ts \ t = \text{Some } ty \implies \text{typ-of1 } (\text{map } (\lambda T . \text{tsubstT } T \ \varrho) \ Ts) \ (\text{tsubst } t \ \varrho) = \text{Some } (\text{tsubstT } ty \ \varrho)$

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.prem*s **obtain** *u-ty* **where** *u-ty: typ-of1 Ts u = Some u-ty* **by** (*auto simp add: bind-eq-Some-conv*)

from *this 5.prem*s **have** *f-ty: typ-of1 Ts f = Some (u-ty \rightarrow ty)*

by (*auto simp add: bind-eq-Some-conv typ-of1-arg-ty[OF 5.prem(1)] split: if-splits typ.splits option.splits*)

from *u-ty 5.IH(1)* **have** $\text{typ-of1 } (\text{map } (\lambda T . \text{tsubstT } T \ \varrho) \ Ts) \ (\text{tsubst } u \ \varrho) = \text{Some } (\text{tsubstT } u\text{-ty } \varrho)$

by *simp*

moreover from *u-ty f-ty 5.IH(2)* **have** $\text{typ-of1 } (\text{map } (\lambda T . \text{tsubstT } T \ \varrho) \ Ts) \ (\text{tsubst } f \ \varrho)$

$= \text{Some } (\text{tsubstT } (u\text{-ty} \rightarrow \text{ty}) \ \varrho)$

by *simp*

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

qed *auto*

corollary *typ-of1-tsubst-weak*:

assumes $\text{typ-of1 } Ts \ t = \text{Some } ty$

assumes $\text{typ-of1 } (\text{map } (\lambda T . \text{tsubstT } T \ \varrho) \ Ts) \ (\text{tsubst } t \ \varrho) = \text{Some } ty'$

shows $\text{tsubstT } ty \ \varrho = ty'$

using *assms typ-of1-tsubst* **by** *auto*

lemma *tsubstT-no-change[simp]*: $\text{tsubstT } T \ Tv = T$

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

lemma *term-ok-mk-eq-same-ty*:

assumes *wf-theory* Θ

assumes *term-ok* Θ (*mk-eq s t*)

shows $\text{typ-of } s = \text{typ-of } t$

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*: $\text{typ-of } f = \text{Some } (\tau \rightarrow \tau') \implies \text{typ-of } (\text{Abs } \tau \ (f \ \$ \ Bv \ 0)) = \text{Some } (\tau \rightarrow \tau')$

using *typ-of1-weaken* **by** (*fastforce simp add: bind-eq-Some-conv typ-of-def*)

```

lemma term-okI: term-ok' (sig  $\Theta$ ) t  $\implies$  typ-of t  $\neq$  None  $\implies$  term-ok  $\Theta$  t
  by (simp add: wt-term-def)
lemma term-okD1: term-ok  $\Theta$  t  $\implies$  term-ok' (sig  $\Theta$ ) t
  by (simp add: wt-term-def)
lemma term-okD2: term-ok  $\Theta$  t  $\implies$  typ-of t  $\neq$  None
  by (simp add: wt-term-def)

lemma term-ok-imp-typ-ok': assumes wf-theory  $\Theta$  term-ok  $\Theta$  t shows typ-ok  $\Theta$ 
(the (typ-of t))
proof-
  obtain ty where ty: typ-of t = Some ty
    by (meson assms option.exhaust term-okD2)
  hence typ-ok  $\Theta$  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  $\Theta$  term-ok  $\Theta$  s term-ok  $\Theta$  t typ-of s = typ-of t
  show term-ok  $\Theta$  (mk-eq s t)
proof (rule term-okI)
  have typ-ok  $\Theta$  (the (typ-of t))
    using assms(1) assms(3) term-ok-imp-typ-ok' by blast
  hence typ-ok-sig (sig  $\Theta$ ) (the (typ-of t))
    by simp
  then show term-ok' (sig  $\Theta$ ) (mk-eq s t)
    using assms apply -
    apply (drule term-okD1)+
    apply (cases  $\Theta$  rule: theory-full-exhaust)
    by (auto split: option.splits simp add: tinstT-def)
next
  show typ-of (mk-eq s t)  $\neq$  None
    using assms typ-of-def by (auto dest: term-okD2 simp add: wt-term-def)
qed

lemma typ-of1-decr':  $\neg$  loose-bvar1 t 0  $\implies$  typ-of1 (T#Ts) t = Some  $\tau$   $\implies$ 
typ-of1 Ts (decr 0 t) = Some  $\tau$ 
proof (induction Ts t arbitrary: T  $\tau$  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*: $\neg \text{loose-bvar1 } t \ 0 \implies$
 $\text{typ-of1 } Ts \ (\text{Abs } \tau \ (t \ \$ \ Bv \ 0)) = \text{Some } (\tau \rightarrow \tau') \implies \text{typ-of1 } Ts \ (\text{decr } 0 \ t) = \text{Some}$
 $(\tau \rightarrow \tau')$
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*: $\neg \text{is-dependent } t \implies$
 $\text{typ-of } (\text{Abs } \tau \ (t \ \$ \ Bv \ 0)) = \text{Some } (\tau \rightarrow \tau') \implies \text{typ-of } (\text{decr } 0 \ t) = \text{Some } (\tau \rightarrow$
 $\tau')$
using *typ-of-def is-dependent-def typ-of1-eta-red-step-pre* **by** *simp*

lemma *distinct-add-vars'*: $\text{distinct } acc \implies \text{distinct } (\text{add-vars}' \ t \ acc)$
unfolding *add-vars'-def*
by (*induction t arbitrary: acc*) *auto*

lemma *distinct-add-tvarsT'*: $\text{distinct } acc \implies \text{distinct } (\text{add-tvarsT}' \ T \ acc)$
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'*: $\text{distinct } acc \implies \text{distinct } (\text{add-tvars}' \ t \ acc)$
by (*induction t arbitrary: acc*) (*simp-all add: add-tvars'-def fold-types-def distinct-add-tvarsT'*)

lemma *proved-terms-well-formed-pre*: $\Theta, \Gamma \vdash p \implies \text{typ-of } p = \text{Some } \text{propT} \wedge$
 $\text{term-ok } \Theta \ p$
proof (*induction $\Gamma \ p$ rule: proves.induct*)
case (*axiom A ϱ*)

from *axiom* **have** *ty*: $\text{typ-of1 } \square \ A = \text{Some } \text{propT}$
by (*cases Θ rule: theory-full-exhaust*) (*simp add: wt-term-def typ-of-def*)
let *?l* = $\text{add-tvars}' \ A \ \square$
let *?l'* = $\text{map } (\lambda(v, S) . ((v, S), \varrho \ v \ S)) \ ?l$
have *dist*: $\text{distinct } ?l$
using *distinct-add-tvars'* **by** *simp*
moreover **have** $\forall (v, S) \in \text{set } ?l . \text{typ-ok } \Theta \ (Tv \ v \ S)$
proof–
have $\text{typ-ok } \Theta \ (Tv \ v \ T)$ **if** $(v, T) \in \text{tvs } A$ **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** $\text{set } ?l = \text{tvs } A$
by *auto*

```

ultimately show ?thesis
  by auto
qed
moreover hence  $\forall (v, S) \in \text{set } ?l . \text{has-sort } (\text{osig } (\text{sig } \Theta)) (Tv\ v\ S)\ S$ 
  using axiom.hyps(1) has-sort-Tv-refl' by blast

ultimately have inst-ok  $\Theta\ ?l'$ 
  apply – apply (rule wf-inst-imp-inst-ok)
  using axiom.hyps(1) axiom.hyps(3) by blast+

have simp: tsubst  $A\ \varrho = \text{subst-typ}'\ ?l'\ A$ 
  using dist subst-typ'-simulates-tsubst-gen' by auto

have typ-of1  $\square (tsubst\ A\ \varrho) = \text{Some prop}T$ 
  using tsubst-simulates-subst-typ' axioms-typ-of-propT typ-of1-tsubst ty by fast-
force
hence 1: typ-of1  $\square (\text{subst-typ}'\ ?l'\ A) = \text{Some prop}T$ 
  using simp by simp

from axiom have term-ok' (sig  $\Theta$ )  $A$ 
  by (cases  $\Theta$  rule: theory-full-exhaust) (simp add: wt-term-def)
hence 2: term-ok' (sig  $\Theta$ ) (subst-typ}'\ ?l'\ A)
  using axiom term-ok'-subst-typ' apply (cases  $\Theta$  rule: theory-full-exhaust)
  apply (simp add: list-all-iff wt-term-def typ-of-def)
  by (metis (no-types, lifting) <inst-ok  $\Theta$  (map  $(\lambda(v, S). ((v, S), \varrho\ v\ S))$ ) (add-tvars'
 $A\ \square$ ))>
```

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  $\Gamma\ B\ x\ \tau$ )
hence term-ok' (sig  $\Theta$ )  $B$  and typ-of  $B = \text{Some prop}T$ 
  by (simp-all add: wt-term-def)
show ?case using typ-of-mk-all forall-intro
  term-ok-mk-all[OF  $\langle \text{wf-theory } \Theta \rangle \langle \text{term-ok}' (\text{sig } \Theta) B \rangle$ 
 $\langle \text{typ-of } B = \text{Some prop}T \rangle$  -, of - x]  $\langle \text{wf-type } (\text{sig } \Theta) \tau \rangle$ 
  by auto
next
case (forall-elim  $\Gamma\ \tau\ B\ a$ )
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  $\Gamma\ B\ A$ )

```

```

then show ?case
  by (cases  $\Theta$  rule: wf-theory.cases) (auto simp add: typ-of-def wt-term-def tinstT-def)
next
  case (implies-elim  $\Gamma_1 A B \Gamma_2$ )

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  $\Theta$  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 ( $\beta$ -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  $\Theta$  (mk-eq (Abs T t $ x) (subst-bv x t))
  proof-
    have typ-of (mk-eq (Abs T t $ x) (subst-bv x t))  $\neq$  None
      using 1 by simp

  moreover have term-ok' (sig  $\Theta$ ) (mk-eq (Abs T t $ x) (subst-bv x t))
  proof-
    have term-ok' (sig  $\Theta$ ) (Abs T t $ x)
    using  $\beta$ -conversion.hyps(2)  $\beta$ -conversion.hyps(3) term-ok'.simps(4) wt-term-def
    term-ok-def by blast
    moreover hence term-ok' (sig  $\Theta$ ) (subst-bv x t)
      using subst-bv-def term-ok'-subst-bv1 by auto
    moreover have const-type (sig  $\Theta$ ) STR "Pure.eq"
      = Some ((Tv (Var (STR "'a'", 0)) full-sort)  $\rightarrow$  ((Tv (Var (STR "'a'", 0))
    full-sort)  $\rightarrow$  propT))
      using  $\beta$ -conversion.hyps(1) by (cases  $\Theta$ ) 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  $\Theta$ ) t'
      using  $\beta$ -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  $\Theta$ ) (t'  $\rightarrow$  propT)
      using <wf-theory  $\Theta$ > by (cases  $\Theta$  rule: theory-full-exhaust) auto
    moreover have tinstT (T  $\rightarrow$  (T  $\rightarrow$  propT)) ((Tv (Var (STR "'a'", 0))
    full-sort)  $\rightarrow$  ((Tv (Var (STR "'a'", 0)) full-sort)  $\rightarrow$  propT))

```

unfolding *tinstT-def* **by** *auto*
moreover have *tinstT* ($t' \rightarrow (t' \rightarrow \text{prop}T)$) ((*Tv* (*Var* (*STR* "'a'", 0))
full-sort) \rightarrow ((*Tv* (*Var* (*STR* "'a'", 0)) *full-sort*) \rightarrow *propT*))
unfolding *tinstT-def* **by** *auto*
ultimately show *?thesis* **using** $\langle \text{wf-theory } \Theta \rangle$ **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* τ ($t \ \$ \ Bv \ 0$)) = *Some* ($\tau \rightarrow \tau'$)
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* ($\tau \rightarrow \tau'$)
using *typ-of1-eta-red-step* **by** *blast*

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

have *typ-ok* Θ ($\tau \rightarrow \tau'$)
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* Θ (*Abs* τ ($t \ \$ \ Bv \ 0$))
using *eta(2) tyeta* **by** (*simp add: wt-term-def*)
moreover have *term-ok* Θ (*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* Θ (*mk-eq* (*Abs* τ ($t \ \$ \ Bv \ 0$)) (*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* Θp
using *assms proved-terms-well-formed-pre* **by** *auto*

lemma *forall-intros*:

wf-theory $\Theta \implies \Theta, \Gamma \vdash B \implies \forall (x, \tau) \in \text{set } \text{frees} . (x, \tau) \notin FV \Gamma \wedge \text{typ-ok } \Theta \tau$
 $\implies \Theta, \Gamma \vdash \text{mk-all-list } \text{frees } B$

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]*: $\text{term-ok } \Theta (Fv \text{idn } \tau) = \text{typ-ok } \Theta \tau$

by (*simp add: wt-term-def typ-of-def*)

lemma *typ-of-var[simp]*: $\text{typ-of } (Fv \text{idn } \tau) = \text{Some } \tau$

by (*simp add: typ-of-def*)

lemma *is-closed-Fv[simp]*: $\text{is-closed } (Fv \text{idn } \tau)$ **by** (*simp add: is-open-def*)

corollary *proved-terms-closed*: $\Theta, \Gamma \vdash B \implies \text{is-closed } B$

by (*simp add: proved-terms-well-formed(1) typ-of-imp-closed*)

lemma *not-loose-bvar-bind-fv2*:

$\neg \text{loose-bvar } t \text{ lev} \implies \neg \text{loose-bvar } (\text{bind-fv2 } v \text{ lev } t) (\text{Suc } \text{lev})$

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

lemma *not-loose-bvar-bind-fv2-*:

$\neg \text{loose-bvar } (\text{bind-fv2 } v \text{ lev } t) \text{ lev} \implies \neg \text{loose-bvar } t \text{ lev}$

by (*induction t arbitrary: lev*) (*auto split: if-splits*)

lemma *fold-add-vars'-FV-pre*: $\text{set } (\text{fold } \text{add-vars}' Hs \text{ acc}) = \text{set } \text{acc} \cup FV (\text{set } Hs)$

by (*induction Hs arbitrary: acc*) (*auto simp add: add-vars'-fv-pre*)

corollary *fold-add-vars'-FV[simp]*: $\text{set } (\text{fold } (\text{add-vars}') Hs []) = FV (\text{set } Hs)$

using *fold-add-vars'-FV-pre* **by** *simp*

lemma *forall-intro-vars*:

assumes *wf-theory* $\Theta \Theta, \text{set } Hs \vdash B$

shows $\Theta, \text{set } Hs \vdash \text{forall-intro-vars } B Hs$

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* $\Theta \text{term-ok } \Theta B \text{typ-of } B = \text{Some } \text{propT } \forall (idn, ty) \in \text{set } vs . \text{typ-ok } \Theta ty$

shows $\text{term-ok } \Theta (\text{mk-all-list } vs B) \wedge \text{typ-of } (\text{mk-all-list } vs B) = \text{Some } \text{propT}$

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.prem*s **by** *simp*
then show ?*case* **using** I *s term-ok-mk-all snoc.prem*s(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*) \wedge *typ-of* (*forall-intro-vars B Hs*) =
Some propT
proof-
have 1 : $\forall (idn,ty) \in set (add-vars' B [])$. *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*: $fv (bind-fv (idn, \tau) t) = fv t - \{(idn, \tau)\}$
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 $\forall (idn, \tau) \in set l$. *typ-ok* Θ τ
shows *term-ok* Θ (*mk-all-list l B*) \wedge *typ-of* (*mk-all-list l B*) = *Some propT*
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 : $v = (idn, \tau)$ **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.prem*s 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*: $tvs (bind-fv2 (v, T) lev t) \cup tvsT T = tvs t \cup tvsT T$
by (*induction* (v, T) lev t *rule*: *bind-fv2.induct*) *auto*

lemma *tvs-bind-fv*: $tvs (bind-fv (v, T) t) \cup tvsT T = tvs t \cup tvsT T$
using *tvs-bind-fv2 bind-fv-def* **by** *simp*

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

lemma *tvs-mk-all-list*:

$tvs (mk-all-list vs B) = tvs B \cup tvsT-Set (snd \text{' set } vs)$

proof(*induction vs rule*: *rev-induct*)

case *Nil*

then show *?case* **by** *simp*

next

case (*snoc v vs*)

obtain *idn* τ **where** $v = (idn, \tau)$ **by** *fastforce*

show *?case* **using** *snoc v tvs-mk-all'* **by** (*auto simp add*: *mk-all-list-def*)

qed

lemma *tvs-occs*: $occs v t \implies tvs v \subseteq tvs t$
by (*induction t*) *auto*

lemma *tvs-forall-intro-vars*: $tvs (forall-intro-vars B Hs) = tvs B$

proof–

have $\forall (idn, ty) \in fv B . occs (Fv idn ty) B$

using *fv-occs* **by** *blast*

hence $\forall (idn, ty) \in fv B . tvs (Fv idn ty) \subseteq tvs B$

using *tvs-occs* **by** *blast*

hence $\forall (idn, ty) \in fv B . tvsT ty \subseteq tvs B$

by *simp*

hence $tvsT-Set (snd \text{' fv } B) \subseteq tvs B$

by *fastforce*

hence $tvsT-Set (snd \text{' set } (add-vars' B [])) \subseteq tvs B$

by (*simp add*: *add-vars'-fv*)

thus *?thesis* **using** *tvs-mk-all-list* **by** *auto*

qed

lemma *strip-all-single-var* $B = \text{Some } \tau \implies \text{strip-all-single-body } B \neq B$
using *strip-all-vars-step* **by** *fastforce*

lemma *strip-all-body-unchanged-iff-strip-all-single-body-unchanged*:

$\text{strip-all-body } B = B \iff \text{strip-all-single-body } B = B$

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 $B = B \implies$ *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 $B = B \implies$ *strip-all-body* $B = B$
by (*auto elim!: strip-all-single-body.elims*)

lemma *strip-all-single-var-imp-strip-all-body-single-unchanged*:
strip-all-single-var $B = \text{None} \implies$ *strip-all-single-body* $B = B$
by (*auto elim!: strip-all-single-var.elims*)

lemma *strip-all-single-form*: *strip-all-single-var* $B = \text{Some } \tau$
 \implies *Ct STR "Pure.all"* $((\tau \rightarrow \text{prop}T) \rightarrow \text{prop}T) \$ \text{Abs } \tau$ (*strip-all-single-body* B) = B
by (*auto elim!: strip-all-single-var.elims split: if-splits*)

lemma *proves-strip-all-single*:
assumes $\Theta, \Gamma \vdash B$ *strip-all-single-var* $B = \text{Some } \tau$
typ-of $t = \text{Some } \tau$ *term-ok* Θt
shows $\Theta, \Gamma \vdash$ *subst-bv* t (*strip-all-single-body* B)
proof –
have 1: *Ct STR "Pure.all"* $((\tau \rightarrow \text{prop}T) \rightarrow \text{prop}T) \$ \text{Abs } \tau$ (*strip-all-single-body* B) = B
using *assms(2) strip-all-single-form* **by** *blast*
hence $\Theta, \Gamma \vdash \text{Abs } \tau$ (*strip-all-single-body* B) $\cdot t$
using *assms forall-elim*
proof –
have *has-typ* $t \tau$
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 $\Theta, \Gamma \vdash B$ *strip-all-single-var* $B = \text{Some } \tau$
shows $\Theta, \Gamma \vdash$ *subst-bv* $(Fv x \tau)$ (*strip-all-single-body* B)
proof –
have *ok*: *term-ok* ΘB
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 $s: B = Ct STR "Pure.all" ((\tau \rightarrow propT) \rightarrow propT) \$ Abs \tau (strip-all-single-body B)$
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-all-vars $B = [] \implies strip-all-body B = B$
by (*auto elim!: strip-all-vars.elims*)

lemma *strip-all-vars* $B = (\tau s@[\tau]) \implies strip-all-body B$
 $= strip-all-single-body (Ct STR "Pure.all" ((\tau \rightarrow propT) \rightarrow propT) \$ Abs \tau (strip-all-body B))$
by *simp*

lemma *strip-all-vars-incr-bv*: *strip-all-vars (incr-bv inc lev t) = strip-all-vars t*
by (*induction t arbitrary: lev rule: strip-all-vars.induct*) *auto*

lemma *strip-all-vars-incr-boundvars*: *strip-all-vars (incr-boundvars inc t) = strip-all-vars t*
using *incr-boundvars-def strip-all-vars-incr-bv* **by** *simp*

lemma *strip-all-vars-subst-bv1-Fv*:
strip-all-vars (subst-bv1 B lev (Fv x τ)) = strip-all-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-all-vars (subst-bv (Fv x τ) B) = strip-all-vars B
by (*simp add: strip-all-vars-subst-bv1-Fv subst-bv-def*)

lemma *strip-all-single-var* $B = Some \tau$
 $\implies strip-all-vars (subst-bv (Fv x τ) (strip-all-single-body B)) = tl (strip-all-vars B)$
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-all-vars B) $\Theta, \Gamma \vdash B$*
shows $\Theta, \Gamma \vdash fold (\lambda(x,\tau). subst-bv (Fv x \tau) o strip-all-single-body) (zip xs (strip-all-vars B)) B$

```

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.premis 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.premis proves-strip-all-single-Fv)
  have  $\Theta, \Gamma \vdash \text{fold } (\lambda(x, \tau). \text{subst-bv } (Fv\ x\ \tau) \circ \text{strip-all-single-body})$ 
    (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  $\Theta, \{\} \vdash c \mapsto c$ 
  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*: $\Theta, \Gamma \vdash B$
assumes *a-ok*: *term-ok* Θ *a*
assumes *typ-a*: *typ-of* *a* = *Some* τ
assumes *free*: $(x, \tau) \notin FV \Gamma$
shows $\Theta, \Gamma \vdash \text{subst-term } [((x, \tau), a)] B$

proof–

have *s1*: *mk-all* $x \tau B = Ct STR "Pure.all" ((\tau \rightarrow propT) \rightarrow propT) \$$
Abs $\tau (bind-fv (x, \tau) B)$
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* $\Theta \tau$ **using** *wt-term-def typ-ok-def term-ok-imp-typ-ok*
using *a-ok wf-theory typ-a* **by** *blast*

hence *p1*: $\Theta, \Gamma \vdash \text{mk-all } x \tau B$
using *forall-intro[OF wf-theory B] B typ-a wt-term-def wf-theory*
term-ok-imp-typ-ok free **by** *simp*

have $\Theta, \Gamma \vdash \text{subst-bv } a (bind-fv (x, \tau) B)$
using *forall-elim[of - -] p1 typ-a a-ok proves-strip-all-single*
by (*meson has-typ-iff-typ-of term-ok-def wt-term-def*)

have $\Theta, \Gamma \vdash \text{subst-bv } a ((bind-fv (x, \tau) B))$
using *forall-elim[of - -] p1 typ-a a-ok proves-strip-all-single*
by (*meson has-typ-iff-typ-of term-ok-def wt-term-def*)

thus $\Theta, \Gamma \vdash \text{subst-term } [((x, \tau), a)] B$
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 $\bigwedge p . p \in fv t \implies p \sim = (x, \tau)$
shows *fv* (*subst-term* $[((x, \tau), t)] B$) = *fv* $B - \{(x, \tau)\} \cup fv t$

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*: $\Theta, \Gamma \vdash B$

assumes *vars-ok*: *list-all* (*term-ok* Θ) (*map snd insts*)
assumes *typs-ok*: *list-all* ($\lambda((idx, ty), t) . \text{typ-of } t = \text{Some } ty$) *insts*
assumes *free*: *list-all* ($\lambda((idx, ty), t) . (idx, ty) \notin FV \Gamma$) *insts*
assumes *typ-a*: *typ-of* $a = \text{Some } \tau$
assumes *distinct*: *distinct* (*map fst insts*)
assumes *no-overlap*: $\bigwedge x . x \in (\bigcup t \in \text{snd } ' (set \text{ insts}) . \text{fv } t) \implies x \notin \text{fst } ' (set \text{ insts})$
shows $\Theta, \Gamma \vdash \text{fold } (\lambda \text{single. subst-term } [single]) \text{ insts } B$
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 $\Theta, \Gamma \vdash \text{fold } (\lambda \text{single. subst-term } [single]) (x \# xs) B$
 $\longleftrightarrow \Theta, \Gamma \vdash \text{fold } (\lambda \text{single. subst-term } [single]) xs (\text{subst-term } [x] B)$
by *simp*
moreover have $\Theta, \Gamma \vdash \text{fold } (\lambda \text{single. subst-term } [single]) xs (\text{subst-term } [x] B)$
proof-
have *single*: $\Theta, \Gamma \vdash (\text{subst-term } [x] B)$ **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'*:
 $is\text{-std}\text{-sig } \Sigma \implies \text{term}\text{-ok}' \Sigma t \implies is\text{-closed } st \implies \text{occs } st t \implies \text{term}\text{-ok}' \Sigma st$
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 \times typ) set*)
by (*simp add: finite-prod*)

lemma *implies-intro'-pre*:
assumes *wf-theory* $\Theta \Theta, \Gamma \vdash B \text{ term-ok } \Theta A \text{ typ-of } A = \text{Some propT } A \notin \Gamma$
shows $\Theta, \Gamma \vdash A \longmapsto B$
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 $\Theta, \Gamma \vdash B$ *term-ok* ΘA *typ-of* $A = \text{Some propT } A \in \Gamma$
shows $\Theta, \Gamma \vdash A \mapsto B$
proof–
have $1: \Theta, \Gamma - \{A\} \vdash A \mapsto B$
using *assms proves.implies-intro* **by** (*simp add: wt-term-def*)
have $\Theta, \Gamma - \{A\} - \{A\} \vdash A \mapsto (A \mapsto B)$
using *assms proves.implies-intro*
by (*simp add: 1 implies-intro'-pre*)
moreover have $\Theta, \{A\} \vdash A$
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* $[(x, \tau), Fv y \tau]$ 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* $[(x, \tau), Fv y \tau]$ t) = *typ-of* t
using *typ-of-def* **by** *simp*

lemma *subst-term-preserves-term-ok'[simp]*:
term-ok' Σ (*subst-term* $[(x, \tau), Fv y \tau]$ t) \longleftrightarrow *term-ok'* Σ t
by (*induction* t) *auto*

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

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

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

lemma *rename-free*:
assumes *wf-theory*: *wf-theory* Θ
assumes $B: \Theta, \Gamma \vdash B$
assumes *free*: $(x, \tau) \notin FV \Gamma$
shows $\Theta, \Gamma \vdash$ *subst-term* $[(x, \tau), Fv y \tau]$ 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]*: $tvs (subst-term [(x, \tau), Fv y \tau]) A = tvs A$
by (*induction A*) *auto*

lemma *weaken-proves'*: $\Theta, \Gamma \vdash B \implies term-ok \Theta A \implies typ-of A = Some propT$
 $\implies A \notin \Gamma$

$\implies finite \Gamma$
 $\implies \Theta, insert A \Gamma \vdash B$

proof (*induction ΓB arbitrary: A rule: proves.induct*)
case (*axiom A insts $\Gamma A'$*)
then show *?case using proves.axiom axiom by metis*
next
case (*assume A $\Gamma A'$*)
then show *?case using proves.intros by blast*
next
case (*forall-intro $\Gamma B x \tau$*)

have $\exists y . y \notin fst \text{ ` } (fv A \cup fv B \cup FV \Gamma)$

proof-

have *finite (FV Γ)*
using *finite-fv forall-intro.prem* **by** *auto*
hence *finite (fv A \cup fv B \cup FV Γ)* **by** *simp*
hence *finite (fst ` (fv A \cup fv B \cup FV Γ))* **by** *simp*

thus *?thesis using variant-variable-fresh by blast*

qed

from this obtain y where $y \notin fst \text{ ` } (fv A \cup fv B \cup FV \Gamma)$ **by** *auto*

have *not-in-ren: subst-term [(x, τ), Fv y τ] A $\notin \Gamma$*

proof(*cases (x, τ) \in fv A*)

case *True*

show *?thesis*

apply (*rule not-in-FV-in-fv-not-in[of y τ]*)

apply (*metis (full-types) Un-iff $\langle y \notin fst \text{ ` } (fv A \cup fv B \cup FV \Gamma) \rangle$ 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.prem(3)*)

qed

have *term-ok-ren: term-ok Θ (subst-term [(x, τ), Fv y τ] A)*

using *forall-intro.prem(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.prem* **by** *auto*

hence Θ , *insert* (*subst-term* $[(x, \tau), Fv\ y\ \tau]$ A) $\Gamma \vdash B$
using *forall-intro.IH forall-intro.prem1 forall-intro.prem2 forall-intro.prem3 forall-intro.prem4*
not-in-ren term-ok-ren typ-of-ren **by** *blast*
have Θ , *insert* (*subst-term* $[(x, \tau), Fv\ y\ \tau]$ A) $\Gamma \vdash mk\text{-all}\ x\ \tau\ B$
apply (*rule proves.forall-intro*)
apply (*simp add: forall-intro.hyps(1)*)
using $\langle \Theta, \text{insert} (\text{subst-term} [(x, \tau), Fv\ y\ \tau]) A \Gamma \vdash B \rangle$ **apply** *blast*
subgoal using *subst-term-fv* $\langle (x, \tau) \notin FV\ \Gamma \rangle$ **apply** *simp*
by (*metis Un-iff* $\langle y \notin fst\ ' (fv\ A \cup fv\ B \cup FV\ \Gamma) \rangle$ *fst-conv image-eqI*)
using *forall-intro.hyps(4)* **by** *blast*
hence $\Theta, \Gamma \vdash \text{subst-term} [(x, \tau), Fv\ y\ \tau] A \mapsto mk\text{-all}\ x\ \tau\ B$
using *forall-intro.hyps(1) forall-intro.hyps(2) forall-intro.hyps(3)*
forall-intro.prem1 forall-intro.prem2
implies-intro'-pre local.forall-intro not-in-ren proves.forall-intro
subst-term-preserves-typ-of term-ok-ren **by** *auto*
hence $\Theta, \Gamma \vdash \text{subst-term} [(y, \tau), Fv\ x\ \tau]$
(subst-term $[(x, \tau), Fv\ y\ \tau]) A \mapsto mk\text{-all}\ x\ \tau\ B$
by (*smt Un-iff* $\langle y \notin fst\ ' (fv\ A \cup fv\ B \cup FV\ \Gamma) \rangle$ *forall-intro.hyps(1)*
fst-conv image-eqI rename-free)
hence $\Theta, \Gamma \vdash A \mapsto mk\text{-all}\ x\ \tau\ B$
using *forall-intro proves.forall-intro implies-intro'-pre* **by** *auto*
moreover have $\Theta, \{A\} \vdash A$
using *forall-intro.prem1 local.forall-intro(7) trivial-pre-depr* **by** *blast*
ultimately show *?case*
using *implies-elim* **by** *fastforce*
next
case (*forall-elim* $\Gamma\ \tau\ B\ a$)
then show *?case* **using** *proves.forall-elim* **by** *blast*
next
case (*implies-intro* $\Gamma\ B\ N$)
then show *?case*
proof (*cases A=N*)
case *True*

hence $\Theta, \Gamma - \{N\} \vdash N \mapsto B$

using *implies-intro.hyps(1) implies-intro.hyps(2) implies-intro.hyps(3)*
implies-intro.hyps(4) proves.implies-intro **by** *blast*
hence $\Theta, \Gamma - \{N\} \vdash A \mapsto N \mapsto B$
using *True implies-intro'-pre implies-intro.hyps(1) implies-intro.hyps(3)*
implies-intro.hyps(4) implies-intro.prem1 **by** *blast*
hence $\Theta, \text{insert}\ N\ \Gamma \vdash B$
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) im-*
plies-intro.prem1
trivial-pre-depr **by** (*simp add: implies-intro'-pre2 implies-intro.hyps(1)*)
next
case *False*

hence $s: \text{insert } A (\Gamma - \{N\}) = \text{insert } A \Gamma - \{N\}$ **by** *auto*

have $I: \Theta, \text{insert } A \Gamma \vdash B$
using *implies-intro.premis 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* $\Gamma_1 A' B \Gamma_2$)
show *?case*
using *proves.implies-elim implies-elim* **by** (*metis UnCI Un-insert-left finite-Un*)

next
case (*β -conversion* $\Gamma s T t x$)
then show *?case* **using** *proves. β -conversion* **by** *blast*

next
case (*eta* $t \tau \tau'$)
then show *?case* **using** *proves.eta* **by** *simp*

next
case (*of-class* $c T' T \Gamma$)
then show *?case*
by (*simp add: proves.of-class*)

qed

corollary *weaken-proves*: $\Theta, \Gamma \vdash B \implies \text{term-ok } \Theta A \implies \text{typ-of } A = \text{Some prop } T$
 $\implies \text{finite } \Gamma$
 $\implies \Theta, \text{insert } A \Gamma \vdash B$
using *weaken-proves'* **by** (*metis insert-absorb*)

lemma *weaken-proves-set*: *finite* $\Gamma 2 \implies \Theta, \Gamma \vdash B \implies \forall A \in \Gamma 2 . \text{term-ok } \Theta A \implies$
 $\forall A \in \Gamma 2 . \text{typ-of } A = \text{Some prop } T$
 $\implies \text{finite } \Gamma$
 $\implies \Theta, \Gamma \cup \Gamma 2 \vdash B$
by (*induction* $\Gamma 2$ *arbitrary: Γ rule: finite-induct*) (*use weaken-proves in auto*)

lemma *no-tvsT-imp-subst-typ-unchanged*: *tvsT* $T = \text{empty} \implies \text{subst-typ insts } T = T$
by (*simp add: no-tvsT-imp-tsubsT-unchanged tsubstT-simulates-subst-typ*)

lemma *subst-typ-fv*:
shows *apsnd* (*subst-typ insts*) ' *fv* $B = \text{fv} (\text{subst-typ}' \text{ insts } B)$
by (*induction B*) *auto*

lemma *subst-typ-fv-point*:
assumes $(x, \tau) \in \text{fv } B$
shows $(x, \text{subst-typ insts } \tau) \in \text{fv} (\text{subst-typ}' \text{ insts } B)$
using *subst-typ-fv* **by** (*metis apsnd-conv assms image-eqI*)

```

lemma subst-typ-typ-ok:
  assumes typ-ok-sig  $\Sigma$   $\tau$ 
  assumes list-all (typ-ok-sig  $\Sigma$ ) (map snd insts)
  shows typ-ok-sig  $\Sigma$  (subst-typ insts  $\tau$ )
using assms proof (induction  $\tau$ )
  case (Tv idn  $\tau$ )
  then show ?case
    by (cases lookup ( $\lambda x. x = (idn, \tau)$ ) 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  $\in$  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: $\forall x \in \text{set } Ts . \text{typ-ok } \Theta (\text{subst-typ } insts x)$
using Ty **by** (auto simp add: *typ-ok-def* *list-all-iff* *split*: *option.splits*)
moreover **have** ($\forall x \in \text{set } Ts . \text{typ-ok } \Theta (\text{subst-typ } insts x) =$
 $(\forall x \in \text{set } (\text{map } (\text{subst-typ } insts) Ts) . \text{typ-ok } \Theta x)$ **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) \implies *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) \implies *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 \implies *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
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 \neq 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 $\forall C \in S'. \exists Ss. K C = \text{Some } Ss \wedge \text{list-all2 } (has\text{-sort } (sub, tcs)) Ts Ss$

proof (*rule ballI*)

fix *C* **assume** *C: C \in S'*

show $\exists Ss. K C = \text{Some } Ss \wedge \text{list-all2 } (has\text{-sort } (sub, tcs)) Ts Ss$

proof (*cases C \in 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' \in S class-les sub C' C

by (*metis C class-les-def has-sort-Ty.prem(1) has-sort-Ty.prem(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.prem 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.prem(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.prem(2) C'(1) Ss'(1) n list-all2-nthD
  by fastforce
qed

thus  $\exists Ss. K C = \text{Some } Ss \wedge \text{list-all2 } (\text{has-sort } (\text{sub}, \text{tcs})) \text{ } Ts \text{ } Ss$ 
  using Ss by (simp)
qed
qed
qed
qed

```

```

lemma has-sort-sort-leq: wf-theory  $\Theta \implies \text{has-sort } (\text{osig } (\text{sig } \Theta)) \text{ } T \text{ } S$ 
   $\implies \text{sort-leq } (\text{subclass } (\text{osig } (\text{sig } \Theta))) \text{ } S \text{ } S'$ 
   $\implies \text{has-sort } (\text{osig } (\text{sig } \Theta)) \text{ } T \text{ } S'$ 
by (metis has-sort-sort-leq-osig subclass.elims wf-theory-imp-wf-osig)

```

```

lemma subst-typ-preserves-has-sort:
  assumes wf-theory  $\Theta$ 
  assumes has-sort (osig (sig  $\Theta$ )) T S
  assumes list-all ( $\lambda(\text{idn}, S), T$ ). has-sort (osig (sig  $\Theta$ )) T S insts
  shows has-sort (osig (sig  $\Theta$ )) (subst-typ insts T) S
using asms proof(induction T arbitrary: S)
  case (Ty  $\kappa$  Ts)
  obtain cl tcs where cltcs: osig (sig  $\Theta$ ) = (cl, tcs)
    by fastforce
  moreover obtain K where tcsigs (osig (sig  $\Theta$ ))  $\kappa = \text{Some } K$ 
    using Ty.prem(2) has-sort.simps by auto
  ultimately have mgd: tcs  $\kappa = \text{Some } K$ 
    by simp
  have has-sort (osig (sig  $\Theta$ )) (subst-typ insts (Ty  $\kappa$  Ts)) S
    = has-sort (osig (sig  $\Theta$ )) (Ty  $\kappa$  (map (subst-typ insts) Ts)) S
    by simp
  moreover have has-sort (osig (sig  $\Theta$ )) (Ty  $\kappa$  (map (subst-typ insts) Ts)) S
  proof (subst cltcs, rule has-sort-Ty[of tcs, OF mgd], rule ballI)
  fix C assume C: C  $\in$  S
  obtain Ss where Ss: K C = Some Ss
    using C Ty.prem(2) mgd has-sort.simps cltcs by auto
  have list-all2 (has-sort (osig (sig  $\Theta$ ))) (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.premis(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.premis(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.premis 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.premis(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 $\text{typ-of1} (\text{map} (\text{subst-typ insts}) Ts) (\text{subst-typ}' \text{ insts } t1) =$
 $\text{Some} (\text{subst-typ insts} (RT \rightarrow T))$ **using** *App.IH(1)* **by** *blast*
moreover have $\text{typ-of1} (\text{map} (\text{subst-typ insts}) Ts) (\text{subst-typ}' \text{ insts } t2) = \text{Some}$
 $(\text{subst-typ insts } RT)$
using *App* $\langle \text{typ-of1 } Ts \ t1 = \text{Some} (RT \rightarrow T) \rangle$ *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 $\text{typ-of } t = \text{Some } T$
shows $\text{typ-of} (\text{subst-typ}' \text{ insts } t)$
 $= \text{Some} (\text{subst-typ insts } T)$
using *assms subst-typ-preserves-Some-typ-of1 typ-of-def* **by** *fastforce*

lemma *subst-typ'-incr-bv:*

$\text{subst-typ}' \text{ insts} (\text{incr-bv inc lev } t) = \text{incr-bv inc lev} (\text{subst-typ}' \text{ insts } t)$
by (*induction inc lev t rule: incr-bv.induct*) *auto*

lemma *subst-typ'-incr-boundvars:*

$\text{subst-typ}' \text{ insts} (\text{incr-boundvars lev } t) = \text{incr-boundvars lev} (\text{subst-typ}' \text{ insts } t)$
using *subst-typ'-incr-bv incr-boundvars-def* **by** *simp*

lemma *subst-typ'-subst-bv1:* $\text{subst-typ}' \text{ insts} (\text{subst-bv1 } t \ n \ u)$

$= \text{subst-bv1} (\text{subst-typ}' \text{ insts } t) \ n \ (\text{subst-typ}' \text{ insts } u)$
by (*induction t n u rule: subst-bv1.induct*) (*auto simp add: subst-typ'-incr-boundvars*)

lemma *subst-typ'-subst-bv:* $\text{subst-typ}' \text{ insts} (\text{subst-bv } t \ u)$

$= \text{subst-bv} (\text{subst-typ}' \text{ insts } t) (\text{subst-typ}' \text{ insts } u)$
using *subst-typ'-subst-bv1 subst-bv-def* **by** *simp*

lemma *subst-typ-no-tvsT-unchanged:*

$\forall (f, s) \in \text{set insts} . f \notin \text{tvs } T \implies \text{subst-typ insts } T = T$

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:*

$\forall (f, s) \in \text{set insts} . f \notin \text{tvs } t \implies \text{subst-typ}' \text{ insts } t = t$

by (*induction t*) (*use subst-typ-no-tvsT-unchanged in* $\langle \text{fastforce} \rangle$)

lemma *subst-typ'-preserves-term-ok':*

assumes *wf-theory* Θ

```

assumes inst-ok  $\Theta$  insts
assumes term-ok' (sig  $\Theta$ ) t
shows term-ok' (sig  $\Theta$ ) (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  $\Theta$ 
assumes inst-ok  $\Theta$  insts
assumes term-ok  $\Theta$  t
shows term-ok  $\Theta$  (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  $y \notin \text{fst } ' \text{tvs} T$  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  $y \notin \text{fst } ' \text{tvs } t$  assumes  $y \notin \text{fst } ' \text{tvs} T$   $\tau$ 
shows subst-typ' [((y,S), Tv x S)] ((bind-fv2 (x, subst-typ [((x,S), Tv y S)]  $\tau$ ))
  lev (subst-typ' [((x,S), Tv y S)] t)
  = bind-fv2 (x,  $\tau$ ) 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  $y \notin \text{fst } ' \text{fv } t$ 
shows bind-fv2 (y,  $\tau$ ) i (subst-term [((x,  $\tau$ ), Fv y  $\tau$ )] t)
  = bind-fv2 (x,  $\tau$ ) i t

```

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

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

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

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

lemma *tvar-in-fv-in-tvs*: $(a, \tau) \in \text{fv } B \implies (x, S) \in \text{tvsT } \tau \implies (x, S) \in \text{tvs } B$
by (*induction B*) *auto*

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

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

lemma *subst-typ-rename-tvar-preserves-eq*:
 $(y, S) \notin \text{tvsT } T \implies (y, S) \notin \text{tvsT } \tau \implies$
 $\text{subst-typ } [((x, S), Tv\ y\ S)]\ T = \text{subst-typ } [((x, S), Tv\ y\ S)]\ \tau \implies T = \tau$
proof (*induction T arbitrary: \tau*)
case (*Ty n Ts*)
then show *?case*
proof (*induction \tau*)
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  $\tau$ ) (auto split: if-splits)
  qed

```

lemma *subst-typ'-subst-term-rename-var-swap*:

```

  assumes  $b \notin \text{fst } \text{' } \text{fv } B$ 
  assumes  $(y, S) \notin \text{tvs } B$ 
  assumes  $(y, S) \notin \text{tvs } T \ \tau$ 
  shows  $\text{subst-typ}' [((x, S), \text{Tv } y \ S)] (\text{subst-term } [((a, \tau), \text{Fv } b \ \tau)] B)$ 
    =  $\text{subst-term } [((a, (\text{subst-typ}' [((x, S), \text{Tv } y \ S)] \ \tau)), \text{Fv } b \ (\text{subst-typ}' [((x, S), \text{Tv } y \ S)] \ \tau))]$ 
      ( $\text{subst-typ}' [((x, S), \text{Tv } y \ S)] B$ )
  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*:

```

 $(y, S) \in \text{tvs } T \ \tau \implies (y, S) \notin \text{tvs } t \implies (a, \tau) \notin \text{fv } t$ 
by (induction  $t$ ) auto

```

lemma *tvar-not-in-term-imp-free-not-in-term-set*:

```

finite  $\Gamma \implies (y, S) \in \text{tvs } T \ \tau \implies (y, S) \notin \text{tvs-Set } \Gamma \implies (a, \tau) \notin \text{FV } \Gamma$ 
using tvar-not-in-term-imp-free-not-in-term by simp

```

lemma *inst-var-multiple*:

```

  assumes wf-theory: wf-theory  $\Theta$ 
  assumes  $B: \Theta, \Gamma \vdash B$ 
  assumes vars:  $\forall (x, \tau) \in \text{fst } \text{' } \text{set } \text{insts} . \text{term-ok } \Theta (\text{Fv } x \ \tau)$ 
  assumes a-ok:  $\forall a \in \text{snd } \text{' } \text{set } \text{insts} . \text{term-ok } \Theta a$ 
  assumes typ-a:  $\forall ((-, \tau), a) \in \text{set } \text{insts} . \text{typ-of } a = \text{Some } \tau$ 
  assumes free:  $\forall (v, -) \in \text{set } \text{insts} . v \notin \text{FV } \Gamma$ 
  assumes distinct: distinct (map fst insts)
  assumes finite: finite  $\Gamma$ 
  shows  $\Theta, \Gamma \vdash \text{subst-term } \text{insts } B$ 
proof-
  obtain fresh-idns where fresh-idns:
    length fresh-idns = length insts
     $\forall \text{idn} \in \text{set } \text{fresh-idns} .$ 
       $\text{idn} \notin \text{fst } \text{' } (\text{fv } B \cup (\bigcup t \in \text{snd } \text{' } \text{set } \text{insts} . (\text{fv } t)) \cup (\text{fst } \text{' } \text{set } \text{insts})) \cup \text{fst } \text{' } (\text{FV } \Gamma)$ 
    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*
 $= \text{fold } (\lambda \text{single } \text{acc} . \text{subst-term } [\text{single}] \text{acc}) (\text{zip } (\text{zip } \text{fresh-idns } (\text{map } \text{snd } (\text{map } \text{fst } \text{insts}))) (\text{map } \text{snd } \text{insts}))$
 $(\text{fold } (\lambda \text{single } \text{acc} . \text{subst-term } [\text{single}] \text{acc}) (\text{zip } (\text{map } \text{fst } \text{insts}) (\text{map2 } \text{Fv } \text{fresh-idns } (\text{map } \text{snd } (\text{map } \text{fst } \text{insts})))) \text{B})$
using *fresh-idns distinct subst-term-combine' by simp*

from *fresh-idns vars a-ok typ-a free distinct have 1*:
 $\Theta, \Gamma \vdash (\text{fold } (\lambda \text{single } \text{acc} . \text{subst-term } [\text{single}] \text{acc}) (\text{zip } (\text{map } \text{fst } \text{insts}) (\text{map2 } \text{Fv } \text{fresh-idns } (\text{map } \text{snd } (\text{map } \text{fst } \text{insts})))) \text{B})$
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 : $\Theta, \Gamma \vdash \text{fold } (\lambda \text{single} . \text{subst-term } [\text{single}]) (\text{zip } (\text{map } \text{fst } \text{ys}) (\text{map2 } \text{Fv } \text{xs } (\text{map } \text{snd } (\text{map } \text{fst } \text{ys})))) \text{B}$
apply (*rule snoc.IH*)
subgoal using *snoc.premis(1) by (clarsimp split: prod.splits) (smt UN-I Un-iff fst-conv image-iff)*
using *snoc.premis(2-7) by auto*

moreover obtain *yn n where ynn: fst y = (yn, n) by fastforce*
moreover have $\Theta, \Gamma \vdash \text{subst-term } [(fst y, \text{Fv } x \ n)]$
 $(\text{fold } (\lambda \text{single} . \text{subst-term } [\text{single}]) (\text{zip } (\text{map } \text{fst } (\text{ys})) (\text{map2 } \text{Fv } (\text{xs}) (\text{map } \text{snd } (\text{map } \text{fst } (\text{ys})))))) \text{B})$
apply (*simp only: ynn*)
apply (*rule inst-var[of $\Theta \ \Gamma$ (fold (\lambda \text{single} . \text{subst-term } [\text{single}]) (\text{zip } (\text{map } \text{fst } (\text{ys})) (\text{map2 } \text{Fv } (\text{xs}) (\text{map } \text{snd } (\text{map } \text{fst } (\text{ys})))))) \text{B}) (Fv } x \ n) \ n \ \text{yn}]*)
using *snoc.premis <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.premis(6) ynn by auto*

moreover have $\text{fold } (\lambda \text{single} . \text{subst-term } [\text{single}]) (\text{zip } (\text{map } \text{fst } (\text{ys } @ \ [y])) (\text{map2 } \text{Fv } (\text{xs } @ \ [x]) (\text{map } \text{snd } (\text{map } \text{fst } (\text{ys } @ \ [y]))))) \text{B}$
 $= \text{subst-term } [(fst y, \text{Fv } x \ (\text{snd } (\text{fst } y)))]$
 $(\text{fold } (\lambda \text{single} . \text{subst-term } [\text{single}]) (\text{zip } (\text{map } \text{fst } (\text{ys})) (\text{map2 } \text{Fv } (\text{xs}) (\text{map } \text{snd } (\text{map } \text{fst } (\text{ys})))))) \text{B})$
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-of-y: 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 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-of-y 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*:
 \neg *is-dependent* $t \implies \text{term-ok } \Theta (Abs\ T\ (t\ \$\ Bv\ 0)) \implies \text{term-ok } \Theta (decr\ 0\ t)$
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* $\Theta\ t$
shows $\Theta, \{\} \vdash \text{mk-eq } t\ t$
proof–
have *eq-reflexive-ax* \in *axioms* Θ
using *assms* **by** (*cases* Θ *rule: theory-full-exhaust*) *auto*
moreover obtain τ **where** τ : *typ-of* $t = \text{Some } \tau$ **using** *assms wt-term-def* **by**
auto
moreover hence *typ-ok* $\Theta\ \tau$ **using** *assms term-ok-imp-typ-ok* **by** *blast*
ultimately have $\Theta, \{\} \vdash \text{subst-typ}' [((\text{Var } (STR\ "'a''), 0), \text{full-sort}), \tau]$ *eq-reflexive-ax*
using *axiom-subst-typ' assms* **by** (*simp del: term-ok-def*)
hence $\Theta, \{\} \vdash \text{subst-term} [((\text{Var } (STR\ "'x''), 0), \tau), t]$
 $(\text{subst-typ}' [((\text{Var } (STR\ "'a''), 0), \text{full-sort}), \tau])$ *eq-reflexive-ax*
using τ *assms(1) assms(2) inst-var* **by** *auto*
moreover have $\text{subst-term} [((\text{Var } (STR\ "'x''), 0), \tau), t]$
 $(\text{subst-typ}' [((\text{Var } (STR\ "'a''), 0), \text{full-sort}), \tau])$ *eq-reflexive-ax*
 $= \text{mk-eq } t\ t$
using τ **by** (*simp add: eq-axs-def typ-of-def*)
ultimately show *?thesis*
by *simp*

qed

lemma *unsimp-context*: $\Gamma = \{\} \cup \Gamma$
by *simp*

lemma *proves-eq-reflexive*:
assumes *wf-theory* Θ
assumes *term-ok* $\Theta\ t$
assumes *finite* $\Gamma \forall A \in \Gamma. \text{term-ok } \Theta\ A \forall A \in \Gamma. \text{typ-of } A = \text{Some } \text{prop } T$
shows $\Theta, \Gamma \vdash \text{mk-eq } t\ t$
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* $s = \text{typ-of } t$
shows $\Theta, \{\} \vdash \text{mk-eq } s \ t \mapsto \text{mk-eq } t \ s$

proof–

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

moreover hence *typ-ok* Θ τ **using** *assms* *term-ok-imp-typ-ok* **by** *blast*
ultimately have $\Theta, \{\} \vdash \text{subst-typ}' [((\text{Var } (\text{STR } "'a''", 0), \text{full-sort}), \tau)]$ *eq-symmetric-ax*
using *assms* *axiom-subst-typ'* **by** (*auto simp del: term-ok-def*)
hence $\Theta, \{\} \vdash \text{subst-term} [((\text{Var } (\text{STR } "'x''", 0), \tau), s), ((\text{Var } (\text{STR } "'y''", 0), \tau), t)]$
(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* $s = \text{typ-of } t$
assumes *finite* $\Gamma \forall A \in \Gamma. \text{term-ok } \Theta \ A \ \forall A \in \Gamma. \text{typ-of } A = \text{Some } \text{prop } T$
shows $\Theta, \Gamma \vdash \text{mk-eq } s \ t \mapsto \text{mk-eq } t \ s$
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* $\Gamma \forall A \in \Gamma. \text{term-ok } \Theta \ A \ \forall A \in \Gamma. \text{typ-of } A = \text{Some } \text{prop } T$
shows $\Theta, \Gamma \vdash \text{mk-eq } s \ t \mapsto \text{mk-eq } t \ s$

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* $s = \text{typ-of } t$
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* $s = \text{typ-of } t$
assumes $\Theta, \Gamma \vdash \text{mk-eq } s \ t$
assumes *ctxt*: *finite* $\Gamma \ \forall A \in \Gamma. \text{term-ok } \Theta \ A \ \forall A \in \Gamma. \text{typ-of } A = \text{Some prop } T$
shows $\Theta, \Gamma \vdash \text{mk-eq } t \ s$
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* $s = \text{typ-of } t \ \text{typ-of } t = \text{typ-of } u$
shows $\Theta, \{\} \vdash \text{mk-eq } s \ t \ \mapsto \ \text{mk-eq } t \ u \ \mapsto \ \text{mk-eq } s \ u$

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* $t = \text{Some } \tau$ **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 $\Theta, \{\} \vdash \text{subst-typr}' [((\text{Var } (\text{STR } "'a''", 0), \text{full-sort}), \tau)]$ *eq-transitive-ax*
using *assms* *axiom-subst-typr'* **by** (*auto* *simp* *del*: *term-ok-def*)
hence $\Theta, \{\} \vdash \text{subst-term} [((\text{Var } (\text{STR } "'x''", 0), \tau), s), ((\text{Var } (\text{STR } "'y''", 0), \tau),$
 $t),$
 $((\text{Var } (\text{STR } "'z''", 0), \tau), u)]$
 $(\text{subst-typr}' [((\text{Var } (\text{STR } "'a''", 0), \text{full-sort}), \tau)]$ *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* [(($\text{Var } (\text{STR } "'x''", 0), \tau$), s), (($\text{Var } (\text{STR } "'y''", 0)$,
 τ), t),
 $((\text{Var } (\text{STR } "'z''", 0), \tau), u)]$
 $(\text{subst-typr}' [((\text{Var } (\text{STR } "'a''", 0), \text{full-sort}), \tau)]$ *eq-transitive-ax*)
 $= \text{mk-eq } s \ t \ \mapsto \ \text{mk-eq } t \ u \ \mapsto \ \text{mk-eq } s \ u$
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* $s = \text{typ-of } t$ *typ-of* $t = \text{typ-of } u$
assumes *ctxt*: *finite* $\Gamma \forall A \in \Gamma. \text{term-ok } \Theta A \forall A \in \Gamma. \text{typ-of } A = \text{Some prop } T$
shows $\Theta, \Gamma \vdash \text{mk-eq } s t \mapsto \text{mk-eq } t u \mapsto \text{mk-eq } s u$
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* $\Gamma \forall A \in \Gamma. \text{term-ok } \Theta A \forall A \in \Gamma. \text{typ-of } A = \text{Some prop } T$
shows $\Theta, \Gamma \vdash \text{mk-eq } s t \mapsto \text{mk-eq } t u \mapsto \text{mk-eq } s u$

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* $s = \text{typ-of } t$
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* $t = \text{typ-of } u$
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* $s = \text{typ-of } t$ *typ-of* $t = \text{typ-of } u$
assumes $\Theta, \Gamma \vdash \text{mk-eq } s t$ $\Theta, \Gamma \vdash \text{mk-eq } t u$
assumes *ctxt*: *finite* $\Gamma \forall A \in \Gamma. \text{term-ok } \Theta A \forall A \in \Gamma. \text{typ-of } A = \text{Some prop } T$
shows $\Theta, \Gamma \vdash \text{mk-eq } s u$

proof–

note $1 = \text{proves-eq-transitive}[OF \text{ assms}(1-6), \text{ of } \Gamma]$
note $2 = \text{proves.implies-elim}[OF 1 \text{ assms}(7)]$
note $3 = \text{proves.implies-elim}[OF 2 \text{ assms}(8)]$
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* $A = \text{Some prop } T$
assumes B : *term-ok* Θ B *typ-of* $B = \text{Some prop } T$
shows $\Theta, \{\} \vdash (A \mapsto B) \mapsto (B \mapsto A) \mapsto \text{mk-eq } A B$

proof–

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

have $eq\text{-intr}\text{-}ax \in axioms \Theta$
using thy **by** ($cases \Theta$ $rule: theory\text{-}full\text{-}exhaust$) $auto$

hence $1: \Theta, \{\} \vdash eq\text{-intr}\text{-}ax$
by ($simp$ $add: axiom' thy$)

hence $\Theta, \{\} \vdash subst\text{-}term [((Var (STR "A", 0), propT), A), ((Var (STR "B", 0), propT), B)]$
 $eq\text{-intr}\text{-}ax$
using $assms$ $term\text{-}ok\text{-}var$ $propT\text{-}ok$ **by** ($fastforce$ $intro!: inst\text{-}var\text{-}multiple$ $simp$ $add: eq\text{-intr}\text{-}ax\text{-}def$)
thus $?thesis$ **using** $assms$ **by** ($simp$ $add: eq\text{-}axs\text{-}def$ $typ\text{-}of\text{-}def$)
qed

lemma $proves\text{-}eq\text{-}intr$:
assumes $thy: wf\text{-}theory \Theta$
assumes $A: term\text{-}ok \Theta A$ $typ\text{-}of A = Some propT$
assumes $B: term\text{-}ok \Theta B$ $typ\text{-}of B = Some propT$
assumes $ctxt: finite \Gamma \forall A \in \Gamma. term\text{-}ok \Theta A \forall A \in \Gamma. typ\text{-}of A = Some propT$
shows $\Theta, \Gamma \vdash (A \mapsto B) \mapsto (B \mapsto A) \mapsto mk\text{-}eq A B$
by ($subst$ $unsimp\text{-}context$) (use $assms$ $proves\text{-}eq\text{-}intr\text{-}pre$ $weaken\text{-}proves\text{-}set$ **in** $blast$)

lemma $proves\text{-}eq\text{-}intr\text{-}rule$:
assumes $thy: wf\text{-}theory \Theta$
assumes $A: term\text{-}ok \Theta A$ $typ\text{-}of A = Some propT$
assumes $B: term\text{-}ok \Theta B$ $typ\text{-}of B = Some propT$
assumes $\Theta, \Gamma \vdash (A \mapsto B) \Theta, \Gamma \vdash (B \mapsto A)$
assumes $ctxt: finite \Gamma \forall A \in \Gamma. term\text{-}ok \Theta A \forall A \in \Gamma. typ\text{-}of A = Some propT$
shows $\Theta, \Gamma \vdash mk\text{-}eq A B$

proof–
note $1 = proves\text{-}eq\text{-}intr[OF assms(1–5), of \Gamma]$
note $2 = proves.\text{implies}\text{-}elim[OF 1 assms(6)]$
note $3 = proves.\text{implies}\text{-}elim[OF 2 assms(7)]$
thus $?thesis$ **using** $ctxt$ **by** $simp$
qed

lemma $proves\text{-}eq\text{-}elim\text{-}pre$:
assumes $thy: wf\text{-}theory \Theta$
assumes $A: term\text{-}ok \Theta A$ $typ\text{-}of A = Some propT$
assumes $B: term\text{-}ok \Theta B$ $typ\text{-}of B = Some propT$
shows $\Theta, \{\} \vdash mk\text{-}eq A B \mapsto A \mapsto B$

proof–
have $closed: is\text{-}closed A$ $is\text{-}closed B$
by ($simp\text{-}all$ $add: assms(3)$ $assms(5)$ $typ\text{-}of\text{-}imp\text{-}closed$)
have $eq\text{-}elim\text{-}ax \in axioms \Theta$
using thy **by** ($cases \Theta$ $rule: theory\text{-}full\text{-}exhaust$) $auto$
hence $1: \Theta, \{\} \vdash eq\text{-}elim\text{-}ax$
by ($simp$ $add: axiom' thy$)
hence $\Theta, \{\} \vdash subst\text{-}term [((Var (STR "A", 0), propT), A), ((Var (STR "B", 0), propT), B)]$

```

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  $\Theta$ 
assumes A: term-ok  $\Theta$  A typ-of A = Some propT
assumes B: term-ok  $\Theta$  B typ-of B = Some propT
assumes ctxt: finite  $\Gamma \forall A \in \Gamma. \text{term-ok } \Theta A \forall A \in \Gamma. \text{typ-of } A = \text{Some propT}$ 
shows  $\Theta, \Gamma \vdash \text{mk-eq } A B \mapsto A \mapsto B$ 
  by (subst unsimp-context) (use assms proves-eq-elim-pre weaken-proves-set in
blast)

```

lemma *proves-eq-elim-rule:*

```

assumes thy: wf-theory  $\Theta$ 
assumes A: term-ok  $\Theta$  A typ-of A = Some propT
assumes B: term-ok  $\Theta$  B typ-of B = Some propT
assumes  $\Theta, \Gamma \vdash \text{mk-eq } A B$ 
assumes ctxt: finite  $\Gamma \forall A \in \Gamma. \text{term-ok } \Theta A \forall A \in \Gamma. \text{typ-of } A = \text{Some propT}$ 
shows  $\Theta, \Gamma \vdash A \mapsto B$ 
  using proves.implies-elim[OF proves-eq-elim[OF assms(1-5)]] assms(6), of  $\Gamma$ ,
OF ctxt] by simp

```

lemma *proves-eq-elim2-rule:*

```

assumes thy: wf-theory  $\Theta$ 
assumes A: term-ok  $\Theta$  A typ-of A = Some propT
assumes B: term-ok  $\Theta$  B typ-of B = Some propT
assumes  $\Theta, \Gamma \vdash \text{mk-eq } A B$ 
assumes ctxt: finite  $\Gamma \forall A \in \Gamma. \text{term-ok } \Theta A \forall A \in \Gamma. \text{typ-of } A = \text{Some propT}$ 
shows  $\Theta, \Gamma \vdash B \mapsto A$ 

```

proof–

```

  have  $\Theta, \Gamma \vdash \text{mk-eq } B A$ 
  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  $\Theta$ 
assumes f: term-ok  $\Theta$  f typ-of f = Some ( $\tau \rightarrow \tau'$ )
assumes g: term-ok  $\Theta$  g typ-of g = Some ( $\tau \rightarrow \tau'$ )
assumes x: term-ok  $\Theta$  x typ-of x = Some  $\tau$ 
assumes y: term-ok  $\Theta$  y typ-of y = Some  $\tau$ 
shows  $\Theta, \{\} \vdash \text{mk-eq } f g \mapsto \text{mk-eq } x y \mapsto \text{mk-eq } (f \$ x) (g \$ y)$ 

```

proof–

```

  have ok: typ-ok  $\Theta$   $\tau$  typ-ok  $\Theta$  ( $\tau \rightarrow \tau'$ ) typ-ok  $\Theta$   $\tau'$ 

```

using *term-ok-betapply term-ok-imp-typ-ok thy typ-of-betapply thy x f* **by** *blast+*
have *eq-combination-ax* \in *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-betapply* **by** *meson+*
ultimately **have** *1*: $\Theta, \{\}$ \vdash *subst-typ'*
 $[[(\text{Var } (\text{STR } "'a''', 0), \text{full-sort}), \tau), ((\text{Var } (\text{STR } "'b''', 0), \text{full-sort}), \tau')]$
eq-combination-ax
using *assms axiom-subst-typ'* **by** (*simp del: term-ok-def*)
hence $\Theta, \{\}$ \vdash *subst-term*
 $[[(\text{Var } (\text{STR } "'f''', 0), \tau \rightarrow \tau'), f), ((\text{Var } (\text{STR } "'g''', 0), \tau \rightarrow \tau'), g),$
 $((\text{Var } (\text{STR } "'x''', 0), \tau), x), ((\text{Var } (\text{STR } "'y''', 0), \tau), y)]$
 $(\text{subst-typ}' [((\text{Var } (\text{STR } "'a''', 0), \text{full-sort}), \tau), ((\text{Var } (\text{STR } "'b''', 0), \text{full-sort}),$
 $\tau')]$
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* $f = \text{Some } (\tau \rightarrow \tau')$
assumes *g: term-ok* Θ *g typ-of* $g = \text{Some } (\tau \rightarrow \tau')$
assumes *x: term-ok* Θ *x typ-of* $x = \text{Some } \tau$
assumes *y: term-ok* Θ *y typ-of* $y = \text{Some } \tau$
assumes *ctxt: finite* $\Gamma \forall A \in \Gamma. \text{term-ok } \Theta A \forall A \in \Gamma. \text{typ-of } A = \text{Some prop}T$
shows $\Theta, \Gamma \vdash \text{mk-eq } f g \mapsto \text{mk-eq } x y \mapsto \text{mk-eq } (f \$ x) (g \$ y)$
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* $f = \text{Some } (\tau \rightarrow \tau')$
assumes *g: term-ok* Θ *g typ-of* $g = \text{Some } (\tau \rightarrow \tau')$
assumes *x: term-ok* Θ *x typ-of* $x = \text{Some } \tau$
assumes *y: term-ok* Θ *y typ-of* $y = \text{Some } \tau$
assumes $\Theta, \Gamma \vdash \text{mk-eq } f g \Theta, \Gamma \vdash \text{mk-eq } x y$
assumes *ctxt: finite* $\Gamma \forall A \in \Gamma. \text{term-ok } \Theta A \forall A \in \Gamma. \text{typ-of } A = \text{Some prop}T$
shows $\Theta, \Gamma \vdash \text{mk-eq } (f \$ x) (g \$ y)$

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 $\Theta, \Gamma \vdash \text{mk-eq } f \ g \ \Theta, \Gamma \vdash \text{mk-eq } x \ y$
assumes *f*: *typ-of* *f* = *Some* ($\tau \rightarrow \tau'$)
assumes *x*: *typ-of* *x* = *Some* τ
assumes *ctxt*: *finite* $\Gamma \ \forall A \in \Gamma. \text{term-ok } \Theta \ A \ \forall A \in \Gamma. \text{typ-of } A = \text{Some propT}$
shows $\Theta, \Gamma \vdash \text{mk-eq } (f \ \$ \ x) \ (g \ \$ \ y)$
proof–
have *ok-Apps*: *term-ok* $\Theta \ (\text{mk-eq } f \ g) \ \text{term-ok } \Theta \ (\text{mk-eq } x \ y)$
using *assms*(2–3) *proved-terms-well-formed-pre* **by** *auto*
hence *tyy*: *typ-of* *y* = *Some* τ **and** *tyg*: *typ-of* *g* = *Some* ($\tau \rightarrow \tau'$)
using *term-ok-mk-eq-same-typ* *thy* *x* *f* *term-okD1* **by** *metis+*
moreover **have** *term-ok* $\Theta \ x \ \text{term-ok } \Theta \ y \ \text{term-ok } \Theta \ f \ \text{term-ok } \Theta \ g$
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* $\Theta \ A \ \text{typ-of } A = \text{Some propT}$
assumes *B*: *term-ok* $\Theta \ B \ \text{typ-of } B = \text{Some propT}$
assumes *eq*: $\Theta, \Gamma \vdash \text{mk-eq } A \ B$
assumes *pA*: $\Theta, \Gamma \vdash A$
assumes *ctxt*: *finite* $\Gamma \ \forall A \in \Gamma. \text{term-ok } \Theta \ A \ \forall A \in \Gamma. \text{typ-of } A = \text{Some propT}$
shows $\Theta, \Gamma \vdash B$
proof–
have $\Theta, \Gamma \vdash A \ \mapsto \ B$ **using** *proves-eq-elim-rule*[*OF* *assms*(1–5) *eq* *ctxt*] .
thus $\Theta, \Gamma \vdash B$ **using** *proves.implies-elim* *pA* **by** *fastforce*
qed

lemma *proves-eq-mp-rule-better*:
assumes *thy*: *wf-theory* Θ
assumes *eq*: $\Theta, \Gamma \vdash \text{mk-eq } A \ B$
assumes *pA*: $\Theta, \Gamma \vdash A$
assumes *ctxt*: *finite* $\Gamma \ \forall A \in \Gamma. \text{term-ok } \Theta \ A \ \forall A \in \Gamma. \text{typ-of } A = \text{Some propT}$
shows $\Theta, \Gamma \vdash B$
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* $\Theta \ x \ \text{typ-of } x = \text{Some } \tau$
assumes *y*: *term-ok* $\Theta \ y \ \text{typ-of } y = \text{Some } \tau$
assumes *P*: *term-ok* $\Theta \ P \ \text{typ-of } P = \text{Some } (\tau \rightarrow \text{propT})$
assumes *ctxt*: *finite* $\Gamma \ \forall A \in \Gamma. \text{term-ok } \Theta \ A \ \forall A \in \Gamma. \text{typ-of } A = \text{Some propT}$
assumes *eq*: $\Theta, \Gamma \vdash \text{mk-eq } x \ y$
shows $\Theta, \Gamma \vdash \text{mk-eq } (P \ \$ \ x) \ (P \ \$ \ y)$
proof–

have $\Theta, \Gamma \vdash mk\text{-}eq\ P\ P$ **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 $(Abs\ T\ t)$ $\Theta, \Gamma \vdash (Abs\ T\ t)\ \$\ x$
assumes *x: term-ok* Θ $x\ typ\text{-}of\ x = Some\ T$
assumes *ctxt: finite* $\Gamma \forall A \in \Gamma. term\text{-}ok\ \Theta\ A \forall A \in \Gamma. typ\text{-}of\ A = Some\ propT$
shows $\Theta, \Gamma \vdash subst\text{-}bv\ x\ t$
proof–
have $\Theta, \Gamma \vdash mk\text{-}eq\ ((Abs\ T\ t)\ \$\ x)\ (subst\text{-}bv\ x\ t)$
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* $(subst\text{-}bv\ x\ t) = Some\ propT$
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 $(subst\text{-}bv\ x\ t)$
proof–
have *term-ok'* $(sig\ \Theta)\ t$
using *assms(2) term-ok'.simps(5) wt-term-def term-ok-def* **by** *blast*
hence *term-ok'* $(sig\ \Theta)\ (subst\text{-}bv\ x\ t)$
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:* $\Theta, \Gamma \vdash mk\text{-}eq\ f\ g\ typ\text{-}of\ f = Some\ (\tau \rightarrow \tau')$
assumes *type: typ-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 (Ct\ STR\ "Pure.all" ((\tau \rightarrow propT) \rightarrow propT)\ \$\$
 $(Abs\ \tau\ (mk\text{-}eq'\ \tau'\ (f\ \$\ Bv\ 0)\ (g\ \$\ Bv\ 0))))$
proof–
have *term-ok:* *term-ok* Θ $(mk\text{-}eq\ f\ g)$
using *eq(1) proved-terms-well-formed-pre* **by** *blast*
hence *term-ok': term-ok* Θ $f\ term\text{-}ok\ \Theta\ g$
apply *(simp add: eq(2) wt-term-def)*

using $\langle \text{term-ok } \Theta \text{ (mk-eq } f \ g) \rangle \text{ wt-term-def typ-of-def term-ok-app-eqD}$ **by** *blast*
hence $\text{typ-of } f = \text{typ-of } g$
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 $\text{type}' : \text{typ-of } g = \text{Some } (\tau \rightarrow \tau')$
using $\text{eq}(2)$ **by** *simp*

obtain x **where** $x \notin \text{fst } \langle \text{fv } (mk\text{-eq } f \ g) \cup \text{FV } \Gamma \rangle$
using *finite-fv finite-FV infinite-fv-UNIV variant-variable-fresh ctxt*
by (*meson finite-Un finite-imageI*)
hence $\text{free} : (x, \tau) \notin \text{fv } (mk\text{-eq } f \ g) \cup \text{FV } \Gamma$
by *force*
hence $\Theta, \Gamma \vdash \text{mk-eq } (Fv \ x \ \tau) \ (Fv \ x \ \tau)$
using *ctxt proves-eq-reflexive term-ok-var thy type* **by** *presburger*
hence $\Theta, \Gamma \vdash \text{mk-eq } (f \ \$ \ Fv \ x \ \tau) \ (g \ \$ \ Fv \ x \ \tau)$
apply –
apply (*rule proves-eq-combination-rule*[**where** $\tau' = \tau$])
using *assms term-ok' type'* **by** (*simp-all del: term-ok-def*)
hence $\Theta, \Gamma \vdash \text{mk-all } x \ \tau \ (\text{mk-eq } (f \ \$ \ Fv \ x \ \tau) \ (g \ \$ \ Fv \ x \ \tau))$
apply –
apply (*rule proves.forall-intro*)
using *thy eq type free* **by** *simp-all*
moreover **have** $\text{mk-all } x \ \tau \ (\text{mk-eq } (f \ \$ \ Fv \ x \ \tau) \ (g \ \$ \ Fv \ x \ \tau))$
 $= (Ct \ STR \ "Pure.all'' \ ((\tau \rightarrow \text{prop}T) \rightarrow \text{prop}T) \ \$$
 $\ (\text{Abs } \tau \ (\text{mk-eq}' \ \tau' \ (f \ \$ \ Bv \ 0) \ (g \ \$ \ Bv \ 0))))$
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 $\text{eq} : \Theta, \Gamma \vdash \text{mk-eq } f \ g \ \text{typ-of } f = \text{Some } (\tau \rightarrow \tau')$
assumes *type: typ-ok* $\Theta \ \tau$
assumes *ctxt: finite* $\Gamma \ \forall A \in \Gamma. \text{term-ok } \Theta \ A \ \forall A \in \Gamma. \text{typ-of } A = \text{Some } \text{prop}T$
shows $\Theta, \Gamma \vdash \text{mk-eq } (\text{Abs } \tau \ (f \ \$ \ Bv \ 0)) \ (\text{Abs } \tau \ (g \ \$ \ Bv \ 0))$

proof–

have *ok: term-ok* $\Theta \ f \ \text{term-ok } \Theta \ g$
using $\text{eq}(1)$ *proved-terms-well-formed*(2) *term-ok-mk-eqD* **by** *blast+*
have *g-ty: typ-of* $g = \text{Some } (\tau \rightarrow \tau')$
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 $\text{eq}(2)$ *typ-of-imp-closed* **by** *blast+*

have *ok': term-ok* $\Theta \ (\text{Abs } \tau \ (f \ \$ \ Bv \ 0)) \ \text{term-ok } \Theta \ (\text{Abs } \tau \ (g \ \$ \ Bv \ 0))$
using *type term-ok-eta-expand ok thy eq(2) g-ty* **by** *blast+*

```

have ok-ind: wf-term (sig  $\Theta$ ) f wf-term (sig  $\Theta$ ) 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  $\Gamma$ ]
note 2 = proves.eta[OF thy ok-ind(2) typ-of-imp-has-typ[OF g-ty], of  $\Gamma$ ]

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:  $\Theta, \Gamma \vdash$  mk-eq g (Abs  $\tau$  (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:  $\Theta, \Gamma \vdash$  mk-eq (Abs  $\tau$  (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  $\Theta$ 
  assumes ctxt: finite  $\Gamma \forall A \in \Gamma .$  term-ok  $\Theta$  A  $\forall A \in \Gamma .$  typ-of A = Some propT
  assumes eq:  $\Theta, \Gamma \vdash$  mk-eq (Abs  $\tau$  f) (Abs  $\tau$  g) typ-of (Abs  $\tau$  f) = Some ( $\tau \rightarrow$ 
 $\tau'$ )
  assumes x: term-ok  $\Theta$  x typ-of x = Some  $\tau$ 
  assumes ctxt: finite  $\Gamma \forall A \in \Gamma .$  term-ok  $\Theta$  A  $\forall A \in \Gamma .$  typ-of A = Some propT
  shows  $\Theta, \Gamma \vdash$  mk-eq (subst-bv x f) (subst-bv x g)
proof-
  have term-ok  $\Theta$  (mk-eq (Abs  $\tau$  f) (Abs  $\tau$  g))
    using eq(1) proved-terms-well-formed(2) by blast
  hence term-ok  $\Theta$  (Abs  $\tau$  f) term-ok  $\Theta$  (Abs  $\tau$  g)
    using term-ok-mk-eqD by blast+
  hence typ-of (Abs  $\tau$  f) = typ-of (Abs  $\tau$  g)
    using thy  $\langle$ term-ok  $\Theta$  (mk-eq (Abs  $\tau$  f) (Abs  $\tau$  g)) $\rangle$  by (cases  $\Theta$  rule: theory-full-exhaust)
    (auto simp add: tinstT-def typ-of-def wt-term-def bind-eq-Some-conv)
  hence typ-of (Abs  $\tau$  g) = Some ( $\tau \rightarrow \tau'$ )
    using eq(2) by simp

have  $\Theta, \Gamma \vdash$  mk-eq x x
  by (simp add: ctxt proves-eq-reflexive thy x(1) del: term-ok-def)
  hence 1:  $\Theta, \Gamma \vdash$  mk-eq (Abs  $\tau$  f $ x) (Abs  $\tau$  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 $\langle \text{term-ok } \Theta (Abs \tau g \$ x) \rangle \langle \text{term-ok } \Theta (Abs \tau g) \rangle \langle \text{typ-of } (Abs \tau f \$ x) \rangle$
 $= \text{typ-of } (Abs \tau g \$ x) \rangle$
 $\langle \text{typ-of } (Abs \tau f \$ x) = \text{typ-of } (\text{subst-bv } x f) \rangle \langle \text{typ-of } (Abs \tau g) = \text{Some } (\tau \rightarrow \tau') \rangle$
 $\langle \text{typ-of } (\text{subst-bv } x f) = \text{Some } \tau' \rangle \text{betapply.simps}(1) \text{subst-bv-def term-ok'.simps}(5)$
 $\text{term-ok'-subst-bv1 wt-term-def typ-of-betapply } x(1) x(2)$
by (*meson 3 proved-terms-well-formed*(2) *term-ok-mk-eqD*)

have $\text{typ-of } (\text{subst-bv } x f) = \text{typ-of } (Abs \tau g \$ x)$
using $\langle \text{typ-of } (Abs \tau f \$ x) = \text{typ-of } (Abs \tau g \$ x) \rangle$
 $\langle \text{typ-of } (Abs \tau f \$ x) = \text{typ-of } (\text{subst-bv } x f) \rangle$ **by** *auto*

have $\text{typ-of } (Abs \tau g \$ x) = \text{typ-of } (\text{subst-bv } x g)$
using $\langle \text{typ-of } (Abs \tau f) = \text{typ-of } (Abs \tau g) \rangle \text{eq}(2) \text{typ-of-betapply typ-of-def } x(2)$ **by** *auto*

have $c1: \Theta, \Gamma \vdash \text{mk-eq } (\text{subst-bv } x f) (Abs \tau g \$ x)$
apply (*rule proves-eq-transitive-rule*[**where** $t = Abs \tau f \$ x$];
(use assms 1 2 $\langle \text{term-ok } \Theta (\text{subst-bv } x f) \rangle$ in $\langle \text{solves simp} \rangle$)?)
using $\langle \text{term-ok } \Theta (Abs \tau f \$ x) \rangle$ **apply** *blast*
using $\langle \text{term-ok } \Theta (Abs \tau g \$ x) \rangle$ **apply** *blast*
using $\langle \text{typ-of } (Abs \tau f \$ x) = \text{typ-of } (\text{subst-bv } x f) \rangle$ **apply** *simp*
using $\langle \text{typ-of } (Abs \tau f \$ x) = \text{typ-of } (Abs \tau g \$ x) \rangle$ **apply** *blast*
done

show *?thesis*

apply (*rule proves-eq-transitive-rule*[**where** $t = Abs \tau g \$ x$];
(use assms 1 2 $\langle \text{term-ok } \Theta (\text{subst-bv } x f) \rangle$ in $\langle \text{solves simp} \rangle$)?)
using $\langle \text{term-ok } \Theta (Abs \tau g \$ x) \rangle$
 $\langle \text{term-ok } \Theta (\text{subst-bv } x g) \rangle$
 $\langle \text{typ-of } (\text{subst-bv } x f) = \text{typ-of } (Abs \tau g \$ x) \rangle$
 $\langle \text{typ-of } (Abs \tau g \$ x) = \text{typ-of } (\text{subst-bv } x g) \rangle$
 $\langle \Theta, \Gamma \vdash \text{mk-eq } (\text{subst-bv } x f) (Abs \tau g \$ x) \rangle$
 $\langle \Theta, \Gamma \vdash \text{mk-eq } (Abs \tau g \$ x) (\text{subst-bv } x g) \rangle$ **by** *simp-all*

qed

lemma *proves-descend-abs-rule*:

assumes *thy: wf-theory* Θ
assumes *eq*: $\Theta, \Gamma \vdash \text{mk-eq } (Abs \tau' (\text{bind-fv } (x, \tau') s)) (Abs \tau' (\text{bind-fv } (x, \tau') t))$
is-closed s is-closed t
assumes $x: (x, \tau') \notin FV \Gamma \text{typ-ok } \Theta \tau'$
assumes *ctxt*: *finite* $\Gamma \forall A \in \Gamma. \text{term-ok } \Theta A \forall A \in \Gamma. \text{typ-of } A = \text{Some prop } T$
shows $\Theta, \Gamma \vdash \text{mk-eq } s t$

proof–

have *abs-ok*: $\text{term-ok } \Theta (Abs \text{fv } x \tau' s) \text{term-ok } \Theta (Abs \text{fv } x \tau' t)$
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** $\tau 1: \text{typ-of } (Abs \text{fv } x \tau' s) = \text{Some } (\tau' \rightarrow \tau)$
by (*smt eq proved-terms-well-formed-pre typ-of1-split-App-obtains typ-of-Abs-body-typp'*)

typ-of-def)
hence $\tau 2$: *typ-of* (*Abs-fv* $x \tau' t$) = *Some* ($\tau' \rightarrow \tau$)
by (*metis eq(1) proved-terms-well-formed(2) term-ok-mk-eq-same-typ thy*)

have *add-param*: $\Theta, \Gamma \vdash \text{mk-eq}$
(*Abs* $\tau' (\text{bind-fv } (x, \tau') s) \$ \text{Fv } x \tau'$)
(*Abs* $\tau' (\text{bind-fv } (x, \tau') t) \$ \text{Fv } x \tau'$)
apply(*rule proves-eq-combination-rule; use assms abs-ok $\tau 1 \tau 2$ in \langle (solves simp)? \rangle*)
using *proves-eq-reflexive term-ok-var thy x(2) wt-term-def ctxt by blast+*

have βs : $\Theta, \Gamma \vdash \text{mk-eq}$
(*Abs* $\tau' (\text{bind-fv } (x, \tau') s) \$ \text{Fv } x \tau'$)
(*subst-bv* (*Fv* $x \tau'$) (*bind-fv* (x, τ') s))
by (*rule proves. β -conversion; use assms abs-ok $\tau 1 \tau 2$ in \langle (solves \langle simp add: wt-term-def \rangle ? \rangle*)

moreover **have** *simps*: *subst-bv* (*Fv* $x \tau'$) (*bind-fv* (x, τ') s) = s
using *subst-bv-bind-fv typ-of-imp-closed eq(2) by blast*
ultimately **have** βs : $\Theta, \Gamma \vdash \text{mk-eq}$ (*Abs* $\tau' (\text{bind-fv } (x, \tau') s) \$ \text{Fv } x \tau'$) s
by *simp*

have $t1$: *term-ok* Θs
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* $x \tau' s \$ \text{term.Fv } x \tau'$)
using βs \langle *term-ok* Θs \rangle *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*: $\Theta, \Gamma \vdash \text{mk-eq}$ s (*Abs* $\tau' (\text{bind-fv } (x, \tau') s) \$ \text{Fv } x \tau'$)
apply (*rule proves-eq-symmetric-rule; use assms abs-ok $\tau 1 \tau 2 t1 t2$ in \langle (solves simp)? \rangle*)
using βs *proved-terms-well-formed(2) term-ok-mk-eq-same-typ thy apply blast*
using βs **by** *simp*

have βt : $\Theta, \Gamma \vdash \text{mk-eq}$
(*Abs* $\tau' (\text{bind-fv } (x, \tau') t) \$ \text{Fv } x \tau'$)
(*subst-bv* (*Fv* $x \tau'$) (*bind-fv* (x, τ') t))
by (*rule proves. β -conversion; use assms abs-ok $\tau 1 \tau 2 t1 t2$ in \langle (solves \langle simp add: wt-term-def \rangle ? \rangle*)

moreover **have** *simpt*: *subst-bv* (*Fv* $x \tau'$) (*bind-fv* (x, τ') t) = t
using *subst-bv-bind-fv typ-of-imp-closed eq(3) by blast*
ultimately **have** βt : $\Theta, \Gamma \vdash \text{mk-eq}$ (*Abs* $\tau' (\text{bind-fv } (x, \tau') t) \$ \text{Fv } x \tau'$) t
by *simp*

have $t3$: *term-ok* Θ (*Abs-fv* $x \tau' t \$ \text{term.Fv } x \tau'$)
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 t_4 : $\text{typ-of } s = \text{typ-of } (\text{Abs-fv } x \ \tau' \ t \ \$ \ \text{term.Fv } x \ \tau')$
by (*metis* βs *add-param* *proved-terms-well-formed*(2) *term-ok-mk-eq-same-typ* *thy*)
have t_5 : $\text{typ-of } s = \text{typ-of } (\text{Abs-fv } x \ \tau' \ s \ \$ \ \text{Fv } x \ \tau')$
using βs -*rev* *proved-terms-well-formed*(2) *term-ok-mk-eq-same-typ* *thy* **by** *blast*
have t_6 : $\text{typ-of } (\text{Abs-fv } x \ \tau' \ s \ \$ \ \text{Fv } x \ \tau') = \text{typ-of } (\text{Abs-fv } x \ \tau' \ t \ \$ \ \text{term.Fv } x \ \tau')$
using t_4 t_5 **by** *auto*
have *half*: $\Theta, \Gamma \vdash \text{mk-eq } s \ (\text{Abs } \tau' \ (\text{bind-fv } (x, \tau') \ t) \ \$ \ \text{Fv } x \ \tau')$
apply (*rule* *proves-eq-transitive-rule*[**where** $t = \text{Abs } \tau' \ (\text{bind-fv } (x, \tau') \ s) \ \$ \ \text{Fv } x \ \tau'$]
; *use* *assms* *abs-ok* $\tau 1 \ \tau 2 \ t 1 \ t 2 \ t 3 \ t 4 \ t 5 \ t 6$ **in** $\langle (\text{solves } \text{simp}) ? \rangle$)
using βs -*rev* **apply** *blast*
using *add-param* **by** *blast*

have t_7 : *term-ok* $\Theta \ t$
using βt *proved-terms-well-formed*(2) $t 1 \ t 4$ *term-ok'.simps*(4) *wt-term-def* *term-ok-mk-eq-same-typ* *thy*
by (*meson* *term-ok-app-eqD*)
have t_8 : $\text{typ-of } (\text{Abs-fv } x \ \tau' \ t \ \$ \ \text{term.Fv } x \ \tau') = \text{typ-of } t$
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** $t = \text{Abs } \tau' \ (\text{bind-fv } (x, \tau') \ t) \ \$ \ \text{Fv } x \ \tau'$]
; *use* *assms* *abs-ok* $\tau 1 \ \tau 2 \ t 1 \ t 2 \ t 3 \ t 4 \ t 5 \ t 6 \ t 7 \ t 8$ **in** $\langle (\text{solves } \text{simp}) ? \rangle$)
using *half* **apply** *blast*
using βt **by** *blast*

qed

lemma *obtain-fresh-variable*:

assumes *finite* Γ
obtains x **where** $(x, \tau) \notin \text{fv } t \cup \text{FV } \Gamma$
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** $(x, \tau) \notin \text{fv } t \cup \text{fv } u \cup \text{FV } \Gamma$
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* $\Theta \ f \ \text{typ-of } f = \text{Some } (\tau \rightarrow \tau')$
assumes B : *term-ok* $\Theta \ g \ \text{typ-of } g = \text{Some } (\tau \rightarrow \tau')$
shows $\Theta, \{\} \vdash (\text{Ct } \text{STR } \text{"Pure.all"} ((\tau \rightarrow \text{prop } T) \rightarrow \text{prop } T) \ \$ \ \text{Abs } \tau \ (\text{mk-eq}' \ \tau' \ (f \ \$ \ \text{Bv } 0) \ (g \ \$ \ \text{Bv } 0)))$
 $\mapsto \text{mk-eq } (\text{Abs } \tau \ (f \ \$ \ \text{Bv } 0)) \ (\text{Abs } \tau \ (g \ \$ \ \text{Bv } 0))$

proof–

have *eq-abstract-rule-ax* \in *axioms* Θ
using *thy* **by** (*cases* Θ *rule: theory-full-exhaust*) *auto*
moreover have *ok2: typ-ok* Θ $(\tau \rightarrow \tau')$
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: $\Theta, \{\}$ \vdash subst-typ'*
 $[[(\text{Var } (\text{STR } "'a''", 0), \text{full-sort}), \tau), ((\text{Var } (\text{STR } "'b''", 0), \text{full-sort}), \tau')]$
eq-abstract-rule-ax
using *assms axiom-subst-typ' by (simp del: term-ok-def)*
hence $\Theta, \{\} \vdash$ *subst-term* $[[(\text{Var } (\text{STR } "'g''", 0), \tau \rightarrow \tau'), g),$
 $((\text{Var } (\text{STR } "'f''", 0), \tau \rightarrow \tau'), f)]$ (*subst-typ'*
 $[[(\text{Var } (\text{STR } "'a''", 0), \text{full-sort}), \tau), ((\text{Var } (\text{STR } "'b''", 0), \text{full-sort}), \tau')]$
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* $[[(\text{Var } (\text{STR } "'g''", 0), \tau \rightarrow \tau'), g),$
 $((\text{Var } (\text{STR } "'f''", 0), \tau \rightarrow \tau'), f)]$ (*subst-typ'*
 $[[(\text{Var } (\text{STR } "'a''", 0), \text{full-sort}), \tau), ((\text{Var } (\text{STR } "'b''", 0), \text{full-sort}), \tau')]$
eq-abstract-rule-ax)
 $= (\text{Ct } \text{STR } "'Pure.all'' ((\tau \rightarrow \text{propT}) \rightarrow \text{propT}) \$ \text{Abs } \tau (\text{mk-eq}' \tau' (f \$ \text{Bv } 0)$
 $(g \$ \text{Bv } 0)))$
 $\mapsto \text{mk-eq } (\text{Abs } \tau (f \$ \text{Bv } 0)) (\text{Abs } \tau (g \$ \text{Bv } 0))$
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* $f = \text{Some } (\tau \rightarrow \tau')$
assumes *B: term-ok* Θ *g typ-of* $g = \text{Some } (\tau \rightarrow \tau')$
assumes *ctxt: finite* $\Gamma \forall A \in \Gamma. \text{term-ok } \Theta A \forall A \in \Gamma. \text{typ-of } A = \text{Some propT}$
shows $\Theta, \Gamma \vdash (\text{Ct } \text{STR } "'Pure.all'' ((\tau \rightarrow \text{propT}) \rightarrow \text{propT}) \$ \text{Abs } \tau (\text{mk-eq}' \tau'$
 $(f \$ \text{Bv } 0) (g \$ \text{Bv } 0)))$
 $\mapsto \text{mk-eq } (\text{Abs } \tau (f \$ \text{Bv } 0)) (\text{Abs } \tau (g \$ \text{Bv } 0))$
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* $f = \text{Some } (\tau \rightarrow \tau')$
assumes *B: term-ok* Θ *g typ-of* $g = \text{Some } (\tau \rightarrow \tau')$
assumes $\Theta, \Gamma \vdash (\text{Ct } \text{STR } "'Pure.all'' ((\tau \rightarrow \text{propT}) \rightarrow \text{propT}) \$ \text{Abs } \tau (\text{mk-eq}'$
 $\tau' (f \$ \text{Bv } 0) (g \$ \text{Bv } 0)))$
assumes *ctxt: finite* $\Gamma \forall A \in \Gamma. \text{term-ok } \Theta A \forall A \in \Gamma. \text{typ-of } A = \text{Some propT}$
shows $\Theta, \Gamma \vdash \text{mk-eq } (\text{Abs } \tau (f \$ \text{Bv } 0)) (\text{Abs } \tau (g \$ \text{Bv } 0))$

proof-

note 1 = *proves-eq-abstract-rule*[**where** $\Gamma=\Gamma$, *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* ($\tau \rightarrow \tau'$)

assumes *g*: *term-ok* Θ *g* *typ-of* *g* = *Some* ($\tau \rightarrow \tau'$)

assumes *prem*: $\Theta, \Gamma \vdash \text{Ct STR "Pure.all" } ((\tau \rightarrow \text{propT}) \rightarrow \text{propT}) \text{ \$ Abs } \tau$
(*mk-eq'* τ' (*f* $\text{\$}$ *Bv* 0) (*g* $\text{\$}$ *Bv* 0))

assumes *ctxt*: *finite* $\Gamma \forall A \in \Gamma. \text{term-ok } \Theta A \forall A \in \Gamma. \text{typ-of } A = \text{Some propT}$

shows $\Theta, \Gamma \vdash \text{mk-eq } f g$

proof-

obtain *x* **where** $x: (x, \tau) \notin \text{FV } \Gamma (x, \tau) \notin \text{fv } f (x, \tau) \notin \text{fv } g$

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* $\text{\$}$ *Bv* 0) (*g* $\text{\$}$ *Bv* 0)))

using *prem proved-terms-well-formed*(2) *term-ok-app-eqD* **by** *blast*

have *subst-bv* (*Fv* *x* τ) (*f* $\text{\$}$ *Bv* 0) = *f* $\text{\$}$ *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* $\text{\$}$ *Bv* 0) = *g* $\text{\$}$ *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* $\text{\$}$ *Bv* 0) (*g* $\text{\$}$ *Bv* 0))

= *mk-eq'* τ' (*f* $\text{\$}$ *Fv* *x* τ) (*g* $\text{\$}$ *Fv* *x* τ)

by (*simp add: Core.subst-bv-def*)

hence *simp*: *Abs* τ (*mk-eq'* τ' (*f* $\text{\$}$ *Bv* 0) (*g* $\text{\$}$ *Bv* 0)) \cdot *Fv* *x* τ = *mk-eq* (*f* $\text{\$}$ *Fv* *x* τ) (*g* $\text{\$}$ *Fv* *x* τ)

using *f g* **by** (*auto simp add: typ-of-def*)

hence *simp'*: *subst-bv* (*Fv* *x* τ) (*mk-eq'* τ' (*f* $\text{\$}$ *Bv* 0) (*g* $\text{\$}$ *Bv* 0)) = *mk-eq'* τ' (*f* $\text{\$}$ *Fv* *x* τ) (*g* $\text{\$}$ *Fv* *x* τ)

using *f g* **by** (*auto simp add: typ-of-def*)

have $\Theta, \Gamma \vdash \text{mk-eq}' \tau' (f \text{\$ } Fv \text{ } x \text{ } \tau) (g \text{\$ } Fv \text{ } x \text{ } \tau)$

apply (*subst simp'[symmetric]*)

apply (*rule forall-elim[where* $\tau=\tau'$ *]*)

using *prem* **apply** *blast*

apply *simp*

using $\langle \text{term-ok } \Theta (\text{Abs } \tau (\text{mk-eq}' \tau' (f \text{\$ } Bv \text{ } 0) (g \text{\$ } Bv \text{ } 0))) \rangle \text{term-ok'.simps}(1)$
term-ok'.simps(5) *term-okD1* **by** *blast*

moreover **have** *typ-of* (*f* $\text{\$}$ *Fv* *x* τ) = *Some* τ' *typ-of* (*g* $\text{\$}$ *Fv* *x* τ) = *Some* τ'

using *f*(2) *g*(2) **by** (*simp-all add: typ-of-def*)

ultimately **have** 1: $\Theta, \Gamma \vdash \text{mk-eq} (f \text{\$ } Fv \text{ } x \text{ } \tau) (g \text{\$ } Fv \text{ } x \text{ } \tau)$

by *simp*

have *core*: $\Theta, \Gamma \vdash \text{mk-eq} (\text{Abs } \tau (f \text{\$ } Bv \text{ } 0)) (\text{Abs } \tau (g \text{\$ } Bv \text{ } 0))$

```

apply (rule proves-eq-abstract-rule-rule[OF thy f g - ctxt])
using prem by blast
have  $\Theta, \Gamma \vdash \text{mk-eq} (\text{Abs } \tau (f \$ Bv 0)) f$ 
using f proves.eta term-okD1 thy by blast
have left:  $\Theta, \Gamma \vdash \text{mk-eq} f (\text{Abs } \tau (f \$ Bv 0))$ 
apply (rule proves-eq-symmetric-rule[OF thy f(1) - - - ctxt])
using  $\langle \Theta, \Gamma \vdash \text{mk-eq} (\text{Abs } \tau (f \$ Bv 0)) (\text{Abs } \tau (g \$ Bv 0)) \rangle$  proved-terms-well-formed(2)
term-ok-mk-eqD apply blast
apply (simp add: Logic.typ-of-eta-expand f(2))
using  $\langle \Theta, \Gamma \vdash \text{mk-eq} (\text{Abs } \tau (f \$ Bv 0)) f \rangle$  by blast

have right:  $\Theta, \Gamma \vdash \text{mk-eq} (\text{Abs } \tau (g \$ Bv 0)) g$ 
using g proves.eta term-okD1 thy by blast

```

```

show ?thesis
apply (rule proves-eq-transitive-rule[where  $t = \text{Abs } \tau (f \$ Bv 0)$ , OF thy f(1) -
g(1) - - left - ctxt])
using  $\langle \Theta, \Gamma \vdash \text{mk-eq} (\text{Abs } \tau (f \$ Bv 0)) f \rangle$  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  $t = \text{Abs } \tau (g \$ Bv 0)$ , OF thy - -
g(1) - - core right ctxt])
using  $\langle \Theta, \Gamma \vdash \text{mk-eq} (\text{Abs } \tau (f \$ Bv 0)) f \rangle$  proved-terms-well-formed(2)
term-ok-mk-eqD apply blast
using  $\langle \Theta, \Gamma \vdash \text{mk-eq} (\text{Abs } \tau (g \$ Bv 0)) g \rangle$  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,  $\tau$ ) lev1 (bind-fv2 (x,  $\tau$ ) lev2 t) = bind-fv2 (x,  $\tau$ ) lev2 t
by (induction (x,  $\tau$ ) lev2 t arbitrary: lev1 rule: bind-fv2.induct) auto
corollary bind-fv-idem[simp]:
  bind-fv (x,  $\tau$ ) (bind-fv (x,  $\tau$ ) t) = bind-fv (x,  $\tau$ ) t
using bind-fv-def bind-fv2-idem by simp
corollary bind-fv-Abs-fv[simp]: bind-fv (x,  $\tau$ ) (Abs-fv x  $\tau$  t) = Abs-fv x  $\tau$  t
by (simp add: bind-fv-def)

```

```

lemma bind-fv2 (x,  $\tau$ ) lev (mk-eq'  $\tau'$  s t) = mk-eq'  $\tau'$  (bind-fv2 (x,  $\tau$ ) lev s) (bind-fv2
(x,  $\tau$ ) lev t)
by simp

```

```

lemma bind-fv (x,  $\tau$ ) (mk-eq'  $\tau'$  s t) = mk-eq'  $\tau'$  (bind-fv (x,  $\tau$ ) s) (bind-fv (x,  $\tau$ ) t)
by (simp add: bind-fv-def)

```

```

lemma term-ok-Abs-fvI: term-ok  $\Theta$  s  $\implies$  typ-ok  $\Theta$   $\tau \implies$  term-ok  $\Theta$  (Abs-fv x  $\tau$ 
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*: $(x, \tau) \notin FV \Gamma$ *typ-ok* $\Theta \tau$
assumes *ctxt*: *finite* $\Gamma \forall A \in \Gamma. \text{term-ok } \Theta A \forall A \in \Gamma. \text{typ-of } A = \text{Some propT}$
assumes *eq*: $\Theta, \Gamma \vdash \text{mk-eq } s \ t$
shows $\Theta, \Gamma \vdash \text{mk-eq } (\text{Abs } \tau (\text{bind-fv } (x, \tau) s)) (\text{Abs } \tau (\text{bind-fv } (x, \tau) t))$
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* Θs *term-ok* Θt
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 $\text{Abs } \tau (\text{bind-fv } (x, \tau) (\text{mk-eq } s \ t)) \cdot Fv \ x \ \tau = \text{mk-eq } s \ t$
using *betapply-Abs-fv* **by** *auto*
have $\Theta, \Gamma \vdash \text{mk-all } x \ \tau (\text{mk-eq } s \ t)$
using *eq forall-intro thy typ-ok-def x(1) x(2)* **by** *blast*

have $\Theta, \Gamma \vdash \text{mk-eq } (\text{Abs } \tau (\text{bind-fv } (x, \tau) s) \ \$ \ Fv \ x \ \tau) (\text{subst-bv } (Fv \ x \ \tau) (\text{bind-fv } (x, \tau) s))$
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** $\text{subst-bv } (Fv \ x \ \tau) (\text{bind-fv } (x, \tau) s) = s$
by (*simp add: closed(1) subst-bv-bind-fv*)
ultimately **have** *unfs*: $\Theta, \Gamma \vdash \text{mk-eq } (\text{Abs } \tau (\text{bind-fv } (x, \tau) s) \ \$ \ Fv \ x \ \tau) s$
by *simp*
have $\Theta, \Gamma \vdash \text{mk-eq } (\text{Abs } \tau (\text{bind-fv } (x, \tau) t) \ \$ \ Fv \ x \ \tau) (\text{subst-bv } (Fv \ x \ \tau) (\text{bind-fv } (x, \tau) t))$
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** $\text{subst-bv } (Fv \ x \ \tau) (\text{bind-fv } (x, \tau) t) = t$
by (*simp add: closed(2) subst-bv-bind-fv*)
ultimately **have** *unft*: $\Theta, \Gamma \vdash \text{mk-eq } (\text{Abs } \tau (\text{bind-fv } (x, \tau) t) \ \$ \ Fv \ x \ \tau) t$
by *simp*

have *prem*:
 $\Theta, \Gamma \vdash \text{mk-eq } (\text{Abs } \tau (\text{bind-fv } (x, \tau) s) \ \$ \ Fv \ x \ \tau) (\text{Abs } \tau (\text{bind-fv } (x, \tau) t) \ \$ \ Fv \ x \ \tau)$
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  $\Theta, \Gamma \vdash mk\text{-all } x \tau$ 
  (mk-eq (Abs  $\tau$  (bind-fv (x,  $\tau$ ) s) $ Fv x  $\tau$ ) (Abs  $\tau$  (bind-fv (x,  $\tau$ ) t) $ Fv x  $\tau$ ))
  using forall-intro thy typ-ok-def x(1) x(2) by blast
moreover have mk-all x  $\tau$ 
  (mk-eq (Abs  $\tau$  (bind-fv (x,  $\tau$ ) s) $ Fv x  $\tau$ ) (Abs  $\tau$  (bind-fv (x,  $\tau$ ) t) $ Fv x  $\tau$ ))
  = mk-all x  $\tau$ 
  (mk-eq'  $\tau'$  (Abs  $\tau$  (bind-fv (x,  $\tau$ ) s) $ Fv x  $\tau$ ) (Abs  $\tau$  (bind-fv (x,  $\tau$ ) t) $ Fv x
 $\tau$ ))
  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  $\tau$ 
  (mk-eq'  $\tau'$  (Abs  $\tau$  (bind-fv (x,  $\tau$ ) s) $ Fv x  $\tau$ ) (Abs  $\tau$  (bind-fv (x,  $\tau$ ) t) $ Fv x
 $\tau$ )) =
  Ct STR "Pure.all" (( $\tau \rightarrow propT$ )  $\rightarrow propT$ ) $ Abs  $\tau$ 
  (mk-eq'  $\tau'$  (Abs  $\tau$  (bind-fv (x,  $\tau$ ) s) $ Bv 0) (Abs  $\tau$  (bind-fv (x,  $\tau$ ) t) $ Bv 0))
  by (simp add: bind-fv-def)
ultimately have pre-ext:  $\Theta, \Gamma \vdash Ct\ STR\ "Pure.all"\ ((\tau \rightarrow propT) \rightarrow propT)$ 
$ Abs  $\tau$ 
  (mk-eq'  $\tau'$  (Abs  $\tau$  (bind-fv (x,  $\tau$ ) s) $ Bv 0) (Abs  $\tau$  (bind-fv (x,  $\tau$ ) t) $ Bv 0))
  by simp
show ?thesis
  apply (rule proves-eq-ext-rule[where  $\tau=\tau$  and  $\tau'=\tau'$ , 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-ok $\Theta \tau'$
assumes *ctxt*: finite $\Gamma \forall A \in \Gamma. \text{term-ok } \Theta A \forall A \in \Gamma. \text{typ-of } A = \text{Some propT}$
shows $\Theta, \Gamma \vdash \text{mk-eq } s \ t$
 $\longleftrightarrow \Theta, \Gamma \vdash \text{mk-eq } (\text{Abs } \tau' (\text{bind-fv } (x, \tau') \ s)) \ (\text{Abs } \tau' (\text{bind-fv } (x, \tau') \ t))$
proof (rule iff1)
assume *asm*: $\Theta, \Gamma \vdash \text{mk-eq } s \ t$
hence *term-ok* $\Theta \ s$ *term-ok* $\Theta \ t$
using proved-terms-well-formed(2) *term-ok-mk-eqD* **by** blast+
show $\Theta, \Gamma \vdash \text{mk-eq } (\text{Abs-fv } x \ \tau' \ s) \ (\text{Abs-fv } x \ \tau' \ t)$
by (rule proves-eq-abstract-rule-derived-rule[OF *thy x ctxt asm*])
next
assume *asm*: $\Theta, \Gamma \vdash \text{mk-eq } (\text{Abs-fv } x \ \tau' \ s) \ (\text{Abs-fv } x \ \tau' \ t)$
show $\Theta, \Gamma \vdash \text{mk-eq } s \ t$
using *assms asm proves-descend-abs-rule* **by** blast
qed

lemma *proves-descend-abs-rule'*:

assumes *thy*: wf-theory Θ
assumes *eq*: $\Theta, \Gamma \vdash \text{mk-eq } (\text{Abs } \tau' \ s) \ (\text{Abs } \tau' \ t)$
assumes *x*: $(x, \tau') \notin FV \Gamma$ typ-ok $\Theta \tau'$
assumes *ctxt*: finite $\Gamma \forall A \in \Gamma. \text{term-ok } \Theta A \forall A \in \Gamma. \text{typ-of } A = \text{Some propT}$
shows $\Theta, \Gamma \vdash \text{mk-eq } (\text{subst-bv } (\text{Fv } x \ \tau') \ s) \ (\text{subst-bv } (\text{Fv } x \ \tau') \ t)$
proof–
have *abs-ok*: *term-ok* $\Theta \ (\text{Abs } \tau' \ s)$ *term-ok* $\Theta \ (\text{Abs } \tau' \ t)$
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* $(\text{Abs } \tau' \ s) = \text{Some } (\tau' \rightarrow \tau)$
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* $(\text{Abs } \tau' \ t) = \text{Some } (\tau' \rightarrow \tau)$
by (metis *eq(1) proved-terms-well-formed(2) term-ok-mk-eq-same-typ thy*)

have *add-param*: $\Theta, \Gamma \vdash \text{mk-eq}$
 $(\text{Abs } \tau' \ s \ \$ \ \text{Fv } x \ \tau')$
 $(\text{Abs } \tau' \ t \ \$ \ \text{Fv } x \ \tau')$
apply (rule *proves-eq-combination-rule*; use *assms abs-ok $\tau 1 \ \tau 2$ in* $\langle(\text{solves}$
 $\langle \text{simp del: term-ok-def} \rangle) \rangle$)
using *proves-eq-reflexive term-ok-var thy x(2) ctxt* **by** blast

have βs : $\Theta, \Gamma \vdash \text{mk-eq}$
 $(\text{Abs } \tau' \ s \ \$ \ \text{Fv } x \ \tau')$
 $(\text{subst-bv } (\text{Fv } x \ \tau') \ s)$
by (rule *proves. β -conversion*; use *assms abs-ok $\tau 1 \ \tau 2$ in* $\langle(\text{solves } \langle \text{simp add:}$
 $\text{wt-term-def} \rangle) \rangle$)

```

have t1: term-ok  $\Theta$  (subst-bv (Fv x  $\tau'$ ) s)
  using  $\beta$ s proved-terms-well-formed(2) wt-term-def typ-of-def
  using term-ok-mk-eqD by blast
have t2: term-ok  $\Theta$  (Abs  $\tau'$  s $ term.Fv x  $\tau'$ )
  using  $\beta$ 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  $\beta$ s-rev:  $\Theta, \Gamma \vdash$  mk-eq (subst-bv (Fv x  $\tau'$ ) s) (Abs  $\tau'$  s $ Fv x  $\tau'$ )
  apply (rule proves-eq-symmetric-rule; use assms abs-ok  $\tau$ 1  $\tau$ 2 t1 t2 in  $\langle$ (solves simp) $\rangle$ ?)
  using  $\beta$ s proved-terms-well-formed(2) term-ok-mk-eq-same-typ thy apply blast
  using  $\beta$ s by simp

have  $\beta$ t:  $\Theta, \Gamma \vdash$  mk-eq
  (Abs  $\tau'$  t $ Fv x  $\tau'$ )
  (subst-bv (Fv x  $\tau'$ ) t)
  by (rule proves. $\beta$ -conversion; use assms abs-ok  $\tau$ 1  $\tau$ 2 t1 in  $\langle$ (solves  $\langle$ simp add: wt-term-def $\rangle$ ) $\rangle$ ?)

have t3: term-ok  $\Theta$  (Abs  $\tau'$  t $ term.Fv x  $\tau'$ )
  using  $\beta$ 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  $\tau'$ ) s) = typ-of (Abs  $\tau'$  t $ term.Fv x  $\tau'$ )
  by (metis  $\beta$ s add-param proved-terms-well-formed(2) term-ok-mk-eq-same-typ
thy)
have t5: typ-of (subst-bv (Fv x  $\tau'$ ) s) = typ-of (Abs  $\tau'$  s $ Fv x  $\tau'$ )
  using  $\beta$ s-rev proved-terms-well-formed(2) term-ok-mk-eq-same-typ thy by blast
have t6: typ-of (Abs  $\tau'$  s $ Fv x  $\tau'$ ) = typ-of (Abs  $\tau'$  t $ term.Fv x  $\tau'$ )
  using t4 t5 by auto

have half:  $\Theta, \Gamma \vdash$  mk-eq (subst-bv (Fv x  $\tau'$ ) s) (Abs  $\tau'$  t $ Fv x  $\tau'$ )
  apply (rule proves-eq-transitive-rule[where t=Abs  $\tau'$  s $ Fv x  $\tau'$ ]
; use assms abs-ok  $\tau$ 1  $\tau$ 2 t1 t2 t3 t4 t5 t6 in  $\langle$ (solves simp) $\rangle$ ?)
  using  $\beta$ s-rev apply blast
  using add-param by blast

have t7: term-ok  $\Theta$  (subst-bv (Fv x  $\tau'$ ) t)
  using  $\beta$ 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  $\tau'$  t $ term.Fv x  $\tau'$ ) = typ-of (subst-bv (Fv x  $\tau'$ ) t)
  using  $\beta$ 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  $\tau'$  t $ Fv x  $\tau'$ ]
; use assms abs-ok  $\tau$ 1  $\tau$ 2 t1 t2 t3 t4 t5 t6 t7 t8 in  $\langle$ (solves simp) $\rangle$ ?)
  using half apply blast

```

using βt by blast
qed

lemma proves-ascend-abs-rule':

assumes thy: wf-theory Θ

assumes x: $(x, \tau') \notin FV \Gamma$ $(x, \tau') \notin fv (mk\text{-eq} (Abs \tau' s) (Abs \tau' t))$ $typ\text{-ok} \Theta \tau'$

assumes eq: $\Theta, \Gamma \vdash mk\text{-eq} (subst\text{-bv} (Fv x \tau') s) (subst\text{-bv} (Fv x \tau') t)$

assumes ctxt: finite $\Gamma \forall A \in \Gamma. term\text{-ok} \Theta A \forall A \in \Gamma. typ\text{-of} A = Some \text{prop} T$

shows $\Theta, \Gamma \vdash mk\text{-eq} (Abs \tau' s) (Abs \tau' t)$

proof-

have ok-ind: wf-type (sig Θ) τ'

using $x(\beta)$ by simp

note 1 = proves-eq-abstract-rule-derived-rule[OF thy]

have term-ok $\Theta (subst\text{-bv} (Fv x \tau') s)$

using eq proved-terms-well-formed(2) wt-term-def typ-of-def

by (meson term-ok-app-eqD)

hence is-closed $(subst\text{-bv} (Fv x \tau') s)$

using wt-term-def typ-of-imp-closed by auto

hence loose-s: \neg loose-bvar s 1

using is-closed-subst-bv by simp

hence loose-s': $(\bigwedge x. 1 < x \implies \neg$ loose-bvar1 s x)

by (simp add: not-loose-bvar-imp-not-loose-bvar1-all-greater)

moreover have \neg occs (case-prod $Fv (x, \tau')$) s

proof-

have $(x, \tau') \notin fv s$

using $x(2)$ by auto

thus ?thesis

by (simp add: fv-iff-occs)

qed

ultimately have s: $Abs\text{-fv} x \tau' (subst\text{-bv} (term.Fv x \tau') s) = Abs \tau' s$

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 $\Theta (subst\text{-bv} (Fv x \tau') t)$

using eq proved-terms-well-formed(2) wt-term-def typ-of-def

by (meson term-ok-app-eqD)

hence is-closed $(subst\text{-bv} (Fv x \tau') t)$

using wt-term-def typ-of-imp-closed by auto

hence loose-s: \neg loose-bvar t 1

using is-closed-subst-bv by simp

hence loose-s': $(\bigwedge x. 1 < x \implies \neg$ loose-bvar1 t x)

by (simp add: not-loose-bvar-imp-not-loose-bvar1-all-greater)

moreover have \neg occs (case-prod $Fv (x, \tau')$) t

proof-

have $(x, \tau') \notin fv t$

```

    using x(2) by auto
  thus ?thesis
    by (simp add: fv-iff-occs)
qed
ultimately have t: Abs-fv x  $\tau'$  (subst-bv (term.Fv x  $\tau'$ ) t) = Abs  $\tau'$  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  $\Theta$ 
  assumes x:  $(x, \tau') \notin FV \Gamma$   $(x, \tau') \notin fv$  (mk-eq (Abs  $\tau'$  s) (Abs  $\tau'$  t)) typ-ok  $\Theta$   $\tau'$ 
  assumes ctxt: finite  $\Gamma$   $\forall A \in \Gamma. term-ok \Theta A \forall A \in \Gamma. typ-of A = Some propT$ 
  shows  $\Theta, \Gamma \vdash mk-eq$  (subst-bv (Fv x  $\tau'$ ) s) (subst-bv (Fv x  $\tau'$ ) t)
     $\longleftrightarrow \Theta, \Gamma \vdash mk-eq$  (Abs  $\tau'$  s) (Abs  $\tau'$  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  $\Theta$ 
  assumes finite: finite  $\Gamma$ 
  assumes free:  $\forall (x, \tau) \in set\ vs. (x, \tau) \notin fv\ t \cup FV \Gamma$ 
  assumes term-ok': term-ok  $\Theta$  (subst-bvs (map (case-prod Fv) vs) t)
  assumes beta:  $t \rightarrow_{\beta} u$ 
  assumes ctxt:  $\forall A \in \Gamma. term-ok \Theta A \forall A \in \Gamma. typ-of A = Some propT$ 
  shows  $\Theta, \Gamma \vdash mk-eq$ 
    (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  $\Theta$  (subst-bvs (map (case-prod Fv) vs) (Abs T s))
    term-ok  $\Theta$  (subst-bvs (map (case-prod Fv) vs) t)
  using beta.prem(1) apply simp-all
  using term-ok-app-eqD term-ok-def by blast+

  have  $\forall x \in set$  (map (case-prod Fv) vs) . is-closed x
  using beta.prem(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  $\Theta$  (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.premis(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 (λa. 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.premis(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)

```

$term-ok \Theta (subst-bvs (map (case-prod Fv) vs) u)$
by $(metis\ subst-bvs-App\ term-ok-app-eqD)+$
moreover have $\forall a \in set\ vs.\ case\ a\ of\ (x, \tau) \Rightarrow (x, \tau) \notin fv\ s \cup FV\ \Gamma$
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 $proves-eq-reflexive[OF\ thy\ ok(2),\ of\ \Gamma,\ OF\ finite\ ctxt]$ **by** $blast$
moreover obtain τ **where** $\tau: typ-of$
 $(subst-bvs\ (map\ (case-prod\ Fv)\ vs)\ s) = Some\ \tau$
using $ok\ wt-term-def$ **by** $auto$
moreover obtain τ' **where** $typ-of$
 $(subst-bvs\ (map\ (case-prod\ Fv)\ vs)\ u) = Some\ (\tau \rightarrow \tau')$
using $\tau\ appR.prem(1)\ not-None-eq\ subst-bvs-App\ wt-term-def\ typ-of1-arg-typ$
 $typ-of-def$
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 $\forall a \in set\ vs.\ case\ a\ of\ (x, \tau) \Rightarrow (x, \tau) \notin fv\ s \cup FV\ \Gamma$
using $abs.prem(2)$ **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 $T-ok: typ-ok\ \Theta\ T$
using $abs.prem\ term-ok-Types-typ-ok\ simp\ thy$ **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 $\bigwedge v\ t\ P . (v, t) \notin P \vee v \in fst\ 'P$
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** $x: (x, T) \notin (fv (mk\text{-}eq (Abs\ T (subst\text{-}bvs1'\ s\ 1 (map (case\text{-}prod\ Fv)\ vs)))) \cup FV\ \Gamma \cup fv\ s)$
(Abs T (subst-bvs1' t 1 (map (case-prod Fv) vs)))
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-bvs1' t 1 (map (case-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.premis(1) simp by auto*

thm *subst-bvs-extend-lower-level*

have *combine: (subst-bv (term.Fv x T) (subst-bvs1' s 1 (map ($\lambda(x, y). term.Fv\ x\ y$) vs))) = (subst-bvs (map (case-prod Fv) ((x, T)#vs)) s)*
using *subst-bvs-extend-lower-level*
using $\langle \forall v \in set (map (\lambda(x, y). term.Fv\ x\ y)\ vs). is\text{-}closed\ v \rangle$ **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.premis(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 $T\text{-}ok$ **apply** *simp*
using $1 \langle \forall v \in set (map (\lambda(x, y). term.Fv\ x\ y)\ vs). is\text{-}closed\ v \rangle$ *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* $\Theta\ t$

assumes *beta: t* \rightarrow_{β} u

assumes *ctxt: $\forall A \in \Gamma. term\text{-}ok\ \Theta\ A\ \forall A \in \Gamma. typ\text{-}of\ A = Some\ prop\ T$*

```

shows  $\Theta, \Gamma \vdash mk\text{-eq } t \ u$ 
proof-
  have unsimt:  $t = subst\text{-bvs } (map (case\text{-prod } Fv) []) t$ 
    by simp
  moreover have unsimpu:  $u = subst\text{-bvs } (map (case\text{-prod } Fv) []) u$ 
    by simp
  ultimately have unsimp:  $mk\text{-eq } t \ u = mk\text{-eq}$ 
     $(subst\text{-bvs } (map (case\text{-prod } Fv) []) t)$ 
     $(subst\text{-bvs } (map (case\text{-prod } Fv) []) u)$ 
    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  $\Theta$ 
  assumes finite: finite  $\Gamma$ 
  assumes term-ok: term-ok  $\Theta \ t$ 
  assumes beta:  $t \rightarrow_{\beta}^* u$ 
  assumes ctxt:  $\forall A \in \Gamma. \text{term-ok } \Theta \ A \ \forall A \in \Gamma. \text{typ-of } A = \text{Some prop } T$ 
  shows  $\Theta, \Gamma \vdash mk\text{-eq } t \ u$ 
using beta term-ok proof (induction rule: rtrancl.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  $\Theta, \Gamma \vdash mk\text{-eq } a \ b$  by simp
      moreover have  $\Theta, \Gamma \vdash mk\text{-eq } b \ c$ 
        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.prem 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  $\Theta$ 
  assumes finite: finite  $\Gamma$ 
  assumes term-ok: term-ok  $\Theta \ t$ 
  assumes beta: beta-norm  $t = \text{Some } u$ 
  assumes ctxt:  $\forall A \in \Gamma. \text{term-ok } \Theta \ A \ \forall A \in \Gamma. \text{typ-of } A = \text{Some prop } T$ 
  shows  $\Theta, \Gamma \vdash mk\text{-eq } t \ u$ 
  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*: $\Theta, \Gamma \vdash t$
assumes *beta: beta-norm* $t = \text{Some } u$
assumes *ctxt*: $\forall A \in \Gamma. \text{term-ok } \Theta \ A \ \forall A \in \Gamma. \text{typ-of } A = \text{Some prop } T$
shows $\Theta, \Gamma \vdash u$
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*: $\forall (x, \tau) \in \text{set } vs . (x, \tau) \notin \text{fv } t \cup \text{FV } \Gamma$
assumes *term-ok'*: *term-ok* Θ (*subst-bvs* (*map* (*case-prod Fv*) *vs*) *t*)
assumes *eta*: $t \rightarrow_{\eta} u$
assumes *ctxt*: $\forall A \in \Gamma. \text{term-ok } \Theta \ A \ \forall A \in \Gamma. \text{typ-of } A = \text{Some prop } T$
shows $\Theta, \Gamma \vdash \text{mk-eq}$
(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 *closed*: $\forall x \in \text{set } (\text{map } (\text{case-prod } Fv) \text{ vs}) . \text{is-closed } x$
using *eta.prem*(2) **by** *auto*
hence *simp*: *subst-bvs* (*map* (*case-prod Fv*) *vs*) (*Abs T* (*s* $\$$ *Bv 0*))
 $= \text{Abs } T$ (*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*))
 $= \text{Abs } T$ (*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*: $\neg \text{loose-bvar } (\text{subst-bvs1}' \ s \ 1 \ (\text{map } (\text{case-prod } Fv) \ \text{vs})) \ 1$
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*: $\neg \text{is-dependent } (\text{subst-bvs1}' \ s \ 1 \ (\text{map } (\text{case-prod } Fv) \ \text{vs}))$
using *is-closed-subst-bvs1'-closed*
by (*simp add: closed*s *eta.hyps*)

have *decr-simp*: *subst-bv* *x* (*subst-bvs1'* *s* 1 (*map* (*case-prod Fv*) *vs*))
 $= \text{subst-bvs}$ (*map* (*case-prod Fv*) *vs*) (*decr 0 s*) **for** *x*
apply (*simp add: closed*s *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  $\Theta$  (subst-bvs1' s 1 (map (case-prod Fv) vs))
  by (metis One-nat-def Suc-leI eta.premis(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  $\Theta$ ) (subst-bvs1' s 1 (map (case-prod Fv) vs))
using wt-term-def by simp

obtain  $\tau$  where typ-of (Abs T (subst-bvs1' (s  $\$$  Bv 0) 1 (map (case-prod Fv)
vs))) = Some ( $T \rightarrow \tau$ )
using eta.premis(1) simp wt-term-def typ-of-Abs-body-tyt'
by (smt has-tyt-iff-tyt-of tyt-of-def term-ok-def)
hence ty: typ-of (subst-bvs1' s 1 (map (case-prod Fv) vs)) = Some ( $T \rightarrow \tau$ )
using eta.eta eta-preserves-tyt-of is-closed-decr-unchanged not-dependent
  ok simp simp' wt-term-def tyt-of-imp-closed
by (metis (no-types, lifting) has-tyt-imp-closed term-ok-def)

then show ?case
using proves.eta[OF thy ok-ind, of - -  $\Gamma$ ] ty decr-simp simp'
by (simp add: closed_s eta.hyps subst-bvs-decr tyt-of-imp-closed)
next
case (appL s t u)
hence ok: term-ok  $\Theta$  (subst-bvs (map (case-prod Fv) vs) s)
  term-ok  $\Theta$  (subst-bvs (map (case-prod Fv) vs) u)
  by (metis subst-bvs-App term-ok-app-eqD)
moreover have  $\forall a \in \text{set } vs. \text{case } a \text{ of } (x, \tau) \Rightarrow (x, \tau) \notin \text{fv } s \cup \text{FV } \Gamma$ 
using appL by simp
ultimately have  $\Theta, \Gamma \vdash \text{mk-eq}$  (subst-bvs (map (case-prod Fv) vs) s)
  (subst-bvs (map (case-prod Fv) vs) t)
using appL.IH by blast
moreover have  $\Theta, \Gamma \vdash \text{mk-eq}$  (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  $\Gamma$ , OF finite_ctxt] by blast
moreover obtain  $\tau$  where tyt-of
  (subst-bvs (map (case-prod Fv) vs) u) = Some  $\tau$ 
using ok wt-term-def by auto
moreover obtain  $\tau'$  where tyt-of
  (subst-bvs (map (case-prod Fv) vs) s) = Some ( $\tau \rightarrow \tau'$ )
using  $\tau$  appL.premis(1) not-None-eq subst-bvs-App wt-term-def tyt-of1-arg-tyt
tyt-of-def
by (smt has-tyt-iff-tyt-of tyt-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  $\Theta$  (subst-bvs (map (case-prod Fv) vs) s)
  term-ok  $\Theta$  (subst-bvs (map (case-prod Fv) vs) u)

```

by (metis subst-bvs-App term-ok-app-eqD)+
moreover have $\forall a \in \text{set } vs. \text{ case } a \text{ of } (x, \tau) \Rightarrow (x, \tau) \notin \text{fv } s \cup \text{FV } \Gamma$
using appR **by** simp
ultimately have $\Theta, \Gamma \vdash \text{mk-eq } (\text{subst-bvs } (\text{map } (\text{case-prod } Fv) \text{ vs}) s)$
 $(\text{subst-bvs } (\text{map } (\text{case-prod } Fv) \text{ vs}) t)$
using appR.IH **by** blast
moreover have $\Theta, \Gamma \vdash \text{mk-eq } (\text{subst-bvs } (\text{map } (\text{case-prod } Fv) \text{ vs}) u)$
 $(\text{subst-bvs } (\text{map } (\text{case-prod } Fv) \text{ vs}) u)$
using proves-eq-reflexive[OF thy ok(2), of Γ , OF finite ctxt] **by** blast
moreover obtain τ **where** τ : typ-of
 $(\text{subst-bvs } (\text{map } (\text{case-prod } Fv) \text{ vs}) s) = \text{Some } \tau$
using ok wt-term-def **by** auto
moreover obtain τ' **where** typ-of
 $(\text{subst-bvs } (\text{map } (\text{case-prod } Fv) \text{ vs}) u) = \text{Some } (\tau \rightarrow \tau')$
using τ appR.prem(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 $\forall a \in \text{set } vs. \text{ case } a \text{ of } (x, \tau) \Rightarrow (x, \tau) \notin \text{fv } s \cup \text{FV } \Gamma$
using abs.prem(2) **by** auto

have $\forall v \in \text{set } (\text{map } (\text{case-prod } Fv) \text{ vs}) . \text{is-closed } v$
by auto

hence simp: mk-eq (subst-bvs (map (case-prod Fv) vs) (Abs T s))
 $(\text{subst-bvs } (\text{map } (\text{case-prod } Fv) \text{ vs}) (\text{Abs } T t))$
 $= \text{mk-eq } (\text{Abs } T (\text{subst-bvs1}' s 1 (\text{map } (\text{case-prod } Fv) \text{ vs})))$
 $(\text{Abs } T (\text{subst-bvs1}' t 1 (\text{map } (\text{case-prod } Fv) \text{ vs})))$
by simp

have T-ok: typ-ok Θ T
using abs.prem 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)))
 $(\text{Abs } T (\text{subst-bvs1}' t 1 (\text{map } (\text{case-prod } Fv) \text{ vs})))) \cup \text{FV } \Gamma \cup \text{fv } s)$
using finite finite-fv finite-FV **by** simp
hence $\exists x . (x, T) \notin (\text{fv } (\text{mk-eq } (\text{Abs } T (\text{subst-bvs1}' s 1 (\text{map } (\text{case-prod } Fv) \text{ vs})))$
 $(\text{Abs } T (\text{subst-bvs1}' t 1 (\text{map } (\text{case-prod } Fv) \text{ vs})))) \cup \text{FV } \Gamma \cup \text{fv } s)$
proof –
have $\bigwedge v t P. (v::\text{variable}, t::\text{typ}) \notin P \vee v \in \text{fst } ' P$
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.premis(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 $\langle \forall v \in set (map (\lambda(x, y). term.Fv\ x\ y)\ vs). is\text{-}closed\ v \rangle$ **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 $x\ abs.premis(2)$ **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 $T\text{-}ok$ **apply** *simp*
using $1 \langle \forall v \in set (map (\lambda(x, y). term.Fv\ x\ y)\ vs). is\text{-}closed\ v \rangle$ *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-ok* $\Theta\ t$
assumes *eta: t* $\rightarrow_{\eta}\ u$
assumes *ctxt: $\forall A \in \Gamma. term\text{-}ok\ \Theta\ A\ \forall A \in \Gamma. typ\text{-}of\ A = Some\ prop\ T$*
shows $\Theta, \Gamma \vdash mk\text{-}eq\ t\ u$

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-eta-step-pre)
    using assms by simp-all
qed

lemma proves-eta-steps:
  assumes thy: wf-theory  $\Theta$ 
  assumes finite: finite  $\Gamma$ 
  assumes term-ok: term-ok  $\Theta$  t
  assumes eta:  $t \rightarrow_{\eta}^* u$ 
  assumes ctxt:  $\forall A \in \Gamma. \text{term-ok } \Theta \ A \ \forall A \in \Gamma. \text{typ-of } A = \text{Some propT}$ 
  shows  $\Theta, \Gamma \vdash \text{mk-eq } t \ u$ 
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  $\Theta, \Gamma \vdash \text{mk-eq } a \ b$  by simp
  moreover have  $\Theta, \Gamma \vdash \text{mk-eq } b \ c$ 
    using proves-eta-step rtrancl-into-rtrancl.hyps(2) eta-star-preserves-term-ok
  local.finite
    rtrancl-into-rtrancl.hyps(1) rtrancl-into-rtrancl.prem1 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-tyt thy)
qed

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

```

```

assumes ctxt:  $\forall A \in \Gamma. \text{term-ok } \Theta \ A \ \forall A \in \Gamma. \text{typ-of } A = \text{Some propT}$ 
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  $\Theta$ 
assumes finite: finite  $\Gamma$ 
assumes term-ok:  $\Theta, \Gamma \vdash t$ 
assumes beta-eta: beta-eta-norm  $t = \text{Some } u$ 
assumes ctxt:  $\forall A \in \Gamma. \text{term-ok } \Theta \ A \ \forall A \in \Gamma. \text{typ-of } A = \text{Some propT}$ 
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  $\Theta$ 
assumes all:  $\Theta, \Gamma \vdash \text{Ct STR "Pure.all" } ((\tau \rightarrow \text{propT}) \rightarrow \text{propT}) \ \$ \ B$ 
assumes a: has-typ  $a \ \tau \ \text{wf-term } (\text{sig } \Theta) \ a$ 
assumes ctxt:  $\text{finite } \Gamma \ \forall A \in \Gamma. \text{term-ok } \Theta \ A \ \forall A \in \Gamma. \text{typ-of } A = \text{Some propT}$ 
shows  $\Theta, \Gamma \vdash B \cdot a$ 
proof(cases is-Abs B)
  case True
    from this obtain  $t \ T$  where Abs:  $B = \text{Abs } T \ t$ 
      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  $\tau = \tau$ ])
  next
    case False

    have wf-B: wf-term (sig  $\Theta$ )  $B$ 
      using all proved-terms-well-formed(2) term-okD1 term-ok-app-eqD by blast
    have B-typ:  $\vdash_{\tau} B : \tau \rightarrow \text{propT}$ 
      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  $\tau (B \ \$ \ Bv \ 0) \cdot a = B \ \$ \ a$ 
      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:  $B \cdot a = \text{subst-bv } a \ (B \ \$ \ Bv \ 0)$ 
      by auto

```

```

have 1:  $\Theta, \Gamma \vdash \text{mk-eq} (\text{Abs } \tau (B \$ Bv 0)) B$ 
  by (rule proves.eta[OF thy wf-B B-typ])
have 2:  $\Theta, \Gamma \vdash \text{mk-eq} B (\text{Abs } \tau (B \$ Bv 0))$ 
  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:  $\Theta, \Gamma \vdash \text{mk-eq} (\text{Ct STR "Pure.all" } ((\tau \rightarrow \text{propT}) \rightarrow \text{propT})) (\text{Ct STR$ 
"Pure.all" } ((\tau \rightarrow \text{propT}) \rightarrow \text{propT}))
```

apply (rule proves-eq-reflexive[OF thy - ctxt])
using all proved-terms-well-formed(2) term-ok-app-eqD **by** blast

```

have 4:  $\Theta, \Gamma \vdash \text{mk-eq}$ 
  ( $\text{Ct STR "Pure.all" } ((\tau \rightarrow \text{propT}) \rightarrow \text{propT}) \$ B$ )
  ( $\text{Ct STR "Pure.all" } ((\tau \rightarrow \text{propT}) \rightarrow \text{propT}) \$ (\text{Abs } \tau (B \$ Bv 0))$ )
  apply (rule proves-eq-combination-rule-better[OF thy 3 2 - - ctxt, where  $\tau = (\tau$ 
 $\rightarrow \text{propT})$  and  $\tau' = \text{propT}$ ])
  using typ-of-def apply auto[1]
  using B-typ by blast

have 5:  $\Theta, \Gamma \vdash (\text{Ct STR "Pure.all" } ((\tau \rightarrow \text{propT}) \rightarrow \text{propT}) \$ (\text{Abs } \tau (B \$ Bv$ 
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  $\times$  sort)  $\times$  typ
type-synonym tinst = (variable  $\times$  typ)  $\times$  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  $\Rightarrow$  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  $\Rightarrow$  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 \geq depth P

by (induction P) auto

fun partial-nth :: 'a list \Rightarrow nat \Rightarrow '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 \equiv if n < length xs then Some (nth xs n) else None

lemma partial-nth xs n \equiv partial-nth' xs n

by (induction rule: partial-nth.induct) auto

lemma partial-nth-Some-imp-elem: partial-nth l n = Some x \implies x \in set l

by (induction rule: partial-nth.induct) auto

The core of the proof checker

```
fun replay' :: theory  $\Rightarrow$  (variable  $\times$  typ) list  $\Rightarrow$  variable set
 $\Rightarrow$  term list  $\Rightarrow$  proofterm  $\Rightarrow$  term option where
  replay' thy - - Hs (PAxm t Tis) = (if inst-ok thy Tis  $\wedge$  term-ok thy t
    then if t  $\in$  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)
```

$| \text{replay}' \text{ thy vs ns Hs (Appt p t) =}$
 $(\text{let rep} = \text{replay}' \text{ thy vs ns Hs p in}$
 $\text{let t}' = \text{subst-bvs} (\text{map} (\lambda(x,y) . \text{Fv } x \ y) \ \text{vs}) \ t \ \text{in}$
 $\text{case} (\text{rep}, \text{typ-of } t')$ of
 $(\text{Some } (\text{Ct } s \ (\text{Ty } \text{fun1} \ [\text{Ty } \text{fun2} \ [\tau, \text{Ty } \text{propT1 } \ \text{Nil}], \text{Ty } \text{propT2 } \ \text{Nil}]) \ \$ \ b),$
 $\text{Some } \tau')$ \Rightarrow
 $\text{if } s = \text{STR } \text{"Pure.all"} \wedge \text{fun1} = \text{STR } \text{"fun"} \wedge \text{fun2} = \text{STR } \text{"fun"}$
 $\wedge \text{propT1} = \text{STR } \text{"prop"} \wedge \text{propT2} = \text{STR } \text{"prop"}$
 $\wedge \tau = \tau' \wedge \text{term-ok thy } t'$
 $\text{then } \text{Some } (b \cdot t') \ \text{else } \text{None}$
 $| - \Rightarrow \text{None})$
 $| \text{replay}' \text{ thy vs ns Hs (AbsP t p) =}$
 $(\text{let t}' = \text{subst-bvs} (\text{map} (\lambda(x,y) . \text{Fv } x \ y) \ \text{vs}) \ t \ \text{in}$
 $\text{let rep} = \text{replay}' \text{ thy vs ns } (t' \# \text{Hs}) \ p \ \text{in}$
 $(\text{if } \text{typ-of } t' = \text{Some } \text{propT} \wedge \text{term-ok thy } t' \ \text{then } \text{map-option } (\text{mk-imp } t') \ \text{rep}$
 $\text{else } \text{None}))$
 $| \text{replay}' \text{ thy vs ns Hs (AppP p1 p2) =}$
 $(\text{let rep1} = \text{Option.bind } (\text{replay}' \text{ thy vs ns Hs } p1) \ \text{beta-eta-norm in}$
 $\text{let rep2} = \text{Option.bind } (\text{replay}' \text{ thy vs ns Hs } p2) \ \text{beta-eta-norm in}$
 $(\text{case } (\text{rep1}, \text{rep2}) \ \text{of } ($
 $\text{Some } (\text{Ct } \text{imp} \ (\text{Ty } \text{fn1} \ [\text{Ty } \text{prp1} \ []], \text{Ty } \text{fn2} \ [\text{Ty } \text{prp2} \ []], \text{Ty } \text{prp3} \ [])) \ \$ \ A \ \$ \ B),$
 $\text{Some } A')$ \Rightarrow
 $\text{if } \text{imp} = \text{STR } \text{"Pure.imp"} \wedge \text{fn1} = \text{STR } \text{"fun"} \wedge \text{fn2} = \text{STR } \text{"fun"}$
 $\wedge \text{prp1} = \text{STR } \text{"prop"} \wedge \text{prp2} = \text{STR } \text{"prop"} \wedge \text{prp3} = \text{STR } \text{"prop"} \wedge$
 $A = A'$
 $\text{then } \text{Some } B \ \text{else } \text{None}$
 $| - \Rightarrow \text{None}))$
 $| \text{replay}' \text{ thy vs ns Hs (OfClass ty c) = (if has-sort (osig (sig thy)) ty \{c\}$
 $\wedge \text{typ-ok thy ty}$
 $\text{then } (\text{case } \text{const-type } (\text{sig thy}) \ (\text{const-of-class } c) \ \text{of}$
 $\text{Some } (\text{Ty } \text{fun} \ [\text{Ty } \text{it} \ [\text{ity}], \text{Ty } \text{prop} \ []]) \ \Rightarrow$
 $\text{if } \text{ity} = \text{variable } \text{STR } \text{"a"} \wedge \text{fun} = \text{STR } \text{"fun"} \wedge \text{prop} = \text{STR } \text{"prop"} \wedge \text{it}$
 $= \text{STR } \text{"itself"}$
 $\text{then } \text{Some } (\text{mk-of-class ty } c) \ \text{else } \text{None} \ | - \Rightarrow \text{None}) \ \text{else } \text{None})$
 $| \text{replay}' \text{ thy vs ns Hs (Hyp t) = (if } t \in \text{set } Hs \ \text{then } \text{Some } t \ \text{else } \text{None})$

lemma *fv-subst-bv1*:

$\text{fv } (\text{subst-bv1 } t \ \text{lev } u) = \text{fv } t \cup (\text{if } \text{loose-bvar1 } t \ \text{lev} \ \text{then } \text{fv } u \ \text{else } \{\})$
by (*induction* $t \ \text{lev } u$ *rule*: *subst-bv1.induct*) (*auto simp add*: *incr-boundvars-def*)

corollary *fv-subst-bvs-upper-bound*:

assumes *is-closed* t
shows $\text{fv } (\text{subst-bvs } us \ t) \subseteq \text{fv } t \cup (\bigcup x \in \text{set } us . (\text{fv } x))$
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*:

$\text{fv } (\text{subst-bvs1 } t \ \text{lev } us) \subseteq \text{fv } t \cup (\bigcup x \in \text{set } us . (\text{fv } x))$

```

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*: $wf\text{-theory } thy \implies t \in \text{axioms } thy \implies \text{typ-of } t = \text{Some prop } T$

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]*: $\text{typ-ok } thy (Tv \text{idn } S) = wf\text{-sort } (\text{subclass } (\text{osig } (\text{sig } thy))) S$

by simp

lemma *typ-ok-contained-tvars-typ-ok*: $\text{typ-ok } thy ty \implies (\text{idn}, S) \in \text{tvs } T ty \implies \text{typ-ok } thy (Tv \text{idn } S)$

by (induction ty) (use split-list typ-ok-Ty **in** $\langle \text{all } \langle \text{fastforce split: option.splits} \rangle \rangle$)

lemma *typ-ok-sig-contained-tvars-typ-ok-sig*:

$\text{typ-ok-sig } \Sigma ty \implies (\text{idn}, S) \in \text{tvs } T ty \implies \text{typ-ok-sig } \Sigma (Tv \text{idn } S)$

by (induction ty) (use split-list typ-ok-sig-Ty **in** $\langle \text{all } \langle \text{fastforce split: option.splits} \rangle \rangle$)

lemma *term-ok'-contained-tvars-typ-ok-sig*:

$\text{term-ok}' \Sigma t \implies (\text{idn}, S) \in \text{tvs } t \implies \text{typ-ok-sig } \Sigma (Tv \text{idn } S)$

proof (induction t)

case (Ct n T)

hence *typ-ok-sig* ΣT

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  $\implies$  (idn, S)  $\in$  tvs t  $\implies$  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  $\implies$   $\forall (-, ty) \in$  set insts . typ-ok thy ty  $\implies$  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  $x \in$  set Ts for  $x$ 

```

```

  by (metis (full-types) 1.premis(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  $x \in$  set Ts for  $x$ 

```

```

    using that 1 by simp

```

```

  then show ?case

```

```

    using 1.premis(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.premis(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.premis(1) by simp
  qed
qed

```

lemma *typ-ok-sig-subst-typ*:

typ-ok-sig Σ $T \implies \forall (-, ty) \in \text{set insts} . \text{typ-ok-sig } \Sigma \text{ ty} \implies \text{typ-ok-sig } \Sigma (\text{subst-typ insts } T)$

proof (*induction insts T rule: subst-typ.induct*)

case (1 insts n Ts)

have *typ-ok-sig* Σ x if $x \in \text{set } Ts$ for x

using 1.premis(1) split-list that *typ-ok-sig-Ty* by fastforce

hence *typ-ok-sig* Σ (*subst-typ insts* x) if $x \in \text{set } Ts$ for x

using that 1 by simp

then show ?case

using 1.premis(1) by (auto simp add: list-all-iff split: option.splits)

next

case (2 insts idn S)

then show ?case

proof(*cases (idn, S) \in set (map fst insts)*)

case True

obtain *ty* where *ty*: *lookup* ($\lambda k. k=(idn,S)$) *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.premis(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.premis(1) by simp

qed

qed

lemma *typ-ok-sig-imp-sortsT-ok-sig*: *typ-ok-sig* Σ $T \implies S \in \text{Sorts } T \implies \text{wf-sort (subclass (osig } \Sigma)) S$

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 $t \implies S \in \text{Sorts } t \implies \text{wf-sort (subclass (osig } \Sigma)) S$

by (*induction t*) (*use typ-ok-sig-imp-sorts T-ok-sig in* \langle (*fastforce split: option.splits*) \rangle)

lemma *replay'-sound-pre*:
assumes *thy: wf-theory thy*

assumes *HS-invs*:
 $\bigwedge x. x \in \text{set } Hs \implies \text{term-ok } thy \ x$
 $\bigwedge x. x \in \text{set } Hs \implies \text{typ-of } x = \text{Some } \text{prop } T$

assumes *ns-invs*:
finite ns
fst ' FV (set Hs) \subseteq ns
fst ' fv-Proof P \subseteq ns

assumes *vs-invs*:
fst ' set vs \subseteq ns

assumes *replay' thy vs ns Hs P = Some res*
shows *thy, (set Hs) \vdash 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 \in 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, {} \vdash res*
using *res 1.premis(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.premis(2) 1.premis(3) by auto*

next
case (*2 thy ux uy Hs n*)
hence *res \in 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* **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' \notin ns* **using** *3.premis variant-variable-fresh names*
by (*metis fst-conv*)

```

hence  $s' \notin \text{fst } \langle FV \text{ (set } Hs) \rangle$  using  $3.\text{prems}$  by blast
hence free:  $(s', T) \notin FV \text{ (set } Hs)$  by force

have typ-ok: wf-type (sig thy)  $T$ 
  using names  $3.\text{prems}$  by (auto split: if-splits)
have  $I:\text{thy}, \text{ set } Hs \vdash \text{bres}$ 
  apply (rule  $3.IH[OF - \text{names}]$ )
  using names  $3.\text{prems}$  apply (solves  $\langle \text{simp split: if-splits} \rangle$ ) +
  using names  $3.\text{prems}$   $\langle ns' = \text{insert } s' \text{ ns} \rangle$  apply fastforce
  using  $3.\text{prems}(7)$   $\langle ns' = \text{insert } s' \text{ ns} \rangle$  apply auto[1]
  using  $3.\text{prems}(8)$   $\langle ns' = \text{insert } s' \text{ ns} \rangle$  apply auto[1]
  using  $3.\text{prems}(6)$  apply fastforce
  using  $3.\text{prems}(7)$   $\langle ns' = \text{insert } s' \text{ ns} \rangle$  apply auto[1]
  using  $3.\text{prems}(8)$   $\langle ns' = \text{insert } s' \text{ ns} \rangle$  apply auto[1]
  using bres by fastforce
  have res:  $\text{res} = \text{mk-all } s' \text{ } T \text{ bres}$  using names bres  $3$  by (auto split: if-splits
prod.splits)
  show ?case using proves.forall-intro[OF  $\langle \text{wf-theory thy} \rangle$   $I$  free typ-ok] res by
simp
next
  case ( $4 \text{ thy vs ns } Hs \text{ } p \text{ } t$ )
  from  $\langle \text{replay}' \text{ thy vs ns } Hs \text{ } (Appt \text{ } p \text{ } t) = \text{Some } \text{res} \rangle$  obtain  $\text{rep } t' \text{ } b \text{ } s \text{ } \text{fun1 } \text{fun2}$ 
propT1 propT2  $\tau \tau'$  where
    conds:  $\text{replay}' \text{ thy vs ns } Hs \text{ } p = \text{Some } \text{rep}$ 
     $t' = \text{subst-bvs } (\text{map } (\lambda(x,y) . Fv \text{ } x \text{ } y) \text{ vs}) \text{ } t$ 
    typ-of  $t' = \text{Some } \tau'$ 
     $\tau = \tau'$ 
    term-ok  $\text{thy } t'$ 
     $s = STR \text{ "Pure.all"} \wedge \text{fun1} = STR \text{ "fun"} \wedge \text{fun2} = STR \text{ "fun"} \wedge \text{propT1} =$ 
 $STR \text{ "prop"} \wedge \text{propT2} = STR \text{ "prop"}$ 
     $\text{rep} = Ct \text{ } s \text{ } (Ty \text{ } \text{fun1} \text{ } [Ty \text{ } \text{fun2} \text{ } [\tau, Ty \text{ } \text{propT1 } Nil], Ty \text{ } \text{propT2 } Nil]) \text{ } \$ \text{ } b$ 
    and  $\text{res} = (b \cdot t')$ 

  by (auto split: term.splits typ.splits list.splits if-splits option.splits simp add:
Let-def)

have ctxt:  $\text{finite } (\text{set } Hs) \forall A \in \text{set } Hs . \text{term-ok } \text{thy } A \forall A \in \text{set } Hs . \text{typ-of } A$ 
 $= \text{Some } \text{propT}$ 
  using  $4$  by auto

show ?case
  using conds  $4.\text{prems } \text{ctxt}$ 
  by (auto simp add: res wt-term-def simp del: FV-def
intro!: forall-elim'[OF  $4.\text{prems}(1) - - - \text{ctxt}$ ]  $4.IH$ )
next
  case ( $5 \text{ thy vs ns } Hs \text{ } t \text{ } p$ )
  from this obtain  $t' \text{ } \text{rep}$  where
    conds:  $\text{subst-bvs } (\text{map } (\lambda(x,y) . Fv \text{ } x \text{ } y) \text{ vs}) \text{ } t = t'$ 

```

```

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.prem1 True by (auto simp add: conds(1) conds(2)[symmetric]
conds(3))
  hence thy, set Hs - {t'} ⊢ t' ⟶ rep
  using implies-intro 5.prem1(1) 5.prem1(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.prem1(5-8) conds(3-4) frees-bound
  by (auto simp add: 5.prem1(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)
 $Option.bind (replay' thy vs ns Hs p2) beta-eta-norm = Some A'$
 $imp = STR "Pure.imp" \wedge fn1 = STR "fun" \wedge fn2 = STR "fun"$
 $\wedge prp1 = STR "prop" \wedge prp2 = STR "prop" \wedge prp3 = STR "prop" \wedge A=A'$
and $res: res = B$
by (*auto split: term.splits typ.splits list.splits if-splits option.splits simp add:*
Let-def)

obtain C **where** $C: Option.bind (replay' thy vs ns Hs p1) beta-eta-norm = Some$
 $(C \mapsto res)$
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** $norm-pre: beta-eta-norm pre = Some (C \mapsto res)$ **by** *simp*
from $pre-C$ pre C **conds** **have** $norm-pre-C: beta-eta-norm pre-C = Some C$ **by**
auto

have $thy, set Hs \vdash pre-C$
by (*rule 6.IH(2)*) (*use 6.prem.s conds in <auto simp add: pre pre-C>*)
hence $I1: thy, set Hs \vdash C$
using *beta-eta-norm-preserved-proves norm-pre-C <wf-theory thy>*
using *6.prem.s(2) 6.prem.s(3)* **by** *blast*

have $thy, set Hs \vdash pre$
by (*rule 6.IH(1)*) (*use 6.prem.s conds in <auto simp add: pre pre-C>*)
hence $I2: thy, set Hs \vdash C \mapsto res$
using *beta-eta-norm-preserved-proves norm-pre <wf-theory thy>*
using *6.prem.s(2) 6.prem.s(3)* **by** *blast*

from $I1$ $I2$ **have** $thy, set Hs \cup set Hs \vdash res$ **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** $thy, set Hs \vdash mk-of-class ty c$
by (*rule proves.of-class[where T=ty, OF 7.prem.s(1)]*) (*use conds in auto*)

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

```

next
  case ( $\delta$  thy ux uy Hs n)
  hence  $res \in set\ Hs$ 
    by (metis not-None-eq option.inject replay'.simps( $\delta$ ))
  then show ?case using proves.assume  $\delta$  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  $\equiv$  Option.bind (replay' thy vs ns Hs P)
beta-eta-norm

```

```

lemma replay''-sound:
  assumes wf-theory thy

```

```

  assumes HS-invs:
     $\bigwedge x. x \in set\ Hs \implies term-ok\ thy\ x$ 
     $\bigwedge x. x \in set\ Hs \implies typ-of\ x = Some\ propT$ 

```

```

  assumes ns-invs:
    finite ns
     $fst\ 'FV\ (set\ Hs) \subseteq ns$ 
     $fst\ 'fv-Proof\ P \subseteq ns$ 

```

```

  assumes vs-invs:
     $fst\ 'set\ vs \subseteq ns$ 

```

```

  assumes replay'' thy vs ns Hs P = Some res
  shows thy, (set Hs)  $\vdash$  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( $\delta$ ) by auto
  moreover have thy, set Hs  $\vdash$  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 []  $\vdash$  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 *hypos* :: *proofterm* \Rightarrow *term list* **where**

hypos (*Abst* - *p*) = *hypos p*
| *hypos* (*AbsP* - *p*) = *hypos p*
| *hypos* (*Appt* *p* -) = *hypos p*
| *hypos* (*AppP* *p1* *p2*) = *List.union* (*hypos p1*) (*hypos p2*)
| *hypos* (*Hyp* *t*) = [*t*]
| *hypos* - = []

lemma *replay''-sound-pre-hyps*:

assumes *wf-theory thy*

assumes $\bigwedge x. x \in \text{set } (\text{hypos } P) \Longrightarrow \text{term-ok } \text{thy } x$

assumes $\bigwedge x. x \in \text{set } (\text{hypos } P) \Longrightarrow \text{typ-of } x = \text{Some propT}$

assumes *replay'' thy* [] (*fst* ' (*fv-Proof* *P* \cup *FV* (*set* (*hypos* *P*)))) (*hypos* *P*) *P* =
Some res

shows *thy, set* (*hypos* *P*) \vdash *res*

apply (*rule replay''-sound*[**where** *vs*=[] **and** *ns*=(*fst* ' (*fv-Proof* *P* \cup *FV* (*set* (*hypos* *P*)))) **and** *P*=*P* **and** *Hs*=*hypos* *P*]

 ; (*use assms finite-fv-Proof replay'-sound-pre in* ' *solves simp*)?)

by *blast+*

definition [*simp*]: *replay thy P* \equiv

(*if* $\forall x \in \text{set } (\text{hypos } P) . \text{term-ok } \text{thy } x \wedge \text{typ-of } x = \text{Some propT}$ *then*

replay'' thy [] (*fst* ' (*fv-Proof* *P* \cup *FV* (*set* (*hypos* *P*)))) (*hypos* *P*) *P* *else None*)

lemma *replay-sound-pre-hyps*:

assumes *wf-theory thy*

assumes *replay thy P* = *Some res*

shows *thy, set* (*hypos* *P*) \vdash *res*

using *replay''-sound-pre-hyps assms* **by** (*simp split: if-splits*)

definition *check-proof thy P res* \equiv *wf-theory thy* \wedge *replay thy P* = *Some res*

lemma *check-proof-sound*:

shows *check-proof thy P res* \Longrightarrow *thy, set* (*hypos* *P*) \vdash *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* (*hypos* *P*) \Vdash *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** $\forall x \in \text{set } (\text{hypos } P) . \text{term-ok } \text{thy } x \wedge \text{typ-of } x = \text{Some propT}$

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*) *execlases* ≡ *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 (execlases 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*: \neg *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* \Rightarrow *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* \equiv (\forall *ars* \in *snd* ' *set arss* .
 \forall *ss*₁ \in *snd* ' *set ars*. \forall *ss*₂ \in *snd* ' *set ars*. *length ss*₁ = *length ss*₂)

lemma *in-alist-imp-in-map-of*: *distinct* (*map fst arss*)
 \Rightarrow (*name*, *ars*) \in *set arss* \Rightarrow *translate-ars arss name* = *Some* (*map-of ars*)
by (*induction arss*) (*auto simp add: rev-image-eqI*)

lemma *exe-ars-conds arss* \Rightarrow \exists *name* . *map-of* (*map* (*apsnd map-of*) *arss*) *name*
= *Some ars*
 \Rightarrow \exists *name arsl* . (*name*, *arsl*) \in *set arss* \wedge *map-of arsl* = *ars*
by (*force simp add: exe-ars-conds-def*)

lemma *exe-ars-conds arss*
 \Rightarrow (*name*, *arsl*) \in *set arss* \wedge *map-of arsl* = *ars*
 \Rightarrow *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 \Rightarrow *consistent-length-tcsigs* (*translate-ars arss*)
 \Rightarrow *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* \in *ran* (*map-of* (*map* (*apsnd map-of*) *arss*)) *ss*₁ \in *ran ars* *ss*₂ \in
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*) \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 ss*₁ *ars c2* = *Some ss*₂
by (*metis in-range-if-ex-key p*(2) *p*(3))

hence $(c1, ss_1) \in \text{set arsl } (c2, ss_2) \in \text{set arsl}$
by (*simp-all add: <map-of arsl = ars> map-of-SomeD*)
hence $\text{length } ss_1 = \text{length } ss_2$
using *assms(2) <(name, arsl) \in 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 \implies
consistent-length-tcsigs (translate-ars arss) \longleftrightarrow 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*
 $\equiv (\forall \text{ars} \in \text{snd } \text{' set arss } .$
 $\forall (c_1, c_2) \in \text{set } cs . c_1 \in \text{fst } \text{' set ars} \longrightarrow c_2 \in \text{fst } \text{' set ars})$

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 \in ran (map-of (map (apsnd map-of) arss))*
(a, b) \in 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) \in set arss map-of arsl = ars*
using *assms(1) by (auto simp add: exe-ars-conds-def)*
hence $(a, y) \in \text{set arsl}$
by (*simp add: map-of-SomeD p(3)*)
hence $\exists y. \text{ars } b = \text{Some } y$
using *assms(2) <(name, arsl) \in 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*: $\text{exe-ars-conds } arss \implies$
 $\text{complete-tcsigs } (\text{set } cs) (\text{translate-ars } arss) \implies \text{exe-complete-tcsigs } cs \text{ } arss$
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*:
 $\text{exe-ars-conds } arss \implies$
 $\text{complete-tcsigs } (\text{set } cs) (\text{translate-ars } arss) \iff \text{exe-complete-tcsigs } cs \text{ } arss$
using *exe-complete-tcsigs-imp-complete-tcsigs complete-tcsigs-imp-exe-complete-tcsigs*
by *blast*

definition *exe-coregular-tcsigs* ($cs :: (\text{class} \times \text{class}) \text{ list}$) $arss$
 $\equiv (\forall ars \in \text{snd } ' \text{ set } arss .$
 $\forall c_1 \in \text{fst } ' \text{ set } ars. \forall c_2 \in \text{fst } ' \text{ set } ars.$
 $(\text{class-leq } (\text{set } cs) \ c_1 \ c_2 \longrightarrow$
 $\text{list-all2 } (\text{sort-leq } (\text{set } cs)) (\text{the } (\text{lookup } (\lambda x. x=c_1) \ ars)) (\text{the } (\text{lookup } (\lambda x.$
 $x=c_2) \ ars))))$

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 $p: ars \in \text{ran } (\text{map-of } (\text{map } (\text{apsnd } \text{map-of}) \ arss)) \ ars \ c_1 = \text{Some } ss1$
 $ars \ c_2 = \text{Some } ss2$
 $\text{class-leq } (\text{set } cs) \ c_1 \ c_2$
from $p(1)$ **obtain** $name$ **where** $\text{map-of } (\text{map } (\text{apsnd } \text{map-of}) \ arss) \ name =$
 $\text{Some } ars$
by (*meson in-range-if-ex-key*)
from this obtain $ar sl$ **where** $(name, ar sl) \in \text{set } arss \ \text{map-of } ar sl = ars$
using *assms(1)* **by** (*auto simp add: exe-ars-conds-def*)
from this obtain $c1 \ c2$ **where** $ars \ c1 = \text{Some } ss1 \ ars \ c2 = \text{Some } ss2 \ \text{class-leq}$
 $(\text{set } cs) \ c1 \ c2$
using $p(2) \ p(3) \ p(4)$ **by** *blast*
hence $(c1, ss1) \in \text{set } ar sl \ (c2, ss2) \in \text{set } ar sl$
by (*simp-all add: <map-of ar sl = ars> map-of-SomeD*)
hence $\text{lookup } (\lambda x. x=c1) \ ar sl = \text{Some } ss1 \ \text{lookup } (\lambda x. x=c2) \ ar sl = \text{Some } ss2$
by (*metis <(name, ar sl) \in \text{set } arss> assms(1) exe-ars-conds-def*
image-eqI lookup-present-eq-key snd-conv)
hence $\text{list-all2 } (\text{sort-leq } (\text{set } cs)) \ ss1 \ ss2$
using *assms(2)* $\langle (name, ar sl) \in \text{set } arss \rangle \langle (c1, ss1) \in \text{set } ar sl \rangle \langle (c2, ss2) \in$
 $\text{set } ar sl \rangle$
 $\langle \text{class-leq } (\text{set } cs) \ c1 \ c2 \rangle$
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 \implies *coregular-tcsigs (set cs) (translate-ars arss)* \iff *exe-coregular-tcsigs cs arss*

using *coregular-tcsigs-imp-exe-coregular-tcsigs exe-coregular-tcsigs-imp-coregular-tcsigs*

by *blast*

lemma *wf-subclass sub* \implies *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* \implies *class-ex (insert (a,b) r) c = (a=c \vee b=c \vee class-ex r c)*

by (*induction r rule: finite-induct*) (*auto simp add: class-ex-def*)

definition [simp]: $\text{execl}\text{-ex rel } c = \text{List.member (exfield rel) } c$

lemma $\text{execl}\text{-ex-code: class-ex (set rel) } c = \text{execl}\text{-ex rel } c$
by (metis *Field-set-code class-ex-def execl}\text{-ex-def in-set-member*)

definition [simp]: $\text{exesort}\text{-ex rel } S = (\forall x \in S . (\text{List.member (exfield rel) } x))$

lemma $\text{sort}\text{-ex-code: sort-ex (set rel) } S = \text{exesort}\text{-ex rel } S$
by (simp add: *execl}\text{-ex-code sort-ex-class-ex*)

definition [simp]: $\text{execl}\text{-les } cs \ c1 \ c2 = (\text{List.member } cs \ (c1, c2) \wedge \neg \text{List.member } cs \ (c2, c1))$

lemma $\text{execl}\text{-les-code: class-les (set } cs) \ c1 \ c2 = \text{execl}\text{-les } cs \ c1 \ c2$
by (simp add: *class-leq-def class-les-def member-def*)

definition [simp]: $\text{exenormalize}\text{-sort } cs \ (s::\text{sort})$
 $= \{c \in s . \neg (\exists c' \in s . \text{execl}\text{-les } cs \ c' \ c)\}$

definition [simp]: $\text{exenormalized}\text{-sort } cs \ s \equiv (\text{exenormalize}\text{-sort } cs \ s) = s$

lemma $\text{normalize}\text{-sort-code}[code]: \text{normalize}\text{-sort (set } cs) \ s = \text{exenormalize}\text{-sort } cs \ s$
by (auto simp add: *normalize-sort-def List.member-def list-ex-iff class-leq-def class-les-def*)

lemma $\text{normalized}\text{-sort-code}[code]: \text{normalized}\text{-sort (set } cs) \ s = \text{exenormalized}\text{-sort } cs \ s$
using *exenormalized-sort-def normalize-sort-code by presburger*

definition [simp]: $\text{exewf}\text{-sort sub } S \equiv \text{exenormalized}\text{-sort sub } S \wedge \text{exesort}\text{-ex sub } S$

lemma $\text{wf}\text{-sort-code:}$
assumes *exe-class-conds sub*
shows $\text{wf}\text{-sort (set sub) } S = \text{exewf}\text{-sort sub } S$
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]: $\text{exewf}\text{-sort sub } S \equiv (S = \{\}) \vee \text{exenormalized}\text{-sort sub } S \wedge \text{exesort}\text{-ex sub } S$
by *simp (smt ball-empty bot-set-def empty-Collect-eq)*

definition $\text{exe-all-normalized-and-ex-tcsigs } cs \ arss$
 $\equiv (\forall ars \in \text{snd ' set } arss . \forall ss \in \text{snd ' set } ars . \forall s \in \text{set } ss . \text{exewf}\text{-sort } cs \ s)$

lemma $\text{all-normalized-and-ex-tcsigs-imp-exe-all-normalized-and-ex-tcsigs:}$
assumes *exe-ars-conds arss all-normalized-and-ex-tcsigs (set } cs) (translate-ars arss)*
shows $\text{exe-all-normalized-and-ex-tcsigs } cs \ arss$
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 \implies all-normalized-and-ex-tcsigs (set cs) (translate-ars arss)
 \longleftrightarrow *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 \times class) list) arss \equiv*
exe-coregular-tcsigs cs arss
 \wedge *exe-complete-tcsigs cs arss*
 \wedge *exe-consistent-length-tcsigs arss*
 \wedge *exe-all-normalized-and-ex-tcsigs cs arss*

lemma *wf-tcsigs-iff-exe-wf-tcsigs*:
exe-ars-conds arss \implies wf-tcsigs (set cs) (translate-ars arss) \longleftrightarrow 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 \times 'a) list \Rightarrow bool **where**
exe-antisym [] \longleftrightarrow True
 $|$ *exe-antisym ((x,y)#r) \longleftrightarrow ((y,x) \in set r \longrightarrow x=y) \wedge exe-antisym r**

lemma *exe-antisym-imp-antisym*: *exe-antisym l \implies antisym (set l)*
by (*induction l*) (*auto simp add: antisym-def*)

lemma *antisym-imp-exe-antisym*: *antisym (set l) \implies 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) \wedge exe-antisym cs \wedge Refl (set cs))*

lemma *wf-classes-iff-exe-wf-classes*: *wf-subclass (set cs) \longleftrightarrow exe-wf-subclass cs*

by (*simp add: antisym-iff-exe-antisym exe-wf-subclass-def*)

definition [*simp*]: *exe-wf-osig oss* \equiv *exe-wf-subclass (execlasses oss)*
 \wedge *exe-wf-tcsigs (execlasses oss) (exetcsigs oss) \wedge exe-osig-conds oss*

lemma *exe-wf-osig-imp-wf-osig*: *exe-wf-osig oss* \implies *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* \implies *subclass (translate-osig oss) = set*
(execlasses oss)
by (*cases oss*) *simp-all*

lemma *tcsigs-translate*: *exe-osig-conds oss*
 \implies *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) \implies *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)* \longleftrightarrow *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* \Rightarrow *typ* \Rightarrow $((\text{variable} \times \text{sort}) \rightarrow \text{typ}) \Rightarrow ((\text{variable} \times \text{sort}) \rightarrow \text{typ}) \text{ option}$
and *raw-matches* :: *typ list* \Rightarrow *typ list* \Rightarrow $((\text{variable} \times \text{sort}) \rightarrow \text{typ}) \Rightarrow ((\text{variable}$

```

× sort) → typ) option
where
raw-match (Tv v S) T subs =
  (case subs (v,S) of
    None ⇒ Some (subs((v,S) := Some T))
  | Some U ⇒ (if U = T then Some subs else None))
| raw-match (Ty a Ts) (Ty b Us) subs =
  (if a=b then raw-matches Ts Us subs else None)
| raw-match - - = None
| raw-matches (T#Ts) (U#Us) subs = Option.bind (raw-match T U subs) (raw-matches
Ts Us)
| raw-matches [] [] subs = Some subs
| raw-matches - - subs = None

```

```

function (sequential) raw-match'
:: typ ⇒ typ ⇒ ((variable × sort) → typ) ⇒ ((variable × sort) → typ) option
where
raw-match' (Tv v S) T subs =
  (case subs (v,S) of
    None ⇒ Some (subs((v,S) := Some T))
  | Some U ⇒ (if U = T then Some subs else None))
| raw-match' (Ty a Ts) (Ty b Us) subs =
  (if a=b ∧ length Ts = length Us
    then fold (λ(T, U) subs . Option.bind subs (raw-match' T U)) (zip Ts Us)
  (Some subs)
  else None)
| raw-match' T U subs = (if T = U then Some subs else None)
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 ≠ length Us ⇒ raw-matches
Ts Us subs = None
by (induction Ts Us subs rule: raw-match-raw-matches.induct(2) [where P =
λT 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 =  $\lambda Ts Us subs . raw-matches Ts Us subs$ 
    = (if length Ts = length Us
      then fold ( $\lambda(T, U) subs . Option.bind subs (raw-match' T U)$ ) (zip Ts Us)
      (Some subs)
      else None)])
case ( $\lambda T Ts U Us subs$ )
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  $\lambda None$  by (induction Ts Us rule: list-induct2) auto
  next
    case False
    then show ?thesis using  $\lambda None$  length-neq-imp-not-raw-matches by auto
  qed
next
  case (Some a)
  then show ?thesis using  $\lambda$  by auto
qed
qed simp-all

```

lemma raw-match'-map-le: raw-match' T U subs = Some subs' \implies map-le subs subs'

```

proof (induction T U subs arbitrary: subs' rule: raw-match'.induct)
  case ( $\lambda a Ts b Us subs$ )
  have length Ts = length Us
    using 2.prem1 by (auto split: if-splits)
  moreover have I:  $(a,b) \in set (zip Ts Us) \implies raw-match' a b subs = Some subs'$ 
 $\implies subs \subseteq_m subs'$ 
    for a b subs subs'
    using 2.prem1 by (auto split: if-splits intro: 2.IH)
  ultimately show ?case using 2.prem1
  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 ($\lambda(T, U) \text{ subs} . \text{Option.bind subs (raw-match' T U)} (zip (x\#xs) (y\#ys))$)
(Some subs) = *Some subs'*

obtains point where

raw-match' x y subs = *Some point*

fold ($\lambda(T, U) \text{ subs} . \text{Option.bind subs (raw-match' T U)} (zip (xs) (ys))$) (*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 ($\lambda(T, U) \text{ subs} . \text{Option.bind subs (raw-match' T U)} (zip (xs@[x]) (ys@[y]))$)
(Some subs) = *Some subs'*

obtains point where

fold ($\lambda(T, U) \text{ subs} . \text{Option.bind subs (raw-match' T U)} (zip (xs) (ys))$) (*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 ($\lambda(T, U) \text{ subs} . \text{Option.bind subs (raw-match' T U)} (zip (x\#xs) (y\#ys))$)
(Some subs) = *Some subs'*

shows *raw-match' x y subs* \sim *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* \sim *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* \sim *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'* \implies
tvsT T \subseteq *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.premis 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.premis(1) Cons.premis(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.premis rest apply blast+
  by (metis Cons.premis(3) list.set-intros(2) zip-Cons-Cons)
  moreover have tvsT x ⊆ dom point
  by (metis Cons.premis(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.premis 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.prem*s(1) *Cons.prem*s(2) *fold-matches-first-step-not-None*
*raw-match'.sims*p(2) *raw-match'-Type-con*ds(2))

from *first* **have** *dom point* \subseteq *tvsT* $x \cup$ *dom subs*
using *Cons.prem*s(3) **by** *fastforce*
moreover **have** *dom subs'* \subseteq *tvsT* (*Ty* a xs) \cup *dom point*
apply (*rule* *Cons.IH*)
using *Cons.prem*s(1) **apply** *simp*
using *Cons.prem*s(2) *rest* **apply** *simp*
by (*metis* *Cons.prem*s(3) *list.set-intros*(2) *zip-Cons-Cons*)

ultimately show *?case* **using** *Cons.prem*s *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'* \implies *dom subs'* = *tvsT* $T \cup$ *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 ($\lambda x. \text{None}$) = *Some subs'* \implies *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'* \implies $p \in$ *tvsT* T
 \implies *subs'* $p \sim =$ *None*

using *raw-match'-dom-res-eq-tvsT* **by** *blast*

lemma *raw-match'-extend-map-preserve*:

raw-match' T U *subs* = *Some subs'* \implies *map-le* *subs'* *subs''* \implies $p \in$ *tvsT* $T \implies$
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* \equiv ($\lambda v S .$ *the-default* (*Tv* v S) (*subs* (v , S)))

lemma *map-eq-on-tvsT-imp-map-eq-on-typ*:

$(\bigwedge p . p \in$ *tvsT* $T \implies$ *subs* $p =$ *subs'* $p)$
 \implies *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'
 \implies *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) \implies raw-match' x y subs = Some subs'
 \implies *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.prem4 fold-matches-first-step-not-None*
l(2) length-Cons raw-match'.sims(2))

have *tsubstT x (convert-subs point) = y*
using *Cons.prem2* **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 \implies 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  $\exists$  lesubs'. raw-match' T U lesubs = Some lesubs'  $\wedge$  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:
   $(x, y) \in \text{set } (\text{zip } Ts \ Us) \implies \text{raw-match}' x y \text{ subs} = \text{Some } \text{subs}'$ 
   $\implies \text{lesubs} \subseteq_m \text{subs} \implies \exists \text{lesubs}'. \text{raw-match}' x y \text{ lesubs} = \text{Some } \text{lesubs}' \wedge \text{lesubs}'$ 
 $\subseteq_m \text{subs}'$ 

```

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.prem4 fold-matches-first-step-not-None *l*(2) length-Cons raw-match'.sims(2))

have $\exists \text{lepoint}. \text{raw-match}' x y \text{ lesubs} = \text{Some } \text{lepoint} \wedge \text{lepoint} \subseteq_m \text{point}$

using *Cons first* **by** *auto*

from *this* **obtain** *lepoint* **where**

comp-lepoint: raw-match' x y lesubs = Some *lepoint* **and** *le-lepoint*: *lepoint*

$\subseteq_m \text{point}$

by *auto*

have $\exists \text{lesubs}'. \text{raw-match}' (Ty a xs) (Ty b ys) \text{lepoint} = \text{Some } \text{lesubs}' \wedge \text{lesubs}'$
 $\subseteq_m \text{subs}'$

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'* $\subseteq_m \text{subs}'$

by *auto*

show *?case* **using** *Cons.premis Cons.hypos 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*: $\text{dom } f = \text{dom } g \implies \text{map-le } f \ g \implies f = g$
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 $\text{dom } \text{subs} \subseteq \text{tvsT } T$
assumes *map-le lesubs subs*
shows *raw-match' T U lesubs = Some subs'*
proof-
have $\text{dom } \text{subs}' = \text{tvsT } T$
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 $\text{dom } \text{lesubs}' = \text{tvsT } T$
using $\langle \text{dom } \text{subs}' = \text{tvsT } T \rangle$ *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' \implies dom subs \subseteq tvsT T \implies raw-match' T U ($\lambda p . \text{None}$) = 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 $\text{tvsT } T \subseteq \text{restriction}$
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 $l: \text{length } Ts = \text{length } Us \ a=b$ **by** (*simp-all split: if-splits*)
from *this 2* **have** *better-IH*:
 $(x, y) \in \text{set } (\text{zip } Ts \ Us) \implies \text{raw-match}' \ x \ y \ \text{subs} = \text{Some } \ \text{subs}' \implies \text{tvsT } x \subseteq \text{restriction}$
 $\implies \text{raw-match}' \ x \ y \ (\text{subs} \ |' \ \text{restriction}) = \text{Some } (\text{subs}' \ |' \ \text{restriction})$
for $x \ y \ \text{subs} \ \text{restriction} \ \text{subs}'$ **by** *simp*

```

from  $l$  better-IH show  $?case$  using  $?$ 
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.prem4 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  $\exists$   $subs$ . raw-match' T2 T1 ( $\lambda p$  .  $None$ ) = Some subs
proof-
  obtain  $\rho$  where tsubstT T2  $\rho$  =  $T1$  using assms tinstT-def by auto
  hence raw-match' T2 T1 ( $\lambda(idx, S)$  . Some ( $\rho$   $idx$   $S$ )) = Some ( $\lambda(idx, S)$  . Some
( $\rho$   $idx$   $S$ ))
  using tsubstT-matcher-imp-raw-match'-unchanged by auto

  hence raw-match' T2 T1 ( $\lambda(idx, S)$  . Some ( $\rho$   $idx$   $S$ ))|'tvsT  $T2$ )
    = Some ( $\lambda(idx, S)$  . Some ( $\rho$   $idx$   $S$ ))|'tvsT  $T2$ )
  using raw-match'-restriction-on-tvsT by simp
  moreover have dom ( $\lambda(idx, S)$  . Some ( $\rho$   $idx$   $S$ ))|'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  $\exists$   $subs$ . raw-match' T2 T1 ( $\lambda p$  .  $None$ ) = Some subs

```

shows $tinstT\ T1\ T2$
proof–
obtain $subs$ **where** $raw-match'\ T2\ T1\ (\lambda p . None) = Some\ subs$
using $assms$ **by** $auto$
hence $tsubstT\ T2\ (convert-sub\ subs) = T1$
using $raw-match'-produces-matcher$ **by** $blast$
thus $?thesis\ unfolding\ tinstT-def$ **by** $fast$
qed

corollary $tinstT-iff-ex-raw-match'$:
 $tinstT\ T1\ T2 \longleftrightarrow (\exists\ subs.\ raw-match'\ T2\ T1\ (\lambda p . None) = Some\ subs)$
using $ex-raw-match'-imp-tinstT\ tinstT-imp-ex-raw-match'$ **by** $blast$

function $(sequential)\ raw-match-term$
 $::\ term \Rightarrow term \Rightarrow ((variable \times sort) \rightarrow typ) \Rightarrow ((variable \times sort) \rightarrow typ)\ option$
where
 $raw-match-term\ (Ct\ a\ T)\ (Ct\ b\ U)\ subs = (if\ a = b\ then\ raw-match'\ T\ U\ subs\ else\ None)$
 $| raw-match-term\ (Fv\ a\ T)\ (Fv\ b\ U)\ subs = (if\ a = b\ then\ raw-match'\ T\ U\ subs\ else\ None)$
 $| raw-match-term\ (Bv\ i)\ (Bv\ j)\ subs = (if\ i = j\ then\ Some\ subs\ else\ None)$
 $| raw-match-term\ (Abs\ T\ t)\ (Abs\ U\ u)\ subs =$
 $\quad Option.bind\ (raw-match'\ T\ U\ subs)\ (raw-match-term\ t\ u)$
 $| 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 $raw-match-term-map-le$: $raw-match-term\ t\ u\ subs = Some\ subs' \implies map-le\ subs\ subs'$
by $(induction\ t\ u\ subs\ arbitrary;\ subs'\ rule:\ raw-match-term.induct)$
 $(auto\ simp\ add:\ bind-eq-Some-conv\ raw-match'-map-le\ simp\ add:\ bind-eq-Some-conv)$

lemma $raw-match-term-tvs-subset-dom-res$:
 $raw-match-term\ t\ u\ subs = Some\ subs' \implies tvs\ t \subseteq dom\ subs'$
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 $4.prem\ by\ (auto\ simp\ add:\ bind-eq-Some-conv\ raw-match'-produces-matcher)$

ultimately **have** $1:\ tvs\ t \subseteq dom\ subs'$
using 4 **by** $fastforce$

from $bsubs$ **have** $tvsT\ T \subseteq dom\ bsubs$
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.prem1 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.prem1* **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:*
 $raw-match-term\ t\ u\ subs = Some\ subs' \implies dom\ subs' \subseteq tvs\ t \cup dom\ subs$
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.prem1* **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.prem1 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.prem5 **by** auto

have 1: dom fsubs \subseteq tvs f \cup dom subs
using 5 f u **by** simp

have dom subs' \subseteq tvs u \cup 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: dom subs' \subseteq tvs f \cup tvs u \cup 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' \implies dom subs' = tvs t \cup 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' \implies map-le subs' subs'' \implies p \in tvs t \implies
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:
(\bigwedge p . p \in tvs t \implies subs p = subs' p)
 \implies tsubst t (convert-subs subs)
= tsubst t (convert-subs subs')
by (induction t) (use map-eq-on-tvsT-imp-map-eq-on-tyt **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'
 \implies 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.prem1 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 \subseteq_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.prem1 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.prem1 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 \subseteq_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 \exists subs. raw-match-term t2 t1 ($\lambda p . \text{None}$) = Some subs
 shows tinst t1 t2

proof–

obtain subs where raw-match-term t2 t1 ($\lambda p . \text{None}$) = 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:

$tsubst t \varrho = u \implies \text{raw-match-term } t \ u \ (\lambda(\text{idx}, S). \text{Some } (\varrho \ \text{idx} \ S)) = \text{Some}$

($\lambda(idx, S). \text{Some } (\varrho \text{ idx } S)$)
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 \subseteq 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
 \langle auto split: option.splits if-splits prod.splits simp add: bind-eq-Some-conv \rangle)

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 $\exists \text{lesubs}'.$ *raw-match-term t u lesubs = Some lesubs' \wedge map-le lesubs' subs'*
using *assms proof (induction t u subs arbitrary: lesubs subs' rule: raw-match-term.induct)*
case ($\text{! } T \text{ t } U \text{ u } \text{subs}$)
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 $\text{!}.$ *prems by (auto simp add: bind-eq-Some-conv raw-match'-produces-matcher)*

from *bsubs ! obtain lesubs where*

lesubs: raw-match' T U subs = Some lesubs map-le lesubs 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 lesubs = Some lesubs' map-le lesubs' subs'

using $\text{!}.$ *prems*

by (*auto split: if-splits simp add: bind-eq-Some-conv raw-match'-produces-matcher*)

show *?case*

using *lesubs lesubs' ! apply (auto split: if-splits simp add: bind-eq-Some-conv)*

by (*meson raw-match'-imp-raw-match'-on-map-le*)

next

case ($\text{! } f \text{ u } f' \text{ u' } \text{subs}$)

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 $\text{!}.$ *prems by auto*

from ! **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.prem*s
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 $\exists \text{subs. raw-match-term } t2 \ t1 \ (\lambda p . \text{None}) = \text{Some subs}$
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 (λp . None) = Some subs)
using *ex-raw-match-term-imp-tinst tinst-imp-ex-raw-match-term by blast*

```

function (sequential) assoc-match
  :: typ  $\Rightarrow$  typ  $\Rightarrow$  ((variable  $\times$  sort)  $\times$  typ) list  $\Rightarrow$  ((variable  $\times$  sort)  $\times$  typ) list
  option where
    assoc-match (Tv v S) T subs =
      (case lookup ( $\lambda x. x=(v,S)$ ) subs of
        None  $\Rightarrow$  Some (((v,S), T) # subs)
        | Some U  $\Rightarrow$  (if U = T then Some subs else None))
  | assoc-match (Ty a Ts) (Ty b Us) subs =
      (if a=b  $\wedge$  length Ts = length Us
        then fold ( $\lambda(T, U)$  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 ( $\lambda(T, U, subs) . size T + size U$ ), 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 ( $\lambda(T, U)$  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 ( $\lambda(T, U)$  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'  $\implies$  set subs  $\subseteq$ 
  set subs'
proof (induction T U subs arbitrary: subs' rule: assoc-match.induct)
  case (2 a Ts b Us subs)

```

```

hence  $l: \text{length } Ts = \text{length } Us \ a = b$  by (simp-all split: if-splits)
have better-IH:  $(x, y) \in \text{set } (\text{zip } Ts \ Us) \implies$ 
   $\text{assoc-match } x \ y \ \text{subs} = \text{Some } \text{subs}' \implies \text{set } \text{subs} \subseteq \text{set } \text{subs}'$ 
  for  $x \ y \ \text{subs} \ \text{subs}'$  using 2 by (simp split: if-splits)
from  $l$  better-IH 2.prem1 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:  $\text{assoc-match } x \ y \ \text{subs} = \text{Some } \text{point}$ 
  and rest:  $\text{assoc-match } (Ty \ a \ xs) \ (Ty \ b \ ys) \ \text{point} = \text{Some } \text{subs}'$ 
  using fold-assoc-matches-first-step-not-None
  by (metis (no-types, lifting) Cons.hyps Cons.prem1 assoc-match.simps(2))
assoc-match-Type-conds(2)

  then show ?case
    using Cons.IH Cons.prem1(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:  $\text{assoc-match } T \ U \ \text{subs} = \text{Some } \text{subs}' \implies \text{distinct}$ 
(map fst subs)
 $\implies \text{distinct } (\text{map } \text{fst } \text{subs}')$ 
proof (induction T U subs arbitrary: subs' rule: assoc-match.induct)
  case ( $2 \ a \ Ts \ b \ Us \ \text{subs}$ )
  hence  $l: \text{length } Ts = \text{length } Us \ a = b$  by (simp-all split: if-splits)
  have better-IH:  $(x, y) \in \text{set } (\text{zip } Ts \ Us) \implies$ 
     $\text{assoc-match } x \ y \ \text{subs} = \text{Some } \text{subs}' \implies \text{distinct } (\text{map } \text{fst } \text{subs}) \implies \text{distinct}$ 
    (map fst subs')
  for  $x \ y \ \text{subs} \ \text{subs}'$  using 2 by (simp split: if-splits)
  from  $l$  better-IH 2.prem1 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:  $\text{assoc-match } x \ y \ \text{subs} = \text{Some } \text{point}$ 
    and rest:  $\text{assoc-match } (Ty \ a \ xs) \ (Ty \ b \ ys) \ \text{point} = \text{Some } \text{subs}'$ 
    using fold-assoc-matches-first-step-not-None
    by (metis (no-types, lifting) Cons.hyps Cons.prem1 assoc-match.simps(2))
assoc-match-Type-conds(2)

    have dst-point:  $\text{distinct } (\text{map } \text{fst } \text{point})$ 
    apply (rule Cons.prem1)

```

```

    using first Cons.premis by simp-all

  have distinct (map fst subs')
  apply (rule Cons.IH)
  using Cons.premis rest apply simp
  using Cons.premis apply auto[1]
  using rest apply simp
  using dst-point apply simp
  done

  then show ?case
  using Cons.IH Cons.premis(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.prem
    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.prem 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.prem 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.prem
          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.prem 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.premis(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'  $\implies \forall x \in \text{dom } m . m x = m' x \implies m = m'$ 
by (simp add: map-le-def map-le-same-dom-imp-same-map)

lemma list-of-map:
assumes finite (dom subs)
shows  $\exists l . \text{map-of } l = \text{subs}$ 
proof-
have finite  $\{(k, \text{the } (\text{subs } k)) \mid k . k \in \text{dom } \text{subs}\}$  using assms by simp
from this obtain l where l: set l =  $\{(k, \text{the } (\text{subs } k)) \mid k . k \in \text{dom } \text{subs}\}$ 
using finite-list by fastforce

hence dom (map-of l) = fst '  $\{(k, \text{the } (\text{subs } k)) \mid k . k \in \text{dom } \text{subs}\}$ 
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  $x \in \text{dom } \text{subs}$  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  $\longleftrightarrow$  assoc-match T2 T1 []
 $\sim = \text{None}$ 
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  $\Rightarrow$  term  $\Rightarrow$  ((variable  $\times$  sort)  $\times$  typ) list  $\Rightarrow$  ((variable  $\times$  sort)  $\times$  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]: $tinst\ t1\ t2 \longleftrightarrow assoc-match-term\ t2\ t1$
 $\square \neq None$

```

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 (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)))

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

```

$\wedge (\forall type \in fst \text{ ' set (exetcsigs sa).$
 $(\forall ars \in snd \text{ ' set (the (lookup (\lambda k. k=type) (exetcsigs sa))) .$
 $the (lookup (\lambda k. k=type) tao) = length ars)))$
 $\wedge (\forall ty \in snd \text{ ' set cto . typ-ok-sig (map-of cto, map-of tao, translate-osig sa) ty)$
shows $wf\text{-sig (map-of cto, map-of tao, translate-osig sa)}$
proof–
{
 fix $type\ y$
 assume $p: exe\text{-osig}\text{-conds sa trans (fst (translate-osig sa)) snd (translate-osig sa) type = Some y}$
 hence $exe\text{-ars}\text{-conds (exetcsigs sa)}$
 using $exe\text{-osig}\text{-conds}\text{-def by blast}$
 from p **have** $translate\text{-ars (exetcsigs sa) type = Some y}$
 by $(metis snd\text{-conv translate-osig.elims)}$
 hence $(type, y) \in set (map (apsnd map-of) (exetcsigs sa))$
 using $map\text{-of}\text{-SomeD by force}$
 hence $type \in fst \text{ ' set (exetcsigs sa) by force}$
 from this obtain x **where** $lookup (\lambda x. x=type) (exetcsigs sa) = Some x$
 using $key\text{-present}\text{-imp}\text{-eq}\text{-lookup}\text{-finds}\text{-value by metis}$
 hence $map\text{-of } x = y$
 by $(metis \langle exe\text{-ars}\text{-conds (snd sa) \rangle \langle translate\text{-ars (snd sa) type = Some y \rangle$
 $exe\text{-ars}\text{-conds}\text{-def in}\text{-alist}\text{-imp}\text{-in}\text{-map}\text{-of lookup}\text{-eq}\text{-map}\text{-of}\text{-ap}$
 $map\text{-of}\text{-SomeD option.sel})$
 have $\exists y. (type, y) \in set tao$
 using $\langle type \in fst \text{ ' set (exetcsigs sa) \rangle assms(4) by auto$
}
note $1 = this$

{
 fix $ars\ type\ y$
 assume $p: exe\text{-osig}\text{-conds sa}$
 $trans (fst (translate-osig sa))$
 $\forall x \in set\ cto. typ\text{-ok}\text{-sig (map-of cto, map-of tao, translate-osig sa) (snd x)}$
 $ars \in ran\ y$
 $snd (translate-osig sa) type = Some\ y$

 hence $exe\text{-ars}\text{-conds (exetcsigs sa)}$
 using $exe\text{-osig}\text{-conds}\text{-def by blast}$
 from $p(1-2) p(5)$ **have** $translate\text{-ars (exetcsigs sa) type = Some y}$
 by $(metis snd\text{-conv translate-osig.elims)}$
 hence $(type, y) \in set (map (apsnd map-of) (exetcsigs sa))$
 using $map\text{-of}\text{-SomeD by force}$
 hence $dom: type \in fst \text{ ' set (exetcsigs sa) by force}$
 from this obtain x **where** $x: lookup (\lambda x. x=type) (exetcsigs sa) = Some x$
 using $key\text{-present}\text{-imp}\text{-eq}\text{-lookup}\text{-finds}\text{-value by metis}$
 hence $map\text{-of } x = y$
 by $(metis \langle exe\text{-ars}\text{-conds (snd sa) \rangle \langle translate\text{-ars (snd sa) type = Some y \rangle$
 $exe\text{-ars}\text{-conds}\text{-def in}\text{-alist}\text{-imp}\text{-in}\text{-map}\text{-of lookup}\text{-eq}\text{-map}\text{-of}\text{-ap map}\text{-of}\text{-SomeD}$
 $option.sel)$
}

```

have  $ars \in snd \text{ ' set } x$ 
  by (metis  $\langle map\text{-of } x = y \rangle image\text{-iff } in\text{-range-if-ex-key } map\text{-of-SomeD } p(4)$ 
snd-conv)

have  $type \in fst \text{ ' set } tao$ 
  apply (simp add:  $\langle type \in fst \text{ ' set } (exetcsigs \ sa) \rangle assms(4)$ )
  using  $assms(4)$  dom by blast
moreover have 1: ( $\forall ars \in snd \text{ ' set } (the (lookup (\lambda k. k=type) (exetcsigs$ 
sa))) .
   $the (lookup (\lambda k. k=type) \ tao) = length \ ars$ 
  using  $\langle type \in fst \text{ ' set } (exetcsigs \ sa) \rangle assms(4)$  by blast

ultimately have  $the (lookup (\lambda k. k = type) \ tao) = length \ ars$ 
  using  $\langle lookup (\lambda x. x = type) (exetcsigs \ sa) = Some \ x \rangle \langle map\text{-of } x = y \rangle$ 
   $in\text{-range-if-ex-key } map\text{-of-SomeD } option.sel \ p(3) \ snd\text{-conv}$ 
  by (simp add: 1  $\langle ars \in snd \text{ ' set } x \rangle$ )
  hence  $the (map\text{-of } tao \ type) = length \ ars$ 
  by (metis  $\langle the (lookup (\lambda k. k = type) \ tao) = length \ ars \rangle lookup\text{-eq-map-of-ap}$ )
}
note 2 = this
{
  fix  $a \ b \ x \ y$ 
  assume  $p: fst \text{ ' set } b = fst \text{ ' set } tao$ 
   $(x, y) \in set \ tao$ 
   $sa = (a, b)$ 

  have  $x \in fst \text{ ' set } b$ 
  by (metis  $fst\text{-conv } image\text{-iff } p(1) \ p(2)$ )
  from this obtain  $ars$  where  $lookup (\lambda k. k=x) \ b = Some \ ars$ 
  by (metis  $key\text{-present-imp-eq-lookup-finds-value}$ )
  hence  $(x, ars) \in set \ b$ 
  by (simp add:  $lookup\text{-present-eq-key}'$ )
  hence  $lookup (\lambda k. k=x) (map (apsnd \ map\text{-of}) \ b) = Some (map\text{-of } ars)$ 
  by (metis  $assms(3) \ exe\text{-ars-conds-def } exe\text{-osig-conds-def } in\text{-alist-imp-in-map-of}$ 
   $lookup\text{-eq-map-of-ap } p(3) \ snd\text{-conv } translate\text{-ars.simps}$ )
  hence  $\exists y. map\text{-of } (map (apsnd \ map\text{-of}) \ b) \ x = Some \ y$ 
  by (metis  $lookup\text{-eq-map-of-ap}$ )
}
note 3 = this
{
  fix  $a \ b \ x$ 
  assume  $p: alist\text{-conds } cto$ 
   $x \in ran (map\text{-of } cto)$ 
   $sa = (a, b)$ 
  have  $typ\text{-ok-sig } (map\text{-of } cto, map\text{-of } tao, set \ a, map\text{-of } (map (apsnd \ map\text{-of})$ 
b)) \ x
  using  $assms(4) \ p(1) \ p(2) \ p(3) \ ran\text{-distinct}$  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
   $\wedge$  fst ' set (exetsigs sa) = fst ' set tao
   $\wedge$  ( $\forall$  type  $\in$  fst ' set (exetsigs sa).
    ( $\forall$  ars  $\in$  snd ' set (the (lookup ( $\lambda$ k. k=type) (exetsigs sa))) .
      the (lookup ( $\lambda$ k. k=type) tao) = length ars)))
   $\wedge$  ( $\forall$  ty  $\in$  snd ' set cto . typ-ok-sig (map-of cto, map-of tao, translate-osig sa)
ty)
proof-
{
  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)  $\in$  set b

  hence x  $\in$  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  $\times$  (String.literal  $\times$  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)  $\in$  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  $\in$  fst ' set (map (apsnd map-of) tcs)
  by (meson domI domIff map-of-eq-None-iff)
  then have n  $\in$  fst ' set tcs
  by force
}
note 2 = this
{
  fix cl tcs n K c Ss

```

assume $p: (n, K) \in \text{set } tcs$
 $(c, Ss) \in \text{set } (\text{the } (\text{lookup } (\lambda k. k = n) \text{ } tcs))$
 $\text{exe-ars-conds } tcs$
 $\text{dom } (\text{map-of } (\text{map } (\text{apsnd } \text{map-of}) \text{ } tcs)) = \text{dom } (\text{map-of } \text{tao})$
 $\forall \text{type} \in \text{dom } (\text{map-of } \text{tao}). \forall \text{ars} \in \text{ran } (\text{the } (\text{map-of } (\text{map } (\text{apsnd } \text{map-of}) \text{ } tcs) \text{ } \text{type}))$.
 $\text{the } (\text{map-of } \text{tao } \text{type}) = \text{length } \text{ars}$

have 1: $\text{translate-ars } tcs \ n = \text{Some } (\text{map-of } K)$
using $\text{exe-ars-conds-def in-alist-imp-in-map-of } p(1-3)$ **by** blast
have 2: $\text{map-of } K \ c = \text{Some } Ss$
using $p(1-3)$
by $(\text{metis } \text{Some-eq-map-of-iff } \text{exe-ars-conds-def } \text{image-iff } \text{lookup-eq-map-of-ap } \text{option.sel } \text{snd-conv})$
have $\text{the } (\text{lookup } (\lambda k. k = n) \ \text{tao}) = \text{length } Ss$
using 1 2 $p(4,5)$
by $(\text{metis } \text{domIff } \text{lookup-eq-map-of-ap } \text{option.distinct}(1) \ \text{option.sel } \text{ranI } \text{translate-ars.simps})$
}
note 3 = this

have 1: $\text{wf-osig } (\text{translate-osig } sa) \ \text{dom } (tcsigs \ (\text{translate-osig } sa)) = \text{dom } (\text{map-of } \text{tao})$
 $(\forall \text{type} \in \text{dom } (tcsigs \ (\text{translate-osig } sa))).$
 $(\forall \text{ars} \in \text{ran } (\text{the } (tcsigs \ (\text{translate-osig } sa) \ \text{type}))) \ . \ \text{the } ((\text{map-of } \text{tao}) \ \text{type}) =$
 $\text{length } \text{ars})$
 $(\forall \text{ty} \in \text{Map.ran } (\text{map-of } \text{cto}) \ . \ \text{wf-type } (\text{map-of } \text{cto}, \ \text{map-of } \text{tao}, \ \text{translate-osig } sa) \ \text{ty})$
using $\text{assms}(4)$ **by** auto
note $\text{pre} = 1$

have $\text{exe-wf-osig } sa$
using 1(1) $\text{wf-osig-iff-exe-wf-osig}$ **by** blast
moreover **have** $\text{fst } ' \ \text{set } (\text{snd } sa) = \text{fst } ' \ \text{set } \text{tao}$
proof
show $\text{fst } ' \ \text{set } (\text{snd } sa) \subseteq \text{fst } ' \ \text{set } \text{tao}$
using $\text{assms}(3-4)$
by $(\text{clarsimp } \text{simp } \text{add: } \text{dom-map-of-conv-image-fst } \text{exe-ars-conds-def } \text{exe-osig-conds-def})$
 $(\text{metis } \text{tcsigs-translate } \text{assms}(3) \ \text{domIff } \text{in-alist-imp-in-map-of } \text{option.simps}(3))$
next
show $\text{fst } ' \ \text{set } (\text{snd } sa) \supseteq \text{fst } ' \ \text{set } \text{tao}$
using 1(2) 2 $\text{assms}(2-3)$ tcsigs-translate **by** auto
qed
moreover **have** $(\forall \text{type} \in \text{fst } ' \ \text{set } (\text{snd } sa). \forall \text{ars} \in \text{snd } ' \ \text{set } (\text{the } (\text{lookup } (\lambda k. k = \text{type}) \ (\text{snd } sa))))$.
 $\text{the } (\text{lookup } (\lambda k. k = \text{type}) \ \text{tao}) = \text{length } \text{ars})$
proof $(\text{standard+}, \ \text{goal-cases})$
case $(1 \ n \ Ss)$
obtain c **where** $c: (c, Ss) \in \text{set } (\text{the } (\text{lookup } (\lambda k. k = n) \ (\text{snd } sa)))$

using 1(2) **by** *force*
have $\text{dom} (\text{map-of} (\text{map} (\text{apsnd} \text{map-of}) (\text{snd} \text{sa}))) = \text{dom} (\text{map-of} \text{tao})$
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 ($\lambda k. k = n$) (snd sa)) snd sa
c])+
qed
moreover **have** $(\forall ty \in \text{Map.ran} (\text{map-of} \text{cto}) . \text{wf-type} (\text{map-of} \text{cto}, \text{map-of} \text{tao},$
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 \implies alist-conds tao \implies exe-osig-conds*
sa
 $\implies \text{wf-sig} (\text{map-of} \text{cto}, \text{map-of} \text{tao}, \text{translate-osig} \text{sa}) = (\text{exe-wf-osig} \text{sa}$
 $\wedge \text{fst} \text{' set } (\text{exetcSIGs} \text{sa}) = \text{fst} \text{' set } \text{tao}$
 $\wedge (\forall \text{type} \in \text{fst} \text{' set } (\text{exetcSIGs} \text{sa}).$
 $(\forall \text{ars} \in \text{snd} \text{' set } (\text{the} (\text{lookup} (\lambda k. k = \text{type}) (\text{exetcSIGs} \text{sa}))) .$
 $\text{the} (\text{lookup} (\lambda k. k = \text{type}) \text{tao}) = \text{length} \text{ars})) .$
 $\wedge (\forall ty \in \text{snd} \text{' set } \text{cto} . \text{typ-ok-sig} (\text{map-of} \text{cto}, \text{map-of} \text{tao}, \text{translate-osig} \text{sa}) ty))$
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 \implies alist-conds tao \implies exe-osig-conds*
sa
 $\implies \text{wf-sig} (\text{map-of} \text{cto}, \text{map-of} \text{tao}, \text{translate-osig} \text{sa})$
 $\longleftrightarrow \text{exe-wf-sig} (\text{ExeSignature} \text{cto} \text{tao} \text{sa})$
unfolding *exe-wf-sig.simps*
using *wf-sig-iff-exe-wf-sig-pre* **by** *presburger*

fun *translate-signature* :: *exesignature* \Rightarrow *signature* **where**
translate-signature (*ExeSignature* *cto* *tao* *sa*)
 $= (\text{map-of} \text{cto}, \text{map-of} \text{tao}, \text{translate-osig} \text{sa})$

fun *exetyp-ok-sig* :: *exesignature* \Rightarrow *typ* \Rightarrow *bool* **where**
exetyp-ok-sig Σ (*Ty* *c* *Ts*) = (*case* *lookup* ($\lambda k. k = c$) (*exetyp-arity-of* Σ) *of*
None \Rightarrow *False*
| *Some* *ar* \Rightarrow *length* *Ts* = *ar* \wedge *list-all* (*exetyp-ok-sig* Σ) *Ts*)
| *exetyp-ok-sig* Σ (*Tv* - *S*) = *exewf-sort* (*exeClasses* (*exesorts* Σ)) *S*

thm *exewf-sort-def*

definition [*simp*]: *exesort-ok-sig* Σ *S* \equiv *exesort-ex* (*exeClasses* (*exesorts* Σ)) *S*
 \wedge *exenormalized-sort* (*exeClasses* (*exesorts* Σ)) *S*

lemma *typ-arity-lookup-code: type-arity* (*translate-signature* Σ) *n* = *lookup* ($\lambda k. k$

= 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 (*exetcsgs sa*) = *fst* ' set *tao*
 \wedge (\forall *type* \in *fst* ' set (*exetcsgs sa*).
 (\forall *ars* \in *snd* ' set (the (lookup ($\lambda k. k=type$) (*exetcsgs 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)

\wedge *exe-osig-conds* (*exesorts* Σ)

abbreviation *illformed-theory* \equiv ((*Map.empty*, *Map.empty*, *illformed-osig*), {})

lemma *illformed-theory-not-wf-theory*: \neg *wf-theory* *illformed-theory*
by *simp*

fun *translate-theory* :: *exetheory* \Rightarrow *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*)
 \longleftrightarrow
exe-sig-conds (*ExeSignature* *cto tao sa*) \wedge
 $(\forall p \in \text{set } ax . \text{term-ok } (\text{translate-theory } (\text{ExeTheory } (\text{ExeSignature } \text{cto } \text{tao } \text{sa}) \text{ ax})) p \wedge \text{typ-of } p = \text{Some propT})$
 \wedge *is-std-sig* (*translate-signature* (*ExeSignature* *cto tao sa*))
 \wedge *exe-wf-sig* (*ExeSignature* *cto tao sa*)
 \wedge *eq-axs* \subseteq *set ax*

lemma *wf-sig-iff-exe-wf-sig'*: *exe-sig-conds* $\Sigma \Longrightarrow$
wf-sig (*translate-signature* Σ) \longleftrightarrow
exe-wf-sig Σ
by (*metis exe-sig-conds-def exesignature.exhaust-sel wf-sig-iff-exe-wf-sig translate-signature.simps*)

lemma *wf-sig-imp-exe-wf-sig'*: *exe-sig-conds* $\Sigma \Longrightarrow$
wf-sig (*translate-signature* Σ) \Longrightarrow
exe-wf-sig Σ
by (*metis exe-sig-conds-def exesignature.exhaust-sel wf-sig-iff-exe-wf-sig translate-signature.simps*)

lemma *exe-wf-sig-imp-wf-sig'*: *exe-sig-conds* $\Sigma \Longrightarrow$
exe-wf-sig Σ
 \Longrightarrow *wf-sig* (*translate-signature* Σ)
by (*metis exe-sig-conds-def exesignature.exhaust-sel wf-sig-iff-exe-wf-sig translate-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*))
 \longleftrightarrow *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
 $(\forall ty \in \text{Map.ran } (\text{map-of } (\text{execonst-type-of } (\text{exesig } a))) . \text{typ-ok-sig } (\text{translate-signature } (\text{exesig } a)) \text{ ty})$
 $\longleftrightarrow (\forall ty \in \text{snd 'set } (\text{execonst-type-of } (\text{exesig } a)) . \text{typ-ok-sig } (\text{translate-signature } (\text{exesig } a)) \text{ ty})$
by (*simp add: exe-sig-conds-def ran-distinct*)
moreover have *wf-sig (translate-signature (exesig a))*
 $\longleftrightarrow \text{exe-wf-sig } (\text{exesig } a)$
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*:
 $\text{wf-theory } (\text{translate-theory } a) \longleftrightarrow \text{exe-wf-theory } a$
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) \longleftrightarrow*
 $\text{lookup } (\lambda k. k = \text{STR "fun"}) \text{ tao} = \text{Some } 2 \wedge \text{lookup } (\lambda k. k = \text{STR "prop"}) \text{ tao} = \text{Some } 0$
 $\wedge \text{lookup } (\lambda k. k = \text{STR "itself"}) \text{ tao} = \text{Some } 1$
 $\wedge \text{lookup } (\lambda k. k = \text{STR "Pure.eq"}) \text{ cto} = \text{Some } ((\text{Tv } (\text{Var } (\text{STR "'a'"}, 0)) \text{ full-sort}) \rightarrow ((\text{Tv } (\text{Var } (\text{STR "'a'"}, 0)) \text{ full-sort}) \rightarrow \text{propT}))$
 $\wedge \text{lookup } (\lambda k. k = \text{STR "Pure.all"}) \text{ cto} = \text{Some } ((\text{Tv } (\text{Var } (\text{STR "'a'"}, 0)) \text{ full-sort} \rightarrow \text{propT}) \rightarrow \text{propT})$
 $\wedge \text{lookup } (\lambda k. k = \text{STR "Pure.imp"}) \text{ cto} = \text{Some } (\text{propT} \rightarrow (\text{propT} \rightarrow \text{propT}))$
 $\wedge \text{lookup } (\lambda k. k = \text{STR "Pure.type"}) \text{ cto} = \text{Some } (\text{itselfT } (\text{Tv } (\text{Var } (\text{STR "'a'"}, 0)) \text{ full-sort}))$

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) \longleftrightarrow*
 $\text{exe-sig-conds } (\text{ExeSignature } \text{cto } \text{tao } \text{sa}) \wedge$
 $(\forall p \in \text{set } \text{ax} . \text{exeterm-ok' } (\text{ExeSignature } \text{cto } \text{tao } \text{sa}) \text{ p} \wedge \text{typ-of } \text{p} = \text{Some } \text{propT})$
 $\wedge \text{exeis-std-sig } (\text{ExeSignature } \text{cto } \text{tao } \text{sa})$
 $\wedge \text{exe-wf-sig } (\text{ExeSignature } \text{cto } \text{tao } \text{sa})$
 $\wedge \text{eq-axs} \subseteq \text{set } \text{ax}$

```

lemma term-ok'-code:
  assumes exe-osig-conds (exesorts (ExeSignature cto tao sa))
  shows (term-ok' (translate-signature (ExeSignature cto tao sa)) p  $\wedge$  typ-of p =
    Some propT)
    = (exeterm-ok' (ExeSignature cto tao sa) p  $\wedge$  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  $\wedge$ 
    typ-of p = Some propT)
    = (term-ok' (translate-signature (ExeSignature cto tao sa)) p  $\wedge$  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( $\forall p \in \text{set } ax . \text{term-ok} (\text{translate-theory} (\text{ExeTheory} (\text{ExeSignature } \text{cto } \text{tao } \text{sa}) \text{ax}))$  p  $\wedge$ 
    typ-of p = Some propT)
    = ( $\forall p \in \text{set } ax . \text{exeterm-ok}' (\text{ExeSignature } \text{cto } \text{tao } \text{sa})$  p  $\wedge$  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  $\Theta$  = exe-wf-theory'  $\Theta$ 
  apply (cases  $\Theta$  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  $\Theta \equiv \text{exetyp-ok-sig} (\text{exesig } \Theta)$ 

lemma typ-ok-code:
  assumes exe-wf-theory'  $\Theta$ 
  shows typ-ok (translate-theory  $\Theta$ ) ty = exetyp-ok  $\Theta$  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 = ($\forall c_2 \in s2 . \exists c_1 \in s1. \text{execlass-leq sub } c_1 c_2$)

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* \Rightarrow *typ* \Rightarrow *sort* \Rightarrow *bool* **where**
exehas-sort oss (Tv - S) S' = *exesort-leq* (*execlasses* oss) S S' |
exehas-sort oss (Ty a Ts) S =
 (case *lookup* ($\lambda k. k=a$) (*exetcsigs* oss) of
 None \Rightarrow False |
 Some mgd \Rightarrow ($\forall C \in S.$
 case *lookup* ($\lambda k. k=C$) mgd of
 None \Rightarrow False
 | Some Ss \Rightarrow *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 \Longrightarrow *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.premis 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 \in S

have l: *lookup* ($\lambda k. k=n$) (*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 $\exists x. (\text{lookup } (\lambda k. k=n) \text{ tcs}) = \text{Some } x$

by (*induction* tcs) *auto*

from this obtain pre-mgd **where** pre-mgd: (*lookup* ($\lambda k. k=n$) 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* ($\lambda k. k=c$) pre-mgd = Some Ss

using Ty.premis asm **by** (*auto simp add: pre-mgd split: option.splits*)

hence cond: *list-all2* (*exehas-sort* (sub,tcs)) Ts Ss

using \langle *exehas-sort* (sub, tcs) (Ty n Ts) S \rangle 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  $\exists Ss. mgd\ c = Some\ Ss \wedge list\text{-all2}\ (has\text{-sort}\ (sub',\ tcs'))\ Ts\ Ss$ 
  using cond Ty.IH list.rel-mono-strong sub'-tcs' by force
qed
next
  case (Tv n S)
  then show ?case
    by (metis assms exe-has-sort.simps(1) exesort-les-code has-sort-Tv prod.collapse
translate-osig.simps)
qed

```

lemma *has-sort-imp-exe-has-sort:*

```

assumes exe-osig-conds (sub, tcs)
shows has-sort (translate-osig (sub, tcs)) T S  $\implies$  exe-has-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.prem sub'-tcs' has-sort.simps by (auto split: option.splits)
    hence lookup ( $\lambda k. k=n$ ) (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 ( $\lambda k. k=n$ ) (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  $\exists x. (lookup\ (\lambda k. k=n)\ tcs) = Some\ x$ 
      by (induction tcs) auto
    from this obtain pre-mgd where pre-mgd: (lookup ( $\lambda k. k=n$ ) 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 ( $\lambda k. k=c$ ) pre-mgd = Some Ss
    using  $\langle c \in S \rangle \langle map\text{-of}\ pre\text{-mgd} = mgd \rangle sub'\text{-}tcs'\ mgd\ assms\ Ty.prem
has-sort.simps
    by (auto simp add: dom-map-of-conv-image-fst domIff eq-fst-iff exe-ars-conds-def
map-of-eq-None-iff classes-translate lookup-eq-map-of-ap split: typ.splits)$ 
```

```

      dest!: domD intro!: domI)
  have l: length Ts = length Ss
  using asm mgd pre-mgd Ty.premss assms sub'-tcs' Ss list-all2-lengthD pre-mgd-mgd
  by (fastforce simp add: has-sort.simps lookup-eq-map-of-ap)

  have 1:  $\forall c \in S. \exists Ss . \text{mgd } c = \text{Some } Ss \wedge \text{list-all2 } (\text{has-sort } (sub', tcs')) Ts Ss$ 
  using mgd Ty.premss has-sort.simps sub'-tcs' by auto

  have cond: list-all2 (exe-has-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.premss 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 ( $\forall C \in S.$ 
  case lookup ( $\lambda k. k=C$ ) pre-mgd of
  | None  $\Rightarrow$  False
  | Some Ss  $\Rightarrow$  list-all2 (exe-has-sort (sub,tcs)) Ts Ss)
  by (metis 1 Ty.IH list-all2-conv-all-nth lookup-eq-map-of-ap nth-mem option.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 = exe-has-sort oss T S
  by (metis assms exe-has-sort-imp-has-sort has-sort-imp-exe-has-sort prod.collapse)

lemma has-sort-code':
  assumes exe-wf-theory'  $\Theta$ 
  shows has-sort (osig (sig (translate-theory  $\Theta$ ))) T S
  = exe-has-sort (exesorts (exesig  $\Theta$ )) T S
  apply (cases  $\Theta$  rule: exetheory-full-exhaust) using assms has-sort-code by auto

abbreviation exeinst-ok  $\Theta$  insts  $\equiv$ 
  distinct (map fst insts)
   $\wedge$  list-all (exetyp-ok  $\Theta$ ) (map snd insts)

```

\wedge list-all ($\lambda((idn, S), T) . exehas-sort (exesorts (exesig \Theta)) T S$) 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 ($\lambda((idn, S), T) . has-sort (osig (sig (translate-theory \Theta))) T S$) insts

= list-all ($\lambda((idn, S), T) . exehas-sort (exesorts (exesig \Theta)) T S$) 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* \equiv *exeterm-ok'* (*exesig* Θ) *t* \wedge *typ-of* *t* \neq *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* \Rightarrow (*variable* \times *typ*) *list* \Rightarrow *variable set*

\Rightarrow *term list* \Rightarrow *proofterm* \Rightarrow *term option* **where**

exereplay' *thy* - - *Hs* (*P* λ *m t Tis*) = (*if* *exeinst-ok* *thy Tis* \wedge *exeterm-ok* *thy t* *then* *if* *t* \in *set* (*exearioms-of* *thy*)

then *Some* (*forall-intro-vars* (*subst-typ'* *Tis* *t*) [])

else *None* *else* *None*)

| *exereplay'* *thy* - - *Hs* (*P**Bound* *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* ($\lambda(x,y) . Fv$ *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') \Rightarrow

if *s* = *STR* "*Pure.all*" \wedge *fun1* = *STR* "*fun*" \wedge *fun2* = *STR* "*fun*"

\wedge *propT1* = *STR* "*prop*" \wedge *propT2* = *STR* "*prop*"

\wedge $\tau = \tau' \wedge$ *exeterm-ok* *thy t'*

then *Some* (*b* \cdot *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 = variable 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 \longleftrightarrow exeinst-ok thy tys*

by (*simp add: assms inst-ok-code1 inst-ok-code2*)

moreover have *tok: term-ok (translate-theory thy) ax \longleftrightarrow exeterm-ok thy ax*

using *assms term-ok-code* **by** *blast*

moreover have *ax: ax \in axioms (translate-theory thy) \longleftrightarrow ax \in set (exearioms-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 *\langle auto simp only: replay'.simps exereplay'.simps split: if-splits \rangle*)

abbreviation *exereplay'' thy vs ns Hs P \equiv 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 \equiv*

(if $\forall x \in \text{set (hyps P)}$. exeterm-ok thy x \wedge typ-of x = Some propT then

exereplay'' thy \square (fst ' (fv-Proof P \cup 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 \square (fst ' fv-Proof P) \square P*

definition *exe-check-proof e P res \equiv*

exe-wf-theory' e \wedge exereplay e P = Some res

lemma *exe-check-proof-iff-check-proof:*

exe-check-proof e P res \longleftrightarrow 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  $\implies$  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  $\implies$  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.