

# Isabelle’s Metalogic: Formalization and Proof Checker

Tobias Nipkow and Simon Roßkopf

May 14, 2024

## Abstract

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

The formalization is intentionally kept close to the Isabelle implementation (for example using de 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

<b>1</b>	<b>Core Inference system</b>	<b>2</b>
<b>2</b>	<b>Preliminaries</b>	<b>8</b>
<b>3</b>	<b>Terms</b>	<b>14</b>
<b>4</b>	<b>Sorts</b>	<b>36</b>
<b>5</b>	<b>Wellformed Signature and Theory</b>	<b>40</b>
<b>6</b>	<b>More on Substitutions</b>	<b>43</b>
<b>7</b>	<b>Names</b>	<b>50</b>
<b>8</b>	<b>Beta Normalization</b>	<b>53</b>
<b>9</b>	<b>Eta Normalization</b>	<b>58</b>
<b>10</b>	<b>Logic</b>	<b>61</b>
<b>11</b>	<b>Derived rules on equality and normalization</b>	<b>76</b>

<b>12 Proof Terms and proof checker</b>	<b>87</b>
<b>13 Executable Sorts</b>	<b>92</b>
<b>14 Executable Instance Relations</b>	<b>97</b>
<b>15 Executable Signature and Theory</b>	<b>105</b>
<b>16 Code Generation</b>	<b>113</b>

## 1 Core Inference system

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

```
theory Core
  imports Main
begin
```

Basic types

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

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

```
type-synonym sort = class set
abbreviation full-sort ≡ { }::sort
```

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

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

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

```
abbreviation mk-fun-typ S T ≡ Ty STR "fun" [S, T]
notation mk-fun-typ (infixr  $\rightarrow$  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** *tinstT*  $T1 \ T2 \equiv \exists \varrho. \textit{tsubstT } T2 \ \varrho = T1$

**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*  $\Sigma$  c  $\equiv$  *class-ex* (subclass (osig  $\Sigma$ )) c

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

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

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

Inference system

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

```

    => has-typ a τ => wf-term (sig Θ) a
    => Θ, Γ ⊢ subst-bv a B
| implies-intro: wf-theory Θ => Θ, Γ ⊢ B => wf-term (sig Θ) A => has-typ A
propT
    => Θ, Γ - {A} ⊢ A ⊢ B
| implies-elim: Θ, Γ1 ⊢ A ⊢ B => Θ, Γ2 ⊢ A => Θ, Γ1 ∪ Γ2 ⊢ B
| of-class: wf-theory Θ
    => const-type (sig Θ) (const-of-class c) = Some (Core.itselfT aT → propT)
    => wf-type (sig Θ) T
    => has-sort (osig (sig Θ)) T {c}
    => Θ, Γ ⊢ mk-of-class T c

| β-conversion: wf-theory Θ => wt-term (sig Θ) (Abs T t) => wf-term (sig Θ) u
=> has-typ u T
    => Θ, Γ ⊢ mk-eq (Abs T t $ u) (subst-bv u t)
| eta: wf-theory Θ => wf-term (sig Θ) t => has-typ t (τ → τ')
    => Θ, Γ ⊢ mk-eq (Abs τ (t $ Bv 0)) t

```

Ensure no garbage in Θ,Γ

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

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

**end**

## 2 Preliminaries

**theory** *Preliminaries*

**imports** *Complex-Main*

*List-Index.List-Index*

*HOL-Library.AList*

*HOL-Library.Sublist*

*HOL-Eisbach.Eisbach*

*HOL-Library.Simps-Case-Conv*

**begin**

Stuff about options

**fun** *the-default* :: 'a ⇒ 'a option ⇒ 'a **where**

*the-default* a None = a

| *the-default* - (Some b) = b

**abbreviation** *Or* :: 'a option ⇒ 'a option ⇒ 'a option (**infixl** OR 60) **where**

*e1 OR e2* ≡ case e1 of None ⇒ e2 | p ⇒ p

**lemma** *Or-Some*: (e1 OR e2) = Some x ⟷ e1 = Some x ∨ (e1 = None ∧ e2 = Some x)



*<proof>*

**lemma** *Or-None*:  $(e1 \text{ OR } e2) = \text{None} \longleftrightarrow e1 = \text{None} \wedge e2 = \text{None}$   
*<proof>*

**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*:  $\text{lift2-option } f \ x \ y \neq \text{None} \longleftrightarrow (x \neq \text{None} \wedge y \neq \text{None})$   
*<proof>*

**lemma** *lift2-option-None*:  $\text{lift2-option } f \ x \ y = \text{None} \longleftrightarrow (x = \text{None} \vee y = \text{None})$   
*<proof>*

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  $\implies \exists x \in \text{set } xs. f \ x = \text{Some } p$   
*<proof>*

**lemma** *find-None*:  
  *find* f xs = None  $\longleftrightarrow (\forall x \in \text{set } xs. f \ x = \text{None})$   
*<proof>*

**lemma** *find-ListFind*:  $\text{find } f \ l = \text{Option.bind } (\text{List.find } (\lambda x. \text{case } f \ x \ \text{of } \text{None} \Rightarrow \text{False} \mid - \Rightarrow \text{True}) \ l) \ f$   
*<proof>*

**lemma** *List.find P l = Some p  $\implies \exists p \in \text{set } l . P \ p$*   
*<proof>*

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

**fun** *remdups-on* ::  $('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list}$  **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) ←→ ¬(∃ x' ∈ set xs . cmp x x') ∧ *distinct-on cmp* xs

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

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

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

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

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

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

**lemma** *lookup-present-eq-key*: *distinct* (map fst al) ⇒ (k, v) ∈ set al ←→ *lookup*  
(λx. x=k) al = Some v  
⟨proof⟩

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

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

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

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

**lemma** *lookup-None-iff-find-None*: *lookup* P xs = None ←→ List.find (λ(k,-). P  
k) xs = None

*<proof>*

**lemma** *lookup-eq-order-irrelevant:*

**assumes** *distinct (map fst pairs) and distinct (map fst pairs') and set pairs = set pairs'*

**shows** *lookup (λx. x=k) pairs = lookup (λx. x=k) pairs'*

*<proof>*

**lemma** *lookup-Some-append-back:*

*lookup (λx. x=k) insts = Some v ⇒ lookup (λx. x=k) (insts@[k,v']) = Some v*

*<proof>*

**lemma** *lookup-eq-key-not-present: key ∉ set (map fst inst) ⇒ lookup (λx. x = key) inst = None*

*<proof>*

**lemma** *lookup-in-empty[simp]: lookup f [] = None <proof>*

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

**lemma** *lookup-present-eq-key': lookup (λx. x=k) al = Some v ⇒ (k, v) ∈ set al*

*<proof>*

**lemma** *lookup-present-eq-key'': distinct (map fst al) ⇒ lookup (λx. x=k) al = Some v ⇔ (k, v) ∈ set al*

*<proof>*

**lemma** *key-present-imp-eq-lookup-finds-value: k ∈ fst ` set al ⇒ ∃ v . lookup (λx. x=k) al = Some v*

*<proof>*

**lemma** *list-allI: (λx. x ∈ set l ⇒ P x) ⇒ list-all P l*

*<proof>*

**lemma** *map2-sym: (λx y . f x y = f y x) ⇒ map2 f xs ys = map2 f ys xs*

*<proof>*

**lemma** *idem-map2: assumes (λx. f x x = x) shows map2 f l l = l*

*<proof>*

**lemma** *rev-induct2[consumes 1, case-names Nil snoc]:*

**assumes** *length xs = length ys*

**assumes** *P [] []*

**assumes** *(λx xs y ys. length xs = length ys ⇒ P xs ys ⇒ P (xs @ [x]) (ys @ [y]))*

**shows** *P xs ys*

*<proof>*

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

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

**lemma** *length-alist*:  
**assumes**  $\text{distinct } (\text{map } \text{fst } al)$  **and**  $\text{distinct } (\text{map } \text{fst } al')$  **and**  $\text{set } al = \text{set } al'$   
**shows**  $\text{length } al = \text{length } al'$   
 <proof>

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

**lemma** *in-range-if-ex-key*:  $v \in \text{ran } m \iff (\exists k. m \ k = \text{Some } v)$   
 <proof>

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

**lemma** *list-all-clearjunk-cons*:  
 $\text{list-all } P \ (x\#(AList.clearjunk \ l)) \implies \text{list-all } P \ (AList.clearjunk \ (x\#l))$   
 <proof>

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

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

**definition** *diff-list*  $xs \ ys \equiv \text{fold } \text{removeAll } ys \ xs$

**lemma** *diff-list-set[simp]*:  $\text{set } (\text{diff-list } xs \ ys) = \text{set } xs - \text{set } ys$   
 <proof>

**lemma** *diff-list-set-from-Nil[simp]*:  $\text{diff-list } [] \ ys = []$   
 <proof>

**lemma** *diff-list-set-remove-Nil[simp]*:  $\text{diff-list } xs \ [] = xs$   
 <proof>

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

$x \# \text{diff-list } xs \ ys$   
 ⟨proof⟩

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

**lemma** *fold-Option-bind-eq-Some-start-not-None*:

$\text{fold } (\lambda \text{new option} . \text{Option.bind option } (f \ \text{new})) \ \text{list} \ \text{start} = \text{Some res}$   
 $\implies \text{start} \neq \text{None}$   
 ⟨proof⟩

**lemma** *fold-Option-bind-eq-Some-at-point-not-None*:

$\text{fold } (\lambda \text{new option} . \text{Option.bind option } (f \ \text{new})) \ (l1 @ l2) \ \text{start} = \text{Some res}$   
 $\implies \text{fold } (\lambda \text{new option} . \text{Option.bind option } (f \ \text{new})) \ (l1) \ \text{start} \neq \text{None}$   
 ⟨proof⟩

**lemma** *fold-Option-bind-eq-Some-start-not-None'*:

$\text{fold } (\lambda(x,y) \ \text{option} . \text{Option.bind option } (f \ x \ y)) \ \text{list} \ \text{start} = \text{Some res}$   
 $\implies \text{start} \neq \text{None}$   
 ⟨proof⟩

**lemma** *fold-Option-bind-eq-None-start-None*:

$\text{fold } (\lambda(x,y) \ \text{option} . \text{Option.bind option } (f \ x \ y)) \ \text{list} \ \text{None} = \text{None}$   
 ⟨proof⟩

**lemma** *fold-Option-bind-at-some-point-None-eq-None*:

$\text{fold } (\lambda(x,y) \ \text{option} . \text{Option.bind option } (f \ x \ y)) \ l1 \ \text{start} = \text{None} \implies$   
 $\text{fold } (\lambda(x,y) \ \text{option} . \text{Option.bind option } (f \ x \ y)) \ (l1 @ l2) \ \text{start} = \text{None}$   
 ⟨proof⟩

**lemma** *fold-Option-bind-eq-Some-at-each-point-Some*:

$\text{fold } (\lambda(x,y) \ \text{option} . \text{Option.bind option } (f \ x \ y)) \ (l1 @ l2) \ \text{start} = \text{Some res}$   
 $\implies (\exists \ \text{point} . \text{fold } (\lambda(x,y) \ \text{option} . \text{Option.bind option } (f \ x \ y)) \ l1 \ \text{start} = \text{Some}$   
*point*  
 $\wedge \text{fold } (\lambda(x,y) \ \text{option} . \text{Option.bind option } (f \ x \ y)) \ l2 \ (\text{Some point}) = \text{Some res})$   
 ⟨proof⟩

**lemma** *fold-Option-bind-eq-Some-at-each-point-Some'*:

**assumes**  $\text{fold } (\lambda(x,y) \ \text{option} . \text{Option.bind option } (f \ x \ y)) \ (xs @ ys) \ \text{start} = \text{Some}$   
*res*

**obtains** *point* **where**

$\text{fold } (\lambda(x,y) \ \text{option} . \text{Option.bind option } (f \ x \ y)) \ xs \ \text{start} = \text{Some point}$  **and**  
 $\text{fold } (\lambda(x,y) \ \text{option} . \text{Option.bind option } (f \ x \ y)) \ ys \ (\text{Some point}) = \text{Some res}$   
 ⟨proof⟩

**corollary** *fold-Option-bind-eq-Some-at-point-not-None'*:

$\text{fold } (\lambda(x,y) \ \text{option} . \text{Option.bind option } (f \ x \ y)) \ (l1 @ l2) \ \text{start} = \text{Some res}$   
 $\implies \text{fold } (\lambda(x,y) \ \text{option} . \text{Option.bind option } (f \ x \ y)) \ (l1) \ \text{start} \neq \text{None}$

*<proof>*

**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**

$f\ x\ y\ subs = Some\ point$

$fold (\lambda(T, U) subs . Option.bind\ subs\ (f\ T\ U))\ (zip\ (xs)\ (ys))\ (Some\ point) = Some\ subs'$

*<proof>*

**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**

$fold (\lambda(T, U) subs . Option.bind\ subs\ (f\ T\ U))\ (zip\ (xs)\ (ys))\ (Some\ subs) = Some\ point$

$f\ x\ y\ point = Some\ subs'$

*<proof>*

**end**

### 3 Terms

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

**theory** *Term*

**imports** *Main Core Preliminaries*

**begin**

Collecting parts of `typs/terms` and more substitutions

**fun** *tvsT* :: *typ*  $\Rightarrow$  (*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*:  $tvsT (tsubstT \sigma \varrho) = \bigcup \{tvsT (\varrho a s) \mid a s. (a, s) \in tvsT \sigma\}$   
 ⟨proof⟩

**lemma** *tsubstT-cong*:

$(\forall (v, S) \in tvsT \sigma. \varrho1 v = \varrho2 v) \implies tsubstT \sigma \varrho1 = tsubstT \sigma \varrho2$   
 ⟨proof⟩

**lemma** *tsubstT-ith*:  $i < length Ts \implies map (\lambda T . tsubstT T \varrho) Ts ! i = tsubstT (Ts ! i) \varrho$   
 ⟨proof⟩

**lemma** *tsubstT-fun-typ-dist*:  $tsubstT (T \rightarrow T1) \varrho = tsubstT T \varrho \rightarrow tsubstT T1 \varrho$   
 ⟨proof⟩

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

**definition** *tinst* t1 t2  $\equiv \exists \varrho. tsubst t2 \varrho = t1$

**definition** *inst* t1 t2  $\equiv \exists \varrho. subst t2 \varrho = t1$

**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)

<proof>  
**lemma** *finite-Sorts[simp]: finite (Sorts t)*  
 <proof>  
**lemma** *finite-Types[simp]: finite (Types t)*  
 <proof>  
**lemma** *finite-tvsT[simp]: finite (tvsT T)*  
 <proof>  
**lemma** *no-tvsT-imp-tsubsT-unchanged: tvsT T = {}  $\implies$  tsubstT T  $\rho$  = T*  
 <proof>  
**lemma** *finite-fv[simp]: finite (fv t)*  
 <proof>  
**lemma** *finite-tvs[simp]: finite (tvs t)*  
 <proof>

**lemma** *finite-FV: finite S  $\implies$  finite (FV S)*  
 <proof>  
**lemma** *finite-tvs-Set: finite S  $\implies$  finite (tvs-Set S)*  
 <proof>  
**lemma** *finite-tvsT-Set: finite S  $\implies$  finite (tvsT-Set S)*  
 <proof>

**lemma** *no-tvs-imp-tsubst-unchanged: tvs t = {}  $\implies$  tsubst t  $\rho$  = t*  
 <proof>  
**lemma** *no-fv-imp-subst-unchanged: fv t = {}  $\implies$  subst t  $\rho$  = t*  
 <proof>

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> **thm** *less-Suc-eq nth-butlast*

**lemma** *has-typ1-imp-typ-of1: has-typ1 Ts t ty  $\implies$  typ-of1 Ts t = Some ty*  
 <proof>



**lemma** *typ-of1-imp-has-typ1*:  $\text{typ-of1 } Ts \ t = \text{Some } ty \implies \text{has-typ1 } Ts \ t \ ty$   
(proof)

**corollary** *has-typ1-iff-typ-of1[iff]*:  $\text{has-typ1 } Ts \ t \ ty \longleftrightarrow \text{typ-of1 } Ts \ t = \text{Some } ty$   
(proof)

**corollary** *has-typ-iff-typ-of[iff]*:  $\text{has-typ } t \ ty \longleftrightarrow \text{typ-of } t = \text{Some } ty$   
(proof)

**corollary** *typ-of-imp-has-typ*:  $\text{typ-of } t = \text{Some } ty \implies \text{has-typ } t \ ty$   
(proof)

**lemma** *typ-of1-weaken-Ts*:  $\text{typ-of1 } Ts \ t = \text{Some } ty \implies \text{typ-of1 } (Ts@[T]) \ t = \text{Some } ty$   
(proof)

**lemma** *typ-of1-weaken*:  
assumes  $\text{typ-of1 } Ts \ t = \text{Some } T$   
shows  $\text{typ-of1 } (Ts@[Ts']) \ t = \text{Some } T$   
(proof)

**lemma** *has-typ1-tsubst*:  
 $\text{has-typ1 } Ts \ t \ T \implies \text{has-typ1 } (\text{map } (\lambda T. \text{tsubstT } T \ \varrho) \ Ts) \ (\text{tsubst } t \ \varrho) \ (\text{tsubstT } T \ \varrho)$   
(proof)

**corollary** *has-typ1-unique*:  
assumes  $\text{has-typ1 } \tau s \ t \ \tau 1$  and  $\text{has-typ1 } \tau s \ t \ \tau 2$  shows  $\tau 1 = \tau 2$   
(proof)

**hide-fact** *typ-of-def*

**lemma** *typ-of-def*:  $\text{typ-of } t \equiv \text{typ-of1 } [] \ t$   
(proof)

Loose bound variables

**fun** *loose-bvar* ::  $\text{term} \Rightarrow \text{nat} \Rightarrow \text{bool}$  **where**  
   $\text{loose-bvar } (Bv \ i) \ k \longleftrightarrow i \geq k$   
   $|\ \text{loose-bvar } (t \ \$ \ u) \ k \longleftrightarrow \text{loose-bvar } t \ k \vee \text{loose-bvar } u \ k$   
   $|\ \text{loose-bvar } (Abs \ - \ t) \ k = \text{loose-bvar } t \ (k+1)$   
   $|\ \text{loose-bvar } \_ \_ = \text{False}$

**fun** *loose-bvar1* ::  $\text{term} \Rightarrow \text{nat} \Rightarrow \text{bool}$  **where**  
   $\text{loose-bvar1 } (Bv \ i) \ k \longleftrightarrow i = k$   
   $|\ \text{loose-bvar1 } (t \ \$ \ u) \ k \longleftrightarrow \text{loose-bvar1 } t \ k \vee \text{loose-bvar1 } u \ k$   
   $|\ \text{loose-bvar1 } (Abs \ - \ t) \ k = \text{loose-bvar1 } t \ (k+1)$   
   $|\ \text{loose-bvar1 } \_ \_ = \text{False}$

**lemma** *loose-bvar1-imp-loose-bvar*:  $loose-bvar1\ t\ n \implies loose-bvar\ t\ n$

*<proof>*

**lemma** *not-loose-bvar-imp-not-loose-bvar1*:  $\neg\ loose-bvar\ t\ n \implies \neg\ loose-bvar1\ t\ n$

*<proof>*

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

*<proof>*

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

**abbreviation** *is-closed*  $t \equiv \neg\ is-open\ t$

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

**lemma** *loose-bvar-Suc*:  $loose-bvar\ t\ (Suc\ k) \implies loose-bvar\ t\ k$

*<proof>*

**lemma** *loose-bvar-leq*:  $k \geq p \implies loose-bvar\ t\ k \implies loose-bvar\ t\ p$

*<proof>*

**lemma** *has-typ1-imp-no-loose-bvar*:  $has-typ1\ Ts\ t\ ty \implies \neg\ loose-bvar\ t\ (length\ Ts)$

*<proof>*

**corollary** *has-typ-imp-closed*:  $has-typ\ t\ ty \implies \neg\ is-open\ t$

*<proof>*

**corollary** *typ-of-imp-closed*:  $typ-of\ t = Some\ ty \implies \neg\ is-open\ t$

*<proof>*

Subterms

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

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

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

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

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

*<proof>*

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

*<proof>*

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

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

**lemma**  $bind\text{-fv2}\text{-unchanged}$ :  
 $\neg loose\text{-bvar}\ tm\ lev \implies bind\text{-fv2}\ v\ lev\ tm = tm \implies v \notin fv\ tm$   
*<proof>*

**lemma**  $bind\text{-fv2}\text{-unchanged}'$ :  
 $\neg loose\text{-bvar}\ tm\ lev \implies bind\text{-fv2}\ v\ lev\ tm = tm \implies \neg occs\ (case\text{-prod}\ Fv\ v)\ tm$   
*<proof>*

**lemma**  $bind\text{-fv2}\text{-changed}$ :  
 $bind\text{-fv2}\ v\ lev\ tm \neq tm \implies v \in fv\ tm$   
*<proof>*

**lemma**  $bind\text{-fv2}\text{-changed}'$ :  
 $bind\text{-fv2}\ v\ lev\ tm \neq tm \implies occs\ (case\text{-prod}\ Fv\ v)\ tm$   
*<proof>*

**corollary**  $bind\text{-fv}\text{-changed}$ :  $bind\text{-fv}\ v\ tm \neq tm \implies v \in fv\ tm$   
*<proof>*

**corollary**  $bind\text{-fv}\text{-changed}'$ :  $bind\text{-fv}\ v\ tm \neq tm \implies occs\ (case\text{-prod}\ Fv\ v)\ tm$   
*<proof>*

**corollary**  $bind\text{-fv}\text{-unchanged}$ :  $(x, \tau) \notin fv\ t \implies bind\text{-fv}\ (x, \tau)\ t = t$   
*<proof>*

**inductive-cases**  $has\text{-typ1}\text{-app}\text{-elim}$ :  $has\text{-typ1}\ Ts\ (t\ \$\ u)\ R$

**lemma**  $has\text{-typ1}\text{-arg}\text{-typ}$ :  $has\text{-typ1}\ Ts\ (t\ \$\ u)\ R \implies has\text{-typ1}\ Ts\ u\ U \implies has\text{-typ1}\ Ts\ t\ (U \rightarrow R)$   
*<proof>*

**lemma**  $has\text{-typ1}\text{-fun}\text{-typ}$ :  $has\text{-typ1}\ Ts\ (t\ \$\ u)\ R \implies has\text{-typ1}\ Ts\ t\ (U \rightarrow R) \implies has\text{-typ1}\ Ts\ u\ U$   
*<proof>*

**lemma**  $typ\text{-of1}\text{-arg}\text{-typ}$ :  
 $typ\text{-of1}\ Ts\ (t\ \$\ u) = Some\ R \implies typ\text{-of1}\ Ts\ u = Some\ U \implies typ\text{-of1}\ Ts\ t = Some\ (U \rightarrow R)$   
*<proof>*

**corollary**  $typ\text{-of}\text{-arg}$ :  $typ\text{-of}\ (t\$u) = Some\ R \implies typ\text{-of}\ u = Some\ T \implies typ\text{-of}\ t = Some\ (T \rightarrow R)$   
*<proof>*

**lemma**  $typ\text{-of1}\text{-fun}\text{-typ}$ :  
 $typ\text{-of1}\ Ts\ (t\ \$\ u) = Some\ R \implies typ\text{-of1}\ Ts\ t = Some\ (U \rightarrow R) \implies typ\text{-of1}\ Ts\ u = Some\ U$   
*<proof>*

**corollary** *typ-of-fun*:  $\text{typ-of } (t\$u) = \text{Some } R \implies \text{typ-of } t = \text{Some } (U \rightarrow R) \implies \text{typ-of } u = \text{Some } U$   
 ⟨proof⟩

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

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

**lemma** *typ-of-Abs-bind-fv*:  
 assumes  $\text{typ-of } A = \text{Some } ty$   
 shows  $\text{typ-of } (\text{Abs } bT \ (\text{bind-fv } (v, bT) \ A)) = \text{Some } (bT \rightarrow ty)$   
 ⟨proof⟩

**corollary** *typ-of-Abs-fv*:  
 assumes  $\text{typ-of } A = \text{Some } ty$   
 shows  $\text{typ-of } (\text{Abs-fv } v \ bT \ A) = \text{Some } (bT \rightarrow ty)$   
 ⟨proof⟩

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

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

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

**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**  
 |  $\text{decr } \text{lev } (\text{Bv } i) = (\text{if } i \geq \text{lev} \ \text{then } \text{Bv } (i - 1) \ \text{else } \text{Bv } i)$   
 |  $\text{decr } \text{lev } (\text{Abs } T \ t) = \text{Abs } T \ (\text{decr } (\text{lev} + 1) \ t)$   
 |  $\text{decr } \text{lev } (t \$ u) = (\text{decr } \text{lev } t \$ \text{decr } \text{lev } u)$   
 |  $\text{decr } \text{lev } t = t$

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

**lemma** *loose-bvar-incr-bvar*:  $\text{loose-bvar } t \text{ lev} \iff \text{loose-bvar } (\text{incr-bv } \text{inc } \text{lev } t)$   
(*lev+inc*)  
(*proof*)

**lemma** *no-loose-bvar-no-incr[simp]*:  $\neg \text{loose-bvar } t \text{ lev} \implies \text{incr-bv } \text{inc } \text{lev } t = t$   
(*proof*)

**lemma** *is-close-no-incr-boundvars[simp]*:  $\text{is-closed } t \implies \text{incr-boundvars } \text{inc } t = t$   
(*proof*)

**lemma** *fv-incr-bv [simp]*:  $\text{fv } (\text{incr-bv } \text{inc } \text{lev } t) = \text{fv } t$   
(*proof*)

**lemma** *fv-incr-boundvars [simp]*:  $\text{fv } (\text{incr-boundvars } \text{inc } t) = \text{fv } t$   
(*proof*)

**lemma** *loose-bvar-decr*:  $\neg \text{loose-bvar } t \text{ k} \implies \neg \text{loose-bvar } (\text{decr } k \text{ t}) \text{ k}$   
(*proof*)

**lemma** *loose-bvar-decr-unchanged[simp]*:  $\neg \text{loose-bvar } t \text{ k} \implies \text{decr } k \text{ t} = t$   
(*proof*)

**lemma** *is-closed-decr-unchanged[simp]*:  $\text{is-closed } t \implies \text{decr } 0 \text{ t} = t$   
(*proof*)

**fun** *subst-bv1* ::  $\text{term} \Rightarrow \text{nat} \Rightarrow \text{term} \Rightarrow \text{term}$  **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*:  $\text{incr-bv } m \text{ k } (\text{incr-bv } n \text{ k } s) = \text{incr-bv } (m+n) \text{ k } s$   
(*proof*)

**lemma** *substn-subst-n* :  $\text{subst-bv1 } t \text{ n } s = \text{subst-bv2 } t \text{ n } (\text{incr-bv } n \text{ 0 } s)$   
(*proof*)

**theorem** *substn-subst-0*:  $\text{subst-bv1 } t \text{ 0 } s = \text{subst-bv2 } t \text{ 0 } s$   
(*proof*)

**corollary** *substn-subst-0'*:  $\text{subst-bv } s \text{ t} = \text{subst-bv2 } t \text{ 0 } s$   
(*proof*)

**lemma** *subst-bv2-eq [simp]*:  $\text{subst-bv2 } (\text{Bv } k) \text{ k } u = u$   
(*proof*)

**lemma** *subst-bv2-gt* [simp]:  $i < j \implies \text{subst-bv2 } (Bv\ j)\ i\ u = Bv\ (j - 1)$   
 ⟨proof⟩

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

**lemma** *lift-lift*:  
 $i < k + 1 \implies \text{lift } (\text{lift } t\ i)\ (Suc\ k) = \text{lift } (\text{lift } t\ k)\ i$   
 ⟨proof⟩

**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⟩

**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⟩

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

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

**hide-fact** (open) *subst-bv-def*

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

**fun** *subst-bvs1* ::  $\text{term} \Rightarrow \text{nat} \Rightarrow \text{term list} \Rightarrow \text{term}$  **where**

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

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

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

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

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

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

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

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

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

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

**lemma** *loose-bvar-subst-bv1*:  
 $\neg \text{loose-bvar } (\text{subst-bv1 } t \text{ lev } u) \text{ lev} \implies \neg \text{loose-bvar } t \text{ } (\text{Suc } \text{lev})$   
 ⟨proof⟩

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

**lemma** *subst-bv1-bind-fv2*:  
 assumes  $\neg \text{loose-bvar } t \text{ lev}$   
 shows  $\text{subst-bv1 } (\text{bind-fv2 } (v, T) \text{ lev } t) \text{ lev } (Fv \ v \ T) = t$   
 ⟨proof⟩

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

**fun** *betapply* ::  $\text{term} \Rightarrow \text{term} \Rightarrow \text{term}$  (**infixl** · 52) **where**  
 $\text{betapply } (\text{Abs } - \ t) \ u = \text{subst-bv } u \ t$   
 |  $\text{betapply } t \ u = t \ \$ \ u$

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

**lemma** *typ-of1-imp-no-loose-bvar*:  $\text{typ-of1 } Ts \ t = \text{Some } ty \implies \neg \text{loose-bvar } t$   
*(length Ts)*

*<proof>*

**lemma** *typ-of1-subst-bv*:

**assumes**  $\text{typ-of1 } (Ts@[uty]) \ f = \text{Some } fty$

**and**  $\text{typ-of } u = \text{Some } uty$

**shows**  $\text{typ-of1 } Ts \ (\text{subst-bv1 } f \ (\text{length } Ts) \ u) = \text{Some } fty$

*<proof>*

**lemma** *typ-of1-split-App*:

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

*<proof>*

**corollary** *typ-of1-split-App-obtains*:

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

**obtains** *uty* **where**  $\text{typ-of1 } Ts \ t = \text{Some } (uty \rightarrow ty) \ \text{typ-of1 } Ts \ u = \text{Some } uty$

*<proof>*

**lemma** *typ-of1-incr-bv*:

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

**and**  $lev \leq \text{length } Ts$

**shows**  $\text{typ-of1 } (\text{take } lev \ Ts \ @ \ Ts' \ @ \ \text{drop } lev \ Ts) \ (\text{incr-bv } (\text{length } Ts') \ lev \ t) =$   
 $\text{Some } ty$

*<proof>*

**corollary** *typ-of1-incr-bv-lev0*:

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

**shows**  $\text{typ-of1 } (Ts' \ @ \ Ts) \ (\text{incr-bv } (\text{length } Ts') \ 0 \ t) = \text{Some } ty$

*<proof>*

**lemma** *typ-of1-subst-bv-gen*:

**assumes**  $\text{typ-of1 } (Ts'@[uty]@Ts) \ t = \text{Some } tty$  **and**  $\text{typ-of1 } Ts \ u = \text{Some } uty$

**shows**  $\text{typ-of1 } (Ts' \ @ \ Ts) \ (\text{subst-bv1 } t \ (\text{length } Ts') \ u) = \text{Some } tty$

*<proof>*

**lemma** *typ-of1-subst-bv-gen-depre*:

**assumes**  $\text{typ-of1 } (Ts'@Ts) \ f = \text{Some } (fty)$

**and**  $\text{typ-of1 } (Ts) \ u = \text{Some } uty$

**and**  $\text{last } Ts' = uty$  **and**  $Ts' \neq []$

**shows**  $\text{typ-of1 } (\text{butlast } Ts' \ @ \ Ts) \ (\text{subst-bv1 } f \ (\text{length } Ts' - 1) \ u) = \text{Some } fty$

*<proof>*

**corollary** *typ-of1-subst-bv-gen'*:

**assumes**  $\text{typ-of1 } (uty\#Ts) \ t = \text{Some } tty$

**and**  $\text{typ-of1 } Ts \ u = \text{Some } uty$

**shows**  $\text{typ-of1 } Ts \ (\text{subst-bv1 } t \ 0 \ u) = \text{Some } tty$



*<proof>*

**lemma** *typ-of-betaapply*:

**assumes** *typ-of1 Ts (Abs uty t) = Some (uty → tty)*

**assumes** *typ-of1 Ts u = Some uty*

**shows** *typ-of1 Ts ((Abs uty t) · u) = Some tty*

*<proof>*

**lemma** *no-Bv-Type-param-irrelevant-typ-of*:

*¬exists-subterm (λx . case x of Bv - ⇒ True | - ⇒ False) t*

*⇒ typ-of1 Ts t = typ-of1 Ts' t*

*<proof>*

**lemma** *typ-of1-drop-extra-bounds*:

*¬loose-bvar t (length Ts)*

*⇒ typ-of1 (Ts@rest) t = typ-of1 Ts t*

*<proof>*

**lemma** *typ-of-betaapply*:

**assumes** *typ-of t = Some (uty → tty) typ-of u = Some uty*

**shows** *typ-of (t · u) = Some tty*

*<proof>*

**fun** *beta-reducible* :: *term ⇒ bool* **where**

*beta-reducible (App (Abs - -) -) = True*

| *beta-reducible (Abs - t) = beta-reducible t*

| *beta-reducible (App t u) = (beta-reducible t ∨ beta-reducible u)*

| *beta-reducible - = False*

**fun** *eta-reducible* :: *term ⇒ bool* **where**

*eta-reducible (Abs - (t \$ Bv 0)) = (¬ 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*

*<proof>*

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

*<proof>*

**fun** *depth* :: *term ⇒ nat* **where**

*depth (Abs - t) = depth t + 1*

| *depth (t \$ u) = max (depth t) (depth u) + 1*

| *depth t = 0*

**lemma** *depth-decr*: *depth (decr lev t) = depth t*

*<proof>*

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

**lemma** *loose-bvar1-decr'*:  
 $\neg \text{loose-bvar1 } t \text{ (Suc lev)} \implies \neg \text{loose-bvar1 } t \text{ lev} \implies \neg \text{loose-bvar1 (decr lev } t) \text{ lev}$   
 ⟨proof⟩

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

**lemma** *eta-reducible-Abs2*:  
**assumes**  $\neg (\exists f. t=f \text{ \$ } Bv \ 0) \neg \text{eta-reducible (Abs } T \ t)$   
**shows**  $\neg \text{eta-reducible } t$   
 ⟨proof⟩

**lemma** *eta-reducible-Abs*:  $\neg \text{eta-reducible (Abs } T \ t) \implies \neg \text{eta-reducible } t$   
 ⟨proof⟩

**lemma** *loose-bvar1-decr''*:  $\text{loose-bvar1 } t \text{ lev} \implies lev < lev' \implies \text{loose-bvar1 (decr lev' } t) \text{ lev}$   
 ⟨proof⟩

**lemma** *loose-bvar1-decr'''*:  $\text{loose-bvar1 } t \text{ (Suc lev)} \implies lev' \leq lev \implies \text{loose-bvar1 (decr lev' } t) \text{ lev}$   
 ⟨proof⟩

**lemma** *loose-bvar1-decr''''*:  $\neg \text{loose-bvar1 } t \text{ lev'} \implies lev' \leq lev \implies \neg \text{loose-bvar1 } t \text{ (Suc lev)}$   
 $\implies \neg \text{loose-bvar1 (decr lev' } t) \text{ lev}$   
 ⟨proof⟩

**lemma** *not-eta-reducible-decr*:  
 $\neg \text{eta-reducible } t \implies \neg \text{loose-bvar1 } t \text{ lev} \implies \neg \text{eta-reducible (decr lev } t)$   
 ⟨proof⟩

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

**lemma** *eta-norm-reduces-depth*:  $\text{eta-norm-dom } t \implies \text{depth (eta-norm } t) \leq \text{depth } t$   
 ⟨proof⟩

**termination** *eta-norm*

*<proof>*

**lemma** *loose-bvar1-eta-norm*:  $\text{loose-bvar1 } t \text{ lev} \implies \text{loose-bvar1 } (\text{eta-norm } t) \text{ lev}$

*<proof>*

**lemma** *loose-bvar1-eta-norm'*:  $\neg \text{loose-bvar1 } t \text{ lev} \implies \neg \text{loose-bvar1 } (\text{eta-norm } t) \text{ lev}$

*<proof>*

**lemma** *not-eta-reducible-eta-norm*:  $\neg \text{eta-reducible } (\text{eta-norm } t)$

*<proof>*

**lemma** *not-eta-reducible-imp-eta-norm-no-change*:  $\neg \text{eta-reducible } t \implies \text{eta-norm } t = t$

*<proof>*

**lemma** *eta-norm-collapse*:  $\text{eta-norm } (\text{eta-norm } t) = \text{eta-norm } t$

*<proof>*

**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>*

**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>*

**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>*

**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>*

**corollary** *typ-of-eta-norm*:  $\text{typ-of } t = \text{Some } ty \implies \text{typ-of } (\text{eta-norm } t) = \text{Some } ty$

*<proof>*

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

*<proof>*

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

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

*<proof>*

**lemma** *typ-of-beta-redex-arg*: *typ-of* (Abs T s \$ t) ≠ None ⇒ *typ-of* t = Some T  
*<proof>*

**lemma** [*partial-function-mono*]: *option.mono-body*  
(λbeta-norm. map-option (Abs T) (beta-norm t))  
*<proof>*

**lemma** [*partial-function-mono*]: *option.mono-body*  
(λbeta-norm.  
 case beta-norm x of None ⇒ None  
 | Some (Ct list typ) ⇒  
 map-option ((\$) (Ct list typ)) (beta-norm u)  
 | Some (Fv p typ) ⇒  
 map-option ((\$) (Fv p typ)) (beta-norm u)  
 | Some (Bv n) ⇒  
 map-option ((\$) (Bv n)) (beta-norm u)  
 | Some (Abs T body) ⇒  
 beta-norm (subst-bv u body)  
 | Some (term1 \$ term2) ⇒  
 map-option ((\$) (term1 \$ term2)) (beta-norm u))  
*<proof>*

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

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

**lemma** *not-beta-reducible-imp-beta-norm-unchanged*: ¬ beta-reducible t ⇒ beta-norm  
t = Some t  
*<proof>*

**lemma** *not-beta-reducible-decr*: ¬ beta-reducible t ⇒ ¬ beta-reducible (decr n t)  
*<proof>*

**lemma** ¬ beta-reducible t ⇒ eta-norm t = t' ⇒ ¬ beta-reducible t'  
*<proof>*

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

| *is-variable* - = *False*

**lemma** *fv-occs*:  $(x, \tau) \in \text{fv } t \implies \text{occs } (Fv \ x \ \tau) \ t$   
⟨*proof*⟩

**lemma** *fv-iff-occs*:  $(x, \tau) \in \text{fv } t \iff \text{occs } (Fv \ x \ \tau) \ t$   
⟨*proof*⟩

**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* ⟨*proof*⟩

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

**definition** *strip-comb* :: *term*  $\Rightarrow$  *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*  
⟨*proof*⟩

**corollary** *fst* (*strip-comb* *t*) = *head-of* *t*  
⟨*proof*⟩

**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*)  
⟨*proof*⟩

**corollary** *not-is-app-imp-strip-com-unchanged*:  $\neg$  *is-app* *t*  $\Longrightarrow$  *strip-comb* *t* = (*t*,[])  
⟨*proof*⟩

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

**fun** *add-size-term* :: *term*  $\Rightarrow$  *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*  
⟨*proof*⟩

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

**lemma** *map-aterms id t = t*  
⟨*proof*⟩

**definition** *map-type-tvar f = map-atyps* ( $\lambda x . \text{case } x \text{ of } Tv \text{ iname } s \Rightarrow f \text{ iname } s$   
| *T* ⇒ *T*)

**lemma** *map-types-id[simp]: map-types id t = t*  
⟨*proof*⟩

**lemma** *map-types-id'[simp]: map-types* ( $\lambda a . a$ ) *t = t*  
⟨*proof*⟩

**fun** *fold-atyps* :: (*typ* ⇒ '*a* ⇒ '*a*) ⇒ *typ* ⇒ '*a* ⇒ '*a* **where**  
*fold-atyps* *f* (*Ty* - *Ts*) *s* = *fold* (*fold-atyps* *f*) *Ts* *s*  
| *fold-atyps* *f* *T* *s* = *f* *T* *s*

**definition** *fold-atyps-sorts f =*  
*fold-atyps* ( $\lambda x . \text{case } x \text{ of } Tv \text{ vn } S \Rightarrow f (Tv \text{ vn } S) S$ )

**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* ( $\lambda x . f$ )

**fun** *replace-types* :: *term* ⇒ *typ list* ⇒ *term* × *typ list* **where**  
*replace-types* (*Ct* *c* -) (*T* # *Ts*) = (*Ct* *c* *T*, *Ts*)

$| \text{replace-types } (Fv \ xi \ -) \ (T \ \# \ Ts) = (Fv \ xi \ T, \ Ts)$   
 $| \text{replace-types } (Bv \ i) \ Ts = (Bv \ i, \ Ts)$   
 $| \text{replace-types } (Abs \ - \ b) \ (T \ \# \ Ts) =$   
 $\quad (\text{let } (b', \ Ts') = \text{replace-types } b \ Ts$   
 $\quad \text{in } (Abs \ T \ b', \ Ts'))$   
 $| \text{replace-types } (t \ \$ \ u) \ Ts =$   
 $\quad (\text{let}$   
 $\quad \quad (t', \ Ts') = \text{replace-types } t \ Ts \ \text{in}$   
 $\quad \quad (\text{let } (u', \ Ts'') = \text{replace-types } u \ Ts$   
 $\quad \quad \text{in } (t' \ \$ \ u', \ Ts''))$

**definition**  $\text{add-tvar-names}T' = \text{fold-atyps } (\lambda x \ l . \text{case } x \ \text{of } Tv \ xi \ - \Rightarrow \text{List.insert } xi \ l \ | \ - \Rightarrow l)$

**definition**  $\text{add-tvar-names}' = \text{fold-types } \text{add-tvar-names}T'$

**definition**  $\text{add-tvars}T' = \text{fold-atyps } (\lambda x \ l . \text{case } x \ \text{of } Tv \ idn \ s \Rightarrow \text{List.insert } (idn, s) \ l \ | \ - \Rightarrow l)$

**definition**  $\text{add-tvars}' = \text{fold-types } \text{add-tvars}T'$

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

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

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

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

**definition**  $\text{add-tvar-names}T = \text{fold-atyps } (\lambda x . \text{case } x \ \text{of } Tv \ xi \ - \Rightarrow \text{insert } xi \ | \ - \Rightarrow id)$

**definition**  $\text{add-tvar-names} = \text{fold-types } \text{add-tvar-names}T$

**definition**  $\text{add-tvars}T = \text{fold-atyps } (\lambda x . \text{case } x \ \text{of } Tv \ idn \ s \Rightarrow \text{insert } (idn, s) \ | \ - \Rightarrow id)$

**definition**  $\text{add-tvars} = \text{fold-types } \text{add-tvars}T$

**definition**  $\text{add-var-names} = \text{fold-aterms } (\lambda x . \text{case } x \ \text{of } Fv \ xi \ - \Rightarrow \text{insert } xi \ | \ - \Rightarrow id)$

**definition**  $\text{add-vars} = \text{fold-aterms } (\lambda x . \text{case } x \ \text{of } Fv \ idn \ s \Rightarrow \text{insert } (idn, s) \ | \ - \Rightarrow id)$

**definition**  $\text{add-const-names} = \text{fold-aterms } (\lambda x . \text{case } x \ \text{of } Ct \ c \ - \Rightarrow \text{insert } c \ | \ - \Rightarrow id)$

**definition**  $\text{add-consts} = \text{fold-aterms } (\lambda x . \text{case } x \ \text{of } Ct \ n \ s \Rightarrow \text{insert } (n, s) \ | \ - \Rightarrow id)$

**lemma**  $\text{add-tvars}T' \text{-tvs}T \text{-pre}[\text{simp}]: \text{set } (\text{add-tvars}T' \ T \ \text{acc}) = \text{set } \text{acc} \cup \text{tvs}T \ T$   
*(proof)*



**lemma** *add-tvars'-tvs-pre[simp]*:  $set (add-tvars' t acc) = set acc \cup tvs t$   
 ⟨proof⟩

**lemma** *add-tvarsT T acc* =  $acc \cup tvsT T$   
 ⟨proof⟩

**lemma** *add-vars'-fv-pre*:  $set (add-vars' t acc) = set acc \cup fv t$   
 ⟨proof⟩

**corollary** *add-vars'-fv*:  $set (add-vars' t []) = fv t$   
 ⟨proof⟩

**fun** *strip-all-body* ::  $term \Rightarrow term$  **where**  
*strip-all-body* (Ct all S \$ Abs T t) = (if all= STR "Pure.all"  $\wedge S=(T \rightarrow prop T) \rightarrow prop T$   
 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"  $\wedge S=(T \rightarrow prop T) \rightarrow prop T$   
 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"  $\wedge S=(T \rightarrow prop T) \rightarrow prop T$   
 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"  $\wedge S=(T \rightarrow prop T) \rightarrow prop T$   
 then Some T else None)  
 | *strip-all-single-var* t = None

**fun** *strip-all-multiple-body* ::  $nat \Rightarrow term \Rightarrow term$  **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"  $\wedge$   
 $S=(T \rightarrow prop T) \rightarrow prop T$   
 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* ::  $nat \Rightarrow term \Rightarrow typ list$  **where**  
*strip-all-multiple-vars* 0 - = []  
 | *strip-all-multiple-vars* (Suc n) (Ct all S \$ Abs T t) = (if all= STR "Pure.all"  $\wedge$   
 $S=(T \rightarrow prop T) \rightarrow prop T$

then  $T \# \text{strip-all-multiple-vars } n \ t \text{ else } []$   
 $| \text{strip-all-multiple-vars } - \ t = []$

**lemma** *strip-all-vars-strip-all-multiple-vars*:

$n \geq \text{length } (\text{strip-all-vars } t) \implies \text{strip-all-multiple-vars } n \ t = \text{strip-all-vars } t$   
 $\langle \text{proof} \rangle$

**lemma**  $n \geq \text{length } (\text{strip-all-vars } t) \implies \text{strip-all-multiple-body } n \ t = \text{strip-all-body } t$   
 $\langle \text{proof} \rangle$

**lemma** *length-strip-all-multiple-vars*:  $\text{length } (\text{strip-all-multiple-vars } n \ t) \leq n$   
 $\langle \text{proof} \rangle$

**lemma** *prefix-strip-all-multiple-vars*:  $\text{prefix } (\text{strip-all-multiple-vars } n \ t) (\text{strip-all-vars } t)$   
 $\langle \text{proof} \rangle$

**definition** *mk-all-list*  $l \ t = \text{fold } (\lambda(n,T) \ \text{acc} . \text{mk-all } n \ T \ \text{acc}) \ l \ t$

**lemma** *mk-all-list-empty[simp]*:  $\text{mk-all-list } [] \ t = t \ \langle \text{proof} \rangle$

**fun** *is-all* :: *term*  $\implies$  *bool* **where**

$\text{is-all } (Ct \ \text{all } S \ \$ \ \text{Abs } T \ t) = (\text{all} = \text{STR } "Pure.all" \wedge S = (T \rightarrow \text{prop } T) \rightarrow \text{prop } T)$   
 $| \text{is-all } - = \text{False}$

**lemma** *strip-all-single-var-is-all*:  $\text{strip-all-single-var } t \neq \text{None} \iff \text{is-all } t$   
 $\langle \text{proof} \rangle$

**lemma** *is-all*  $t \implies \text{hd } (\text{strip-all-vars } t) = \text{the } (\text{strip-all-single-var } t)$   
 $\langle \text{proof} \rangle$

**lemma** *strip-all-body-single-simp[simp]*:  $\text{strip-all-body } (\text{strip-all-single-body } t) = \text{strip-all-body } t$   
 $\langle \text{proof} \rangle$

**lemma** *strip-all-body-single-simp'[simp]*:  $\text{strip-all-single-body } (\text{strip-all-body } t) = \text{strip-all-body } t$   
 $\langle \text{proof} \rangle$

**lemma** *strip-all-vars-step*:

$\text{strip-all-single-var } t = \text{Some } T \implies T \# \text{strip-all-vars } (\text{strip-all-single-body } t) = \text{strip-all-vars } t$   
 $\langle \text{proof} \rangle$

**lemma** *is-all-iff-strip-all-vars-not-empty*:  $\text{is-all } t \iff \text{strip-all-vars } t \neq []$   
 $\langle \text{proof} \rangle$

**lemma** *strip-all-vars-bind-fv*:

$\text{strip-all-vars } (\text{bind-fv2 } v \ \text{lev } t) = (\text{strip-all-vars } t)$

*<proof>*

**lemma** *strip-all-vars-mk-all[simp]*:  $\text{strip-all-vars } (\text{mk-all } s \text{ ty } t) = \text{ty} \# \text{strip-all-vars } t$   
*<proof>*

**lemma** *strip-all-vars-mk-all-list*:  
 $\neg \text{is-all } t \implies \text{strip-all-vars } (\text{mk-all-list } l \ t) = \text{rev } (\text{map } \text{snd } l)$   
*<proof>*

**lemma** *subst-bv-no-loose-unchanged*:  
**assumes**  $\bigwedge x . x \geq \text{lev} \implies \neg \text{loose-bvar1 } t \ x$   
**assumes** *is-variable*  $v$   
**shows**  $(\text{subst-bv1 } t \ \text{lev } v) = t$   
*<proof>*

**lemma** *bind-fv2-no-occs-unchanged*:  
**assumes**  $\neg \text{occs } (\text{case-prod } Fv \ v) \ t$   
**shows**  $(\text{bind-fv2 } v \ \text{lev } t) = t$   
*<proof>*

**lemma** *bind-fv2-subst-bv1-cancel*:  
**assumes**  $\bigwedge x . x > \text{lev} \implies \neg \text{loose-bvar1 } t \ x$   
**assumes**  $\neg \text{occs } (\text{case-prod } Fv \ v) \ t$   
**shows**  $\text{bind-fv2 } v \ \text{lev } (\text{subst-bv1 } t \ \text{lev } (\text{case-prod } Fv \ v)) = t$   
*<proof>*

**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**  $\text{bind-fv } v \ (\text{subst-bv } (\text{case-prod } Fv \ v) \ t) = t$   
*<proof>*

**lemma** *not-loose-bvar-imp-not-loose-bvar1-all-greater*:  $\neg \text{loose-bvar } t \ \text{lev} \implies x > \text{lev} \implies \neg \text{loose-bvar1 } t \ x$   
*<proof>*

**lemma** *mk-all'-subst-bv-strip-all-single-body-cancel*:  
**assumes**  $\text{strip-all-single-var } t = \text{Some } T$   
**assumes** *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>*

**lemma** *not-is-all-imp-strip-all-body-unchanged*:  $\neg \text{is-all } t \implies \text{strip-all-body } t = t$   
*<proof>*

**lemma** *no-loose-bvar-imp-no-subst-bvs*:  $is-closed\ t \implies subst-bvs\ []\ t = t$   
*<proof>*

**lemma** *is-closed*  $(Abs\ T\ t) \implies \neg\ loose-bvar\ t\ 1$  *<proof>*

**lemma** *bind-fv2-Fv-fv[simp]*:  $fv\ (bind-fv2\ (x,\ \tau)\ lev\ t) = fv\ t - \{(x,\ \tau)\}$   
*<proof>*

**corollary** *mk-all-fv-unchanged*:  $fv\ (mk-all\ x\ \tau\ B) = fv\ B - \{(x,\ \tau)\}$   
*<proof>*

**lemma** *mk-all-list-fv-unchanged*:  $fv\ (mk-all-list\ l\ B) = fv\ B - set\ l$   
*<proof>*

**abbreviation** *forall-intro-vars*  $t\ Hs \equiv mk-all-list$   
 $(diff-list\ (add-vars'\ t\ [])\ (fold\ (add-vars')\ Hs\ []))\ t$

**end**

## 4 Sorts

**theory** *Sorts*  
**imports** *Term*  
**begin**

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

**definition** *sort-les*  $cs\ s1\ s2 = (sort-leq\ cs\ s1\ s2 \wedge \neg\ sort-leq\ cs\ s2\ s1)$

**definition** *sort-equiv*  $cs\ s1\ s2 = (sort-leq\ cs\ s1\ s2 \wedge sort-leq\ cs\ s2\ s1)$

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

**lemmas** *sort-defs* = *sort-leq-def sort-les-def sort-equiv-def sort-ex-def*

**lemma** *sort-ex-class-ex*:  $sort-ex\ cs\ S \equiv \forall\ c \in S.\ class-ex\ cs\ c$   
*<proof>*

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

**lemma** *class-les-irrefl*:  $\neg\ class-les\ cs\ c\ c$   
*<proof>*

**lemma** *class-les-trans*:  $class-les\ cs\ x\ y \implies class-les\ cs\ y\ z \implies class-les\ cs\ x\ z$   
*<proof>*

**lemma** *class-leq-refl[iff]*:  $class-ex\ cs\ c \implies class-leq\ cs\ c\ c$

*<proof>*  
**lemma** *class-leq-trans*:  $class-leq\ cs\ x\ y \implies class-leq\ cs\ y\ z \implies class-leq\ cs\ x\ z$   
*<proof>*  
**lemma** *class-leq-antisym*:  $class-leq\ cs\ c1\ c2 \implies class-leq\ cs\ c2\ c1 \implies c1=c2$   
*<proof>*

**lemma** *sort-leq-refl*[*iff*]:  $sort-ex\ cs\ s \implies sort-leq\ cs\ s\ s$   
*<proof>*  
**lemma** *sort-leq-trans*:  $sort-leq\ cs\ x\ y \implies sort-leq\ cs\ y\ z \implies sort-leq\ cs\ x\ z$   
*<proof>*  
**lemma** *sort-leq-ex*:  $sort-leq\ cs\ s1\ s2 \implies sort-ex\ cs\ s2$   
*<proof>*

**lemma** *sort-leq-minimize*:  
 $sort-leq\ cs\ s1\ s2 \implies \exists s1'. (\forall c1 \in s1'. \exists c2 \in s2. class-leq\ cs\ c1\ c2) \wedge sort-leq\ cs\ s1'\ s2$   
*<proof>*

**lemma** *sort-ex cs s2  $\implies$  s1  $\subseteq$  s2  $\implies$  sort-ex cs s1*  
*<proof>*

**lemma** *superset-imp-sort-leq*:  $sort-ex\ cs\ s2 \implies s1 \supseteq s2 \implies sort-leq\ cs\ s1\ s2$   
*<proof>*  
**lemma** *full-sort-top*:  $sort-ex\ cs\ s \implies sort-leq\ cs\ s\ full-sort$   
*<proof>*

**lemma** *sort-les-trans*:  $sort-les\ cs\ x\ y \implies sort-les\ cs\ y\ z \implies sort-les\ cs\ x\ z$   
*<proof>*

**lemma** *sort-equivI*:  $sort-leq\ cs\ s1\ s2 \implies sort-leq\ cs\ s2\ s1 \implies sort-equiv\ cs\ s1\ s2$   
*<proof>*  
**lemma** *sort-equiv-refl*:  $sort-ex\ cs\ s \implies sort-equiv\ cs\ s\ s$   
*<proof>*  
**lemma** *sort-equiv-trans*:  $sort-equiv\ cs\ x\ y \implies sort-equiv\ cs\ y\ z \implies sort-equiv\ cs\ x\ z$   
*<proof>*  
**lemma** *sort-equiv-sym*:  $sort-equiv\ cs\ x\ y \implies sort-equiv\ cs\ y\ x$   
*<proof>*

**lemma** *normalize-sort-empty*[*simp*]:  $normalize-sort\ cs\ full-sort = full-sort$   
*<proof>*  
**lemma** *normalize-sort-normalize-sort*[*simp*]:  
 $normalize-sort\ cs\ (normalize-sort\ cs\ s) = normalize-sort\ cs\ s$   
*<proof>*

**lemma** *sort-ex-norm-sort*:  $sort-ex\ cs\ s \implies sort-ex\ cs\ (normalize-sort\ cs\ s)$

*<proof>*

**lemma** *normalized-sort-subset: normalize-sort cs s  $\subseteq$  s*  
*<proof>*

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

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

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

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

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

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

**lemma** *sort-ex cs s  $\implies$  sort-leq cs s (normalize-sort cs s)*  
*<proof>*

**lemma** *sort-equiv-normalize-sort:*  
**assumes** *finite s*  
**assumes** *sort-ex cs s*  
**shows** *sort-equiv cs s (normalize-sort cs s)*  
*<proof>*

**lemma** *normalize-sort-eq-imp-sort-eq*:  $\text{sort-ex } cs \ s1 \implies \text{sort-ex } cs \ s2 \implies \text{finite } s1 \implies \text{finite } s2$   
 $\implies \text{normalize-sort } cs \ s1 = \text{normalize-sort } cs \ s2$   
 $\implies \text{sort-equiv } cs \ s1 \ s2$   
 ⟨proof⟩

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

**lemma** *sort-equiv-imp-normalize-sort-eq*:  
**assumes**  $\text{sort-ex } cs \ s1 \ \text{sort-ex } cs \ s2 \ \text{sort-equiv } cs \ s1 \ s2$   
**shows**  $\text{normalize-sort } cs \ s1 = \text{normalize-sort } cs \ s2$   
 ⟨proof⟩

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

**end**

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

**lemma** *osig-subclass-loc*:  $\text{wf-osig } oss \implies \text{wf-subclass-loc } (subclass \ oss)$   
 ⟨proof⟩

**lemma** *wf-osig-imp-wf-subclass-loc*:  $\text{wf-osig } oss \implies \text{wf-subclass-loc } (subclass \ oss)$   
 ⟨proof⟩

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

**end**

Constants for encoding class/sort constraints in term language

**theory** *SortConstants*

**imports** *Sorts*

**begin**

**fun** *dest-type* ::  $\text{term} \Rightarrow \text{typ option}$  **where**  
 $\text{dest-type } (Ct \ nc \ (Ty \ nt \ [ty])) =$   
 (if  $nc = STR \ "Pure.type"$   $\wedge \ nt = STR \ "Pure.type"$  then  $\text{Some } ty$  else  $\text{None}$ )  
 |  $\text{dest-type } t = \text{None}$

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

**consts** *unsuffix* :: *name* ⇒ *name* ⇒ *name option*

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

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

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

**end**

## 5 Wellformed Signature and Theory

**theory** *Theory*

**imports** *Term* *Sorts* *SortConstants*

**begin**

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

**lemma** *typ-ok-sig-imp-wf-type*: *typ-ok-sig*  $\Sigma$  *T* ⇒ *wf-type*  $\Sigma$  *T*  
  ⟨*proof*⟩

**lemma** *wf-type-imp-typ-ok-sig*: *wf-type*  $\Sigma$  *T* ⇒ *typ-ok-sig*  $\Sigma$  *T*  
  ⟨*proof*⟩

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

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



**lemma** *term-ok'-imp-wf-term*:  $\text{term-ok}' \Sigma t \implies \text{wf-term} \Sigma t$   
 ⟨proof⟩  
**lemma** *wf-term-imp-term-ok'*:  $\text{wf-term} \Sigma t \implies \text{term-ok}' \Sigma t$   
 ⟨proof⟩  
**corollary** *wf-term-iff-term-ok'[iff]*:  $\text{wf-term} \Sigma t = \text{term-ok}' \Sigma t$   
 ⟨proof⟩

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

**lemma** *wf-sig* (*Map.empty*, *Map.empty*, *empty-osig*)  
 ⟨proof⟩

**lemma**  
*term-ok-imp-typ-ok-pre*:  
*is-std-sig*  $\Sigma \implies \text{wf-term} \Sigma t \implies \text{list-all} (\text{typ-ok-sig} \Sigma) Ts$   
 $\implies \text{typ-of1} Ts t = \text{Some } ty \implies \text{typ-ok-sig} \Sigma ty$   
 ⟨proof⟩

**lemma** *theory-full-exhaust*:  $(\bigwedge \text{cto } \text{tao } \text{sorts } \text{axioms}.$   
 $\Theta = ((\text{cto}, \text{tao}, \text{sorts}), \text{axioms}) \implies P$   
 $\implies P$   
 ⟨proof⟩

**definition** [*simp*]:  $\text{typ-ok} \Theta T \equiv \text{wf-type} (\text{sig } \Theta) T$   
**definition** [*simp*]:  $\text{term-ok} \Theta t \equiv \text{wt-term} (\text{sig } \Theta) t$

**corollary** *typ-of-subst-bv-no-change*:  $\text{typ-of } t \neq \text{None} \implies \text{subst-bv } u t = t$   
 ⟨proof⟩

**corollary** *term-ok-subst-bv-no-change*:  $\text{term-ok} \Theta t \implies \text{subst-bv } u t = t$   
 ⟨proof⟩

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

**bundle** *eq-axs-simp*

**begin**

**declare** *eq-axs-def*[*simp*]

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

**end**

**lemma** *typ-of-eq-ax*:  $\text{typ-of} (\text{eq-reflexive-ax}) = \text{Some propT}$   
 $\text{typ-of} (\text{eq-symmetric-ax}) = \text{Some propT}$   
 $\text{typ-of} (\text{eq-transitive-ax}) = \text{Some propT}$   
 $\text{typ-of} (\text{eq-intr-ax}) = \text{Some propT}$   
 $\text{typ-of} (\text{eq-elim-ax}) = \text{Some propT}$   
 $\text{typ-of} (\text{eq-combination-ax}) = \text{Some propT}$   
 $\text{typ-of} (\text{eq-abstract-rule-ax}) = \text{Some propT}$

*<proof>*

**lemma** *term-ok-eq-ax:*

**assumes** *is-std-sig* (sig  $\Theta$ )

**shows** *term-ok*  $\Theta$  (*eq-reflexive-ax*)

*term-ok*  $\Theta$  (*eq-symmetric-ax*)

*term-ok*  $\Theta$  (*eq-transitive-ax*)

*term-ok*  $\Theta$  (*eq-intr-ax*)

*term-ok*  $\Theta$  (*eq-elim-ax*)

*term-ok*  $\Theta$  (*eq-combination-ax*)

*term-ok*  $\Theta$  (*eq-abstract-rule-ax*)

*<proof>*

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

*<proof>*

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

*<proof>*

**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

*<proof>*

**lemma** *axioms-terms-ok:* *wf-theory* thy  $\implies$  *A*  $\in$  *axioms* thy  $\implies$  *term-ok* thy A

*<proof>*

**lemma** *axioms-typ-of-propT:* *wf-theory* thy  $\implies$  *A*  $\in$  *axioms* thy  $\implies$  *typ-of* A = *Some* propT

*<proof>*

**lemma** *propT-ok[simp]:* *wf-theory*  $\Theta \implies$  *typ-ok*  $\Theta$  propT

*<proof>*

**lemma** *term-ok-mk-eqD:* *term-ok*  $\Theta$  (mk-eq s t)  $\implies$  *term-ok*  $\Theta$  s  $\wedge$  *term-ok*  $\Theta$  t

*<proof>*

**lemma** *term-ok-app-eqD:* *term-ok*  $\Theta$  (s \$ t)  $\implies$  *term-ok*  $\Theta$  s  $\wedge$  *term-ok*  $\Theta$  t

*<proof>*

**lemma** *wf-type-Type-imp-mgd:*

*wf-sig*  $\Sigma \implies$  *wf-type*  $\Sigma$  (Ty n Ts)  $\implies$  *tcsigs* (*osig*  $\Sigma$ ) n  $\neq$  None

*<proof>*

**lemma** *term-ok-eta-expand:*

**assumes** *wf-theory*  $\Theta$  *term-ok*  $\Theta$  f *typ-of* f = *Some* ( $\tau \rightarrow \tau'$ ) *typ-ok*  $\Theta$   $\tau$

**shows** *term-ok*  $\Theta$  (Abs  $\tau$  (f \$ Bv 0))

*<proof>*

**lemma** *term-ok'-incr-bv:* *term-ok'*  $\Sigma$  t  $\implies$  *term-ok'*  $\Sigma$  (*incr-bv* inc lev t)

*<proof>*

**lemma** *term-ok'-subst-bv2*:  $term-ok' \Sigma s \implies term-ok' \Sigma u \implies term-ok' \Sigma (subst-bv2 s lev u)$   
 ⟨proof⟩

**lemma** *term-ok'-subst-bv*:  $term-ok' \Sigma (Abs T t) \implies term-ok' \Sigma (subst-bv (Fv x T) t)$   
 ⟨proof⟩

**lemma** *term-ok-subst-bv*:  $term-ok \Theta (Abs T t) \implies term-ok \Theta (subst-bv (Fv x T) t)$   
 ⟨proof⟩

**lemma** *term-ok-subst-bv2-0*:  $term-ok \Theta (Abs T t) \implies term-ok \Theta (subst-bv2 t 0 (Fv x T))$   
 ⟨proof⟩

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

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

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]*:  $subst-typ [] T = T$   
 ⟨proof⟩

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

**lemma** *subst-typ-simulates-tsubstT-gen'*:  $\text{distinct } l \implies \text{tvsT } T \subseteq \text{set } l$   
 $\implies \text{tsubstT } T \varrho = \text{subst-typ } (\text{map } (\lambda(x,y).((x,y), \varrho x y)) l) T$   
 ⟨proof⟩

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

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

**lemma** *tsubstT-simulates-subst-typ*:  $\text{subst-typ } \text{insts } T$   
 $= \text{tsubstT } T (\lambda \text{idn } S . \text{the-default } (T \text{v idn } S) (\text{lookup } (\lambda x. x = (\text{idn}, S)) \text{insts}))$   
 ⟨proof⟩

**lemma** *subst-typ-comp*:  
 $\text{subst-typ } \text{inst1 } (\text{subst-typ } \text{inst2 } T) = \text{subst-typ } (\text{map } (\text{apsnd } (\text{subst-typ } \text{inst1}))$   
 $\text{inst2 } @ \text{inst1}) T$   
 ⟨proof⟩

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

**fun** *subst-type-term* ::  $((\text{variable} \times \text{sort}) \times \text{typ}) \text{ list} \Rightarrow$   
 $((\text{variable} \times \text{typ}) \times \text{term}) \text{ list} \Rightarrow \text{term} \Rightarrow \text{term}$  **where**  
 $\text{subst-type-term } \text{instT } \text{insts } (\text{Ct } c T) = \text{Ct } c (\text{subst-typ } \text{instT } T)$   
 $| \text{subst-type-term } \text{instT } \text{insts } (\text{Fv } \text{idn } T) = (\text{let } T' = \text{subst-typ } \text{instT } T \text{ in}$   
 $\text{the-default } (\text{Fv } \text{idn } T') (\text{lookup } (\lambda x. x = (\text{idn}, T')) \text{insts}))$   
 $| \text{subst-type-term } - - (\text{Bv } n) = \text{Bv } n$   
 $| \text{subst-type-term } \text{instT } \text{insts } (\text{Abs } T t) = \text{Abs } (\text{subst-typ } \text{instT } T) (\text{subst-type-term}$   
 $\text{instT } \text{insts } t)$   
 $| \text{subst-type-term } \text{instT } \text{insts } (t \$ u) = \text{subst-type-term } \text{instT } \text{insts } t \$ \text{subst-type-term}$   
 $\text{instT } \text{insts } u$

**lemma** *subst-type-term-empty-no-change[simp]*:  $\text{subst-type-term } [] [] t = t$   
 ⟨proof⟩

**lemma** *subst-type-term-irrelevant-order*:  
**assumes** *instT-assms*:  $\text{distinct } (\text{map } \text{fst } \text{instT}) \text{ distinct } (\text{map } \text{fst } \text{instT}') \text{ set } \text{instT}$   
 $= \text{set } \text{instT}'$   
**assumes** *insts-assms*:  $\text{distinct } (\text{map } \text{fst } \text{insts}) \text{ distinct } (\text{map } \text{fst } \text{insts}') \text{ set } \text{insts}$   
 $= \text{set } \text{insts}'$   
**shows**  $\text{subst-type-term } \text{instT } \text{insts } t = \text{subst-type-term } \text{instT}' \text{insts}' t$

*<proof>*

**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  $\varrho$  ty)*  $\subseteq$  *set lt*

**shows** *subst (tsubst t  $\varrho$  ty)  $\varrho$  t*

*= subst-type-term (map ( $\lambda(x,y).(x,y), \varrho$  ty x y)) lty) (map ( $\lambda(x,y).(x,y), \varrho$  t x y)) lt) t*

*<proof>*

**corollary** *subst-type-term-simulates-subst-tsubst*: *subst (tsubst t  $\varrho$  ty)  $\varrho$  t*

*= subst-type-term (map ( $\lambda(x,y).(x,y), \varrho$  ty x y)) (SOME lty . *distinct lty*  $\wedge$  *tvs t = set lty*))*

*(map ( $\lambda(x,y).(x,y), \varrho$  t x y)) (SOME lt . *distinct lt*  $\wedge$  *fv (tsubst t  $\varrho$  ty) = set lt*)) t*

*<proof>*

**abbreviation** *subst-typ' pairs t*  $\equiv$  *map-types (subst-typ pairs) t*

**lemma** *subst-typ'-nil[simp]*: *subst-typ' [] A = A*

*<proof>*

**lemma** *subst-typ'-simulates-tsubst-gen'*: *distinct pairs*  $\implies$  *tvs t*  $\subseteq$  *set pairs*

$\implies$  *tsubst t  $\varrho$  = subst-typ' (map ( $\lambda(x,y).(x,y), \varrho$  x y)) pairs) t*

*<proof>*

**lemma** *subst-typ'-simulates-tsubst-gen*: *tsubst t  $\varrho$*

*= subst-typ' (map ( $\lambda(x,y).(x,y), \varrho$  x y)) (SOME l . *distinct l*  $\wedge$  *tvs t*  $\subseteq$  *set l*)) t*

*<proof>*

**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)*)*

*<proof>*

**lemma** *subst-type-add-degenerate-instance*:

*(idx,s)  $\notin$  set (map fst insts)  $\implies$  subst-typ insts T = subst-typ (((idx,s), Tv idx s)#insts) T*

*<proof>*

**lemma** *subst-typ'-add-degenerate-instance*:

*(idx,s)  $\notin$  set (map fst insts)  $\implies$  subst-typ' insts t = subst-typ' (((idx,s), Tv idx s)#insts) t*

*<proof>*

**lemma** *subst-typ'-comp*:

*subst-typ' inst1 (subst-typ' inst2 t) = subst-typ' (map (apsnd (subst-typ inst1))*

*inst2 @ inst1*) *t*  
 ⟨*proof*⟩

**lemma** *subst-typ'-AList-clearjunk*: *subst-typ' insts t = subst-typ' (AList.clearjunk insts) t*  
 ⟨*proof*⟩

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

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

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

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

**lemma** *subst-typ-stepwise*:  
**assumes** *distinct (map fst instT)*  
**assumes**  $\bigwedge x . x \in (\bigcup t \in \text{snd } ' \text{set } \text{instT} . \text{tvsT } t) \implies x \notin \text{fst } ' \text{set } \text{instT}$   
**shows** *subst-typ instT T = fold (λsingle acc . subst-typ [single] acc) instT T*  
 ⟨*proof*⟩

**corollary** *subst-typ-split-first*:  
**assumes** *distinct (map fst (x#xs))*  
**assumes**  $\bigwedge y . y \in (\bigcup t \in \text{snd } ' \text{set } (x\#xs) . \text{tvsT } t) \implies y \notin \text{fst } ' (\text{set } (x\#xs))$   
**shows** *subst-typ (x#xs) T = subst-typ xs (subst-typ [x] T)*  
 ⟨*proof*⟩

**corollary** *subst-typ-split-last*:  
**assumes** *distinct (map fst (xs @ [x]))*  
**assumes**  $\bigwedge y . y \in (\bigcup t \in \text{snd } ' (\text{set } (xs @ [x])) . \text{tvsT } t) \implies y \notin \text{fst } ' (\text{set } (xs @ [x]))$   
**shows** *subst-typ (xs @ [x]) T = subst-typ [x] (subst-typ xs T)*  
 ⟨*proof*⟩

**lemma** *subst-typ'-stepwise*:

**assumes** *distinct* (*map fst instT*)

**assumes**  $\bigwedge x . x \in (\bigcup t \in \text{snd} \text{ ' (set instT) . tvsT } t) \implies x \notin \text{fst} \text{ ' (set instT)}$

**shows** *subst-typ' instT t = fold* (*λsingle acc . subst-typ' [single] acc*) *instT t*

*<proof>*

**lemma** *subst-term-stepwise*:

**assumes** *distinct* (*map fst insts*)

**assumes**  $\bigwedge x . x \in (\bigcup t \in \text{snd} \text{ ' (set insts) . fv } t) \implies x \notin \text{fst} \text{ ' (set insts)}$

**shows** *subst-term insts t = fold* (*λsingle acc . subst-term [single] acc*) *insts t*

*<proof>*

**corollary** *subst-term-split-last*:

**assumes** *distinct* (*map fst (xs @ [x])*)

**assumes**  $\bigwedge y . y \in (\bigcup t \in \text{snd} \text{ ' (set (xs @ [x])) . fv } t) \implies y \notin \text{fst} \text{ ' (set (xs @ [x]))}$

**shows** *subst-term (xs @ [x]) t = subst-term [x] (subst-term xs t)*

*<proof>*

**corollary** *subst-type-term-stepwise*:

**assumes** *distinct* (*map fst instT*)

**assumes**  $\bigwedge x . x \in (\bigcup T \in \text{snd} \text{ ' (set instT) . tvsT } T) \implies x \notin \text{fst} \text{ ' (set instT)}$

**assumes** *distinct* (*map fst insts*)

**assumes**  $\bigwedge x . x \in (\bigcup t \in \text{snd} \text{ ' (set insts) . fv } t) \implies x \notin \text{fst} \text{ ' (set insts)}$

**shows** *subst-type-term instT insts t*

*= fold* (*λsingle . subst-term [single]*) *insts (fold* (*λsingle . subst-typ' [single]*) *instT t)*

*<proof>*

**lemma** *distinct-fst-imp-distinct*: *distinct* (*map fst l*)  $\implies$  *distinct l* *<proof>*

**lemma** *distinct-kv-list*: *distinct l*  $\implies$  *distinct* (*map* ( $\lambda x . (x, f x)$ ) *l*) *<proof>*

**lemma** *subst-subst-term*:

**assumes** *distinct l* **and** *fv t*  $\subseteq$  *set l*

**shows** *subst t ρ = subst-term* (*map* ( $\lambda x . (x, \text{case-prod } \rho x)$ ) *l*) *t*

*<proof>*

**lemma** *subst-term-subst*:

**assumes** *distinct* (*map fst l*)

**shows** *subst-term l t = subst t (fold* ( $\lambda((idn, T), t) f x y . \text{if } x=idn \wedge y=T \text{ then } t$  *else* *f x y*) *l Fv*)

*<proof>*

**lemma** *subst-typ-combine-single*:

**assumes** *fresh-idn*  $\notin$  *fst ' tvsT τ*

**shows**  $\text{subst-typ} [((\text{fresh-idn}, S), \tau 2)] (\text{subst-typ} [((\text{idn}, S), \text{Tv fresh-idn } S)] \tau)$   
 $= \text{subst-typ} [((\text{idn}, S), \tau 2)] \tau$   
 $\langle \text{proof} \rangle$

**lemma** *subst-typ-combine*:

**assumes**  $\text{length fresh-idns} = \text{length insts}$   
**assumes**  $\text{distinct fresh-idns}$   
**assumes**  $\text{distinct} (\text{map fst insts})$   
**assumes**  $\forall \text{idn} \in \text{set fresh-idns} . \text{idn} \notin \text{fst} \text{ ' } (\text{tvsT } \tau \cup (\bigcup \text{ty} \in \text{snd} \text{ ' set insts} .$   
 $(\text{tvsT ty}))$   
 $\cup (\text{fst} \text{ ' set insts}))$   
**shows**  $\text{subst-typ insts } \tau$   
 $= \text{subst-typ} (\text{zip} (\text{zip fresh-idns} (\text{map snd} (\text{map fst insts}))) (\text{map snd insts}))$   
 $(\text{subst-typ} (\text{zip} (\text{map fst insts}) (\text{map2 Tv fresh-idns} (\text{map snd} (\text{map fst insts}))))$   
 $\tau)$   
 $\langle \text{proof} \rangle$

**lemma** *subst-typ-combine'*:

**assumes**  $\text{length fresh-idns} = \text{length insts}$   
**assumes**  $\text{distinct fresh-idns}$   
**assumes**  $\text{distinct} (\text{map fst insts})$   
**assumes**  $\forall \text{idn} \in \text{set fresh-idns} . \text{idn} \notin \text{fst} \text{ ' } (\text{tvsT } \tau \cup (\bigcup \text{ty} \in \text{snd} \text{ ' set insts} .$   
 $(\text{tvsT ty}))$   
 $\cup (\text{fst} \text{ ' set insts}))$   
**shows**  $\text{subst-typ insts } \tau$   
 $= \text{fold} (\lambda \text{single acc} . \text{subst-typ} [\text{single}] \text{acc}) (\text{zip} (\text{zip fresh-idns} (\text{map snd} (\text{map}$   
 $\text{fst insts}))) (\text{map snd insts}))$   
 $(\text{fold} (\lambda \text{single acc} . \text{subst-typ} [\text{single}] \text{acc}) (\text{zip} (\text{map fst insts}) (\text{map2 Tv}$   
 $\text{fresh-idns} (\text{map snd} (\text{map fst insts})))) \tau)$   
 $\langle \text{proof} \rangle$

**lemma** *subst-typ'-combine*:

**assumes**  $\text{length fresh-idns} = \text{length insts}$   
**assumes**  $\text{distinct fresh-idns}$   
**assumes**  $\text{distinct} (\text{map fst insts})$   
**assumes**  $\forall \text{idn} \in \text{set fresh-idns} . \text{idn} \notin \text{fst} \text{ ' } (\text{tvs } t \cup (\bigcup \text{ty} \in \text{snd} \text{ ' set insts} . (\text{tvsT}$   
 $\text{ty}))$   
 $\cup (\text{fst} \text{ ' set insts}))$   
**shows**  $\text{subst-typ}' \text{ insts } t$   
 $= \text{subst-typ}' (\text{zip} (\text{zip fresh-idns} (\text{map snd} (\text{map fst insts}))) (\text{map snd insts}))$   
 $(\text{subst-typ}' (\text{zip} (\text{map fst insts}) (\text{map2 Tv fresh-idns} (\text{map snd} (\text{map fst insts}))))$   
 $t)$   
 $\langle \text{proof} \rangle$

**lemma** *subst-term-combine*:

**assumes**  $\text{length fresh-idns} = \text{length insts}$   
**assumes**  $\text{distinct fresh-idns}$   
**assumes**  $\text{distinct} (\text{map fst insts})$



**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**  $subst-term\ insts\ t$   
 $= subst-term\ (zip\ (zip\ fresh-idns\ (map\ snd\ (map\ fst\ insts)))\ (map\ snd\ insts))$   
 $(subst-term\ (zip\ (map\ fst\ insts)\ (map2\ Fv\ fresh-idns\ (map\ snd\ (map\ fst\ insts))))$   
 $t$   
 $\langle proof \rangle$

**corollary** *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**  $subst-term\ insts\ t$   
 $= fold\ (\lambda single\ acc . subst-term\ [single]\ acc)\ (zip\ (zip\ fresh-idns\ (map\ snd\ (map\ fst\ insts)))\ (map\ snd\ insts))$   
 $(fold\ (\lambda single\ acc . subst-term\ [single]\ acc)\ (zip\ (map\ fst\ insts)\ (map2\ Fv\ fresh-idns\ (map\ snd\ (map\ fst\ insts))))\ t$   
 $\langle proof \rangle$

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

**assumes**  $\neg loose-bvar\ t\ n\ is-closed\ b$   
**shows**  $\neg loose-bvar\ (subst-term\ [(idn, T), b])\ t\ n$   
 $\langle proof \rangle$

**lemma** *bind-fv2-subst-bv1-eq-subst-term*:

**assumes**  $\neg loose-bvar\ t\ n\ is-closed\ b$   
**shows**  $subst-term\ [(idn, T), b]\ t = subst-bv1\ (bind-fv2\ (idn, T)\ n\ t)\ n\ b$   
 $\langle proof \rangle$

**corollary**

**assumes**  $is-closed\ t\ is-closed\ b$   
**shows**  $subst-bv\ b\ (bind-fv\ (idn, T)\ t) = (subst-term\ [(idn, T), b])\ t$   
 $\langle proof \rangle$

**corollary** *instantiate-var-same-typ*:

**assumes**  $typ-a: typ-of\ a = Some\ \tau$   
**assumes**  $closed-B: \neg loose-bvar\ B\ lev$   
**shows**  $subst-bv1\ (bind-fv2\ (x, \tau)\ lev\ B)\ lev\ a = subst-term\ [(x, \tau), a]\ B$   
 $\langle proof \rangle$

**corollary** *instantiate-var-same-typ'*:

**assumes**  $typ-a: typ-of\ a = Some\ \tau$   
**assumes**  $closed-B: is-closed\ B$   
**shows**  $subst-bv\ a\ (bind-fv\ (x, \tau)\ B) = subst-term\ [(x, \tau), a]\ B$   
 $\langle proof \rangle$

**corollary** *instantiate-var-same-type''*:

**assumes** *typ-a*: *typ-of a = Some  $\tau$*

**assumes** *closed-B*: *is-closed B*

**shows** *Abs  $\tau$  (bind-fv (x,  $\tau$ ) B) · a = subst-term [(x,  $\tau$ ), a] B*

*<proof>*

**lemma** *instantiate-vars-same-tyt*:

**assumes** *tyts*: *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*

*<proof>*

**corollary** *instantiate-vars-same-tyt'*:

**assumes** *tyts*: *list-all ( $\lambda$ ((idx, ty), t) . *typ-of t = Some ty*) insts*

**assumes** *closed-B*:  $\neg$  *loose-bvar B lev*

**assumes** *distinct*: *distinct (map fst insts)*

**assumes** *no-overlap*:  $\bigwedge x . x \in (\bigcup t \in \text{snd } \text{'(set insts) . fv t} \implies x \notin \text{fst } \text{'(set insts)}$

**shows** *fold ( $\lambda$ ((idx, ty), t) B . *subst-bv1 (bind-fv2 (idx, ty) lev B) lev t*) insts B*  
*= subst-term insts B*

*<proof>*

**end**

## 7 Names

**theory** *Name*

**imports** *Preliminaries Term*

*HOL-Library.Char-ord*

**begin**

**fun** *fresh-name* :: *string set  $\Rightarrow$  string where*

*fresh-name S = (if S=empty then "a" else replicate (Max (length 'S) + 1) (CHR "a"))*

**lemma** *fresh-name-fresh*:

**assumes** *finite S*

**shows** *fresh-name S  $\notin$  S*

*<proof>*

**context**

**includes** *String.literal.lifting*

**begin**

**lift-definition** *fresh-name'* :: *String.literal set  $\Rightarrow$  String.literal is fresh-name*

*<proof>*

**lemma** [code]:  $\text{fresh-name}' S = \text{String.implode} (\text{fresh-name} (\text{String.explode } ' S))$   
 ⟨proof⟩

**lemma** *fresh-name'-fresh*:  
**assumes** *finite S*  
**shows**  $\text{fresh-name}' S \notin S$   
 ⟨proof⟩  
**end**

**fun** *variant-name* ::  $\text{name} \Rightarrow \text{name set} \Rightarrow (\text{name} \times \text{name set})$  **where**  
*variant-name s S* = (let  $s' = (\text{fresh-name}' S)$  in  $(s', \text{insert } s' S)$ )

**lemma** *variant-name-fresh*:  
**assumes** *finite S*  
**shows**  $\text{fst} (\text{variant-name } s S) \notin S$   
 ⟨proof⟩

**lemma** *variant-name-adds*:  
**shows**  $\text{snd} (\text{variant-name } s S) = \text{insert} (\text{fst} (\text{variant-name } s S)) S$   
 ⟨proof⟩

**fun** *name* ::  $\text{variable} \Rightarrow \text{name}$  **where**  
*name (variable.Free n)* =  $n$   
 | *name (Var (n,-))* =  $n$

**fun** *variant-variable* ::  $\text{variable} \Rightarrow \text{variable set} \Rightarrow (\text{variable} \times \text{variable set})$  **where**  
*variant-variable (variable.Free n) S* = (let  $s' = \text{fresh-name}' (\text{name } ' S)$  in  
 $(\text{Free } s', \text{insert} (\text{variable.Free } s') S)$ )  
 | *variant-variable (Var (n,-)) S* = (let  $s' = \text{fresh-name}' (\text{name } ' S)$  in  
 $(\text{Var } (s',0), \text{insert} (\text{Var } (s',0)) S)$ )

**lemma** *variant-variable-fresh*:  
**assumes** *finite S*  
**shows**  $\text{fst} (\text{variant-variable } s S) \notin S$   
 ⟨proof⟩

**lemma** *variant-variable-adds*:  
**shows**  $\text{snd} (\text{variant-variable } s S) = \text{insert} (\text{fst} (\text{variant-variable } s S)) S$   
 ⟨proof⟩

**fun** *variant-variables* ::  $\text{nat} \Rightarrow \text{variable} \Rightarrow \text{variable set} \Rightarrow (\text{variable list} \times \text{variable set})$  **where**

$variant\text{-}variables\ 0 - S = ([], S)$   
 $| variant\text{-}variables\ (Suc\ n)\ s\ S =$   
 $\quad (let\ (s', S') = variant\text{-}variable\ s\ S\ in$   
 $\quad\quad (let\ (ss, S'') = variant\text{-}variables\ n\ s'\ S'\ in$   
 $\quad\quad\quad (s'\#ss, S''))$

**lemma** *variant-names-fresh*:

**assumes** *finite S*

**shows**  $\forall s \in set\ (fst\ (variant\text{-}variables\ n\ s\ S)) . s \notin S$

*<proof>*

**lemma** *variant-names-distinct*:

**assumes** *finite S*

**shows** *distinct*  $(fst\ (variant\text{-}variables\ n\ s\ S))$

*<proof>*

**corollary** *variant-names-amount*:

**assumes** *finite S*

**shows**  $length\ (fst\ (variant\text{-}variables\ n\ s\ S)) = n$

*<proof>*

**abbreviation** *fresh-rename-ns*  $n\ B\ insts\ G \equiv fst\ (variant\text{-}variables\ n\ (Free\ STR\ "lol"))$

$(fst\ ' (fv\ B \cup (\bigcup t \in snd\ ' set\ insts . fv\ t) \cup (fst\ ' set\ insts)) \cup G)$

**abbreviation** *fresh-rename-idns*  $n\ B\ insts \equiv fresh\text{-}rename\text{-}ns\ n\ B\ insts$

**lemma** *map-Pair-zip-replicate-conv*:  $map\ (\lambda x. Pair\ x\ c)\ l = zip\ l\ (replicate\ (length\ l)\ c)$

*<proof>*

**lemma** *distinct-fresh-rename-ns*:  $finite\ G \implies distinct\ (fresh\text{-}rename\text{-}ns\ n\ B\ insts\ G)$

*<proof>*

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

$nm \notin (fst\ ' (fv\ B \cup (\bigcup t \in snd\ ' set\ insts . (fv\ t)) \cup (fst\ ' set\ insts)) \cup G)$

*<proof>*

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

*<proof>*

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

*<proof>*

**lemma** *fresh-fresh-rename-idns*:  $finite\ G \implies \forall nm \in set\ (fresh\text{-}rename\text{-}idns\ n\ B$

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

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

**end**

## 8 Beta Normalization

**theory** *BetaNorm*  
**imports** *Term*  
**begin**

**inductive** *beta* :: *term*  $\Rightarrow$  *term*  $\Rightarrow$  *bool* (**infixl**  $\rightarrow_\beta$  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**  $\rightarrow_{\beta^*}$  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!*]:

$s \rightarrow_{\beta^*} s' \implies Abs\ T\ s \rightarrow_{\beta^*} Abs\ T\ s'$   
 ⟨*proof*⟩

**lemma** *rtrancl-beta-AppL*:

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

**lemma** *rtrancl-beta-AppR*:

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

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

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

*<proof>*

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

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

*<proof>*

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

*<proof>*

**theorem** *lift-preserves-beta* [simp]:

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

*<proof>*

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

*<proof>*

**theorem** *subst-bv2-preserves-beta2* [simp]:  $r \rightarrow_{\beta} s \implies \text{subst-bv2 } t \ i \ r \rightarrow_{\beta^*} \text{subst-bv2 } t \ i \ s$

*<proof>*

**theorem** *subst-bv2-preserves-beta2'*:  $r \rightarrow_{\beta^*} s \implies \text{subst-bv2 } t \ i \ r \rightarrow_{\beta^*} \text{subst-bv2 } t \ i \ s$

*<proof>*

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

*<proof>*

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

*<proof>*

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

*<proof>*

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

*<proof>*

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

*<proof>*

**lemma** *beta-norm-imp-beta-reds*: **assumes**  $\text{beta-norm } t = \text{Some } t'$  **shows**  $t \rightarrow_{\beta^*} t'$

*<proof>*

**corollary**  $\text{beta-norm } t = \text{Some } t' \implies \text{typ-of1 } Ts \ t = \text{Some } T \implies \text{typ-of1 } Ts \ t' = \text{Some } T$

*<proof>*

**lemma** *beta-imp-beta-norm*: **assumes**  $t \rightarrow_\beta t' \dashv$  *beta-reducible*  $t'$  **shows** *beta-norm*  
 $t = \text{Some } t'$   
*<proof>*

**lemma** *beta-subst-bv1*:  $s \rightarrow_\beta t \implies \text{subst-bv1 } s \text{ lev } x \rightarrow_\beta \text{subst-bv1 } t \text{ lev } x$   
*<proof>*

**lemma** *beta-subst-bv*:  $s \rightarrow_\beta t \implies \text{subst-bv } x \text{ } s \rightarrow_\beta \text{subst-bv } x \text{ } t$   
*<proof>*

**lemma** *subst-bv1-beta*:  
 $\text{subst-bv1 } s \text{ (length } (T\#Ts)) \text{ } x \rightarrow_\beta \text{subst-bv1 } t \text{ (length } (T\#Ts)) \text{ } x$   
 $\implies \text{typ-of1 } Ts \text{ } s = \text{Some } ty$   
 $\implies \text{typ-of1 } Ts \text{ } t = \text{Some } ty$   
 $\implies s \rightarrow_\beta t$   
*<proof>*

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

**lemma** *subst-bvs1'-empty* [*simp*]:  $\text{subst-bvs1'} \ t \ \text{lev } [] = t$   
*<proof>*

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

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

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

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

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

**lemma** *incr-bv-incr-bv*:

$i < k + 1 \implies \text{incr-bv inc2 } (k+\text{incr1}) (\text{incr-bv inc1 } i t) = \text{incr-bv inc1 } i (\text{incr-bv inc2 } k t)$   
(proof)

**lemma** *subst-bvs1-subst-bvs1'*:  $\text{subst-bvs1 } t n s = \text{subst-bvs1'} t n (\text{map } (\text{incr-bv } n 0) s)$   
(proof)

**theorem** *subst-bvs1-subst-bvs1'-0*:  $\text{subst-bvs1 } t 0 s = \text{subst-bvs1'} t 0 s$   
(proof)

**corollary** *subst-bvs-subst-bvs1'*:  $\text{subst-bvs } s t = \text{subst-bvs1'} t 0 s$   
(proof)

**lemma** *no-loose-bvar-subst-bvs1'-unchanged*:  $\neg \text{loose-bvar } t \text{ lev} \implies \text{subst-bvs1'} t \text{ lev args} = t$   
(proof)

**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)

**lemma** *not-loose-bvar-incr-bv*:  $\neg \text{loose-bvar } a \text{ lev} \implies \neg \text{loose-bvar } (\text{incr-bv inc lev } a) (\text{lev+inc})$   
(proof)

**lemma** *not-loose-bvar-incr-bv-less*:  $i < j \implies \neg \text{loose-bvar } (\text{incr-bv inc } i a) (\text{lev+inc}) \implies \neg \text{loose-bvar } (\text{incr-bv inc } j a) (\text{lev+inc})$   
(proof)

**lemma** *subst-bvs1'-step-work*:  $\forall x \in \text{set args} . \text{is-closed } x \implies \neg \text{loose-bvar } (\text{subst-bv2 } t \text{ lev } a) \text{ lev} \implies \text{subst-bvs1'} t \text{ lev } (a\#\text{args}) = \text{subst-bvs1'} (\text{subst-bv2 } t \text{ lev } a) \text{ lev args}$   
(proof)

**lemma** *is-closed-subst-bv2-unchanged*:  $\text{is-closed } t \implies \text{subst-bv2 } t n u = t$   
(proof)

**lemma** *subst-bvs1'-step-extend-lower-level*:  $\forall x \in \text{set } (a\#\text{args}) . \text{is-closed } x \implies \text{subst-bv2 } (\text{subst-bvs1'} t (\text{Suc lev}) \text{ args}) \text{ lev } a = \text{subst-bvs1'} t \text{ lev } (a\#\text{args})$   
(proof)

**corollary** *subst-bvs-extend-lower-level*:



$\forall x \in \text{set } (a\#\text{args}) . \text{is-closed } x \implies$   
 $\text{subst-bv } a \text{ (subst-bvs1' } t \text{ 1 args)} = \text{subst-bvs } (a\#\text{args}) \text{ } t$   
 ⟨proof⟩

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

$\forall x \in \text{set } u . \text{is-closed } x \implies r \rightarrow_\beta s \implies \text{subst-bvs1' } r \text{ } k \text{ } u \rightarrow_\beta \text{subst-bvs1' } s \text{ } k \text{ } u$   
 ⟨proof⟩

**lemma** *subst-bvs1'-fold*:  $\forall x \in \text{set } \text{args} . \text{is-closed } x \implies$

$\text{subst-bvs1' } t \text{ lev args} = \text{fold } (\lambda \text{arg } t . \text{subst-bv2 } t \text{ lev arg}) \text{ args } t$   
 ⟨proof⟩

**lemma** *subst-bvs1'-Abs[simp]*:  $\forall x \in \text{set } \text{args} . \text{is-closed } x \implies$

$\text{subst-bvs1' } (\text{Abs } T \text{ } t) \text{ lev args} = \text{Abs } T \text{ (subst-bvs1' } t \text{ (Suc lev) args)}$   
 ⟨proof⟩

**lemma** *subst-bvs-Abs[simp]*:  $\forall x \in \text{set } \text{args} . \text{is-closed } x \implies$

$\text{subst-bvs } \text{args } (\text{Abs } T \text{ } t) = \text{Abs } T \text{ (subst-bvs1' } t \text{ 1 args)}$   
 ⟨proof⟩

**lemma** *subst-bvs1'-incr-bv [simp]*:

$\text{subst-bvs1' } (\text{incr-bv } (\text{length } ss) \text{ } k \text{ } t) \text{ } k \text{ } ss = t$   
 ⟨proof⟩

**lemma** *lift-subst-bvs1' [simp]*:

$j < i + 1 \implies \text{lift } (\text{subst-bvs1' } t \text{ } j \text{ } ss) \text{ } i$   
 $= \text{subst-bvs1' } (\text{lift } t \text{ (} i + \text{length } ss)) \text{ } j \text{ (map } (\lambda s . \text{lift } s \text{ } i) \text{ } ss)$   
 ⟨proof⟩

**lemma** *lift-subst-bvs1'-lt*:

$i < j + 1 \implies \text{lift } (\text{subst-bvs1' } t \text{ } j \text{ } ss) \text{ } i$   
 $= \text{subst-bvs1' } (\text{lift } t \text{ } i) \text{ (} j + 1 \text{) (map } (\lambda s . \text{lift } s \text{ } i) \text{ } ss)$   
 ⟨proof⟩

**lemma** *subst-bvs1'-subst-bv2*:

$i < j + 1 \implies$   
 $\text{subst-bv2}(\text{subst-bvs1' } t \text{ (Suc } j) \text{ (map } (\lambda v . \text{lift } v \text{ } i) \text{ } vs)) \text{ } i \text{ (subst-bvs1' } u \text{ } j \text{ } vs)$   
 $= \text{subst-bvs1' } (\text{subst-bv2 } t \text{ } i \text{ } u) \text{ } j \text{ } vs$   
 ⟨proof⟩

**lemma** *fv-subst-bv2-upper-bound*:  $\text{fv } (\text{subst-bv2 } t \text{ lev } u) \subseteq \text{fv } t \cup \text{fv } u$

⟨proof⟩

**lemma** *beta-fv*:  $s \rightarrow_\beta t \implies \text{fv } t \subseteq \text{fv } s$

⟨proof⟩

**lemma** *loose-bvar1-subst-bvs1'-closed*:  $\neg \text{loose-bvar1 } t \text{ lev} \implies \text{lev} < k \implies \forall x \in \text{set}$   
 $us . \text{is-closed } x$

$\implies \neg \text{loose-bvar1 } (\text{subst-bvs1' } t \text{ } k \text{ } us) \text{ lev}$

⟨proof⟩

**lemma** *is-closed-subst-bvs1'-closed*:  $\neg$  *is-dependent*  $t \implies \forall x \in \text{set } us . \text{is-closed } x$   
 $\implies \neg$  *is-dependent* (*subst-bvs1'*  $t$  (*Suc*  $k$ )  $us$ )  
 ⟨*proof*⟩

**end**

Facts about beta normalization involving theories

**theory** *BetaNormProof*  
**imports** *BetaNorm Theory*  
**begin**

**lemma** *beta-preserves-term-ok'*:  $\text{term-ok}' \Sigma r \implies r \rightarrow_{\beta} s \implies \text{term-ok}' \Sigma s$   
 ⟨*proof*⟩

**lemma** *beta-preserves-term-ok*:  $\text{term-ok } \Theta r \implies r \rightarrow_{\beta} s \implies \text{term-ok } \Theta s$   
 ⟨*proof*⟩

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

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

**corollary** *term-ok-beta-norm*:  $\text{term-ok } \text{thy } t \implies \text{beta-norm } t = \text{Some } t' \implies \text{term-ok } \text{thy } t'$   
 ⟨*proof*⟩

**end**

## 9 Eta Normalization

**theory** *EtaNorm*  
**imports** *Term BetaNorm*  
**begin**

**inductive**

*eta* :: *term*  $\Rightarrow$  *term*  $\Rightarrow$  *bool* (**infixl**  $\rightarrow_{\eta}$  50)

**where**

*eta* [*simp*, *intro*]:  $\neg$  *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**  $\rightarrow_{\eta^*}$  50) **where**

$s \rightarrow_{\eta^*} t \equiv \text{eta}^{**} \ s \ t$

**abbreviation**

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

**inductive-cases**  $\text{eta-cases}$  [elim!]:

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

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

**lemma**  $\text{free-lift}$  [simp]:

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

**lemma**  $\text{free-subst-bv2}$  [simp]:

$\text{loose-bvar1 } (\text{subst-bv2 } s k t) i =$   
 $(\text{loose-bvar1 } s k \wedge \text{loose-bvar1 } t i \vee \text{loose-bvar1 } s (\text{if } i < k \text{ then } i \text{ else } i + 1))$   
 ⟨proof⟩

**lemma**  $\text{free-eta}$ :  $s \rightarrow_\eta t \Longrightarrow \text{loose-bvar1 } t i = \text{loose-bvar1 } s i$   
 ⟨proof⟩

**lemma**  $\text{not-free-eta}$ :

$s \rightarrow_\eta t \Longrightarrow \neg \text{loose-bvar1 } s i \Longrightarrow \neg \text{loose-bvar1 } t i$   
 ⟨proof⟩

**lemma**  $\text{no-loose-bvar1-subst-bv2-decr}$ :  $\neg \text{loose-bvar1 } t i \Longrightarrow \text{subst-bv2 } t i x = \text{decr } i t$   
 ⟨proof⟩

**lemma**  $\text{eta-subst-bv2}$  [simp]:

$s \rightarrow_\eta t \Longrightarrow \text{subst-bv2 } s i u \rightarrow_\eta \text{subst-bv2 } t i u$   
 ⟨proof⟩

**theorem**  $\text{lift-subst-bv2-dummy}$ :  $\neg \text{loose-bvar } s i \Longrightarrow \text{lift } (\text{decr } i s) i = s$   
 ⟨proof⟩

**lemma**  $\text{decr-is-closed}$ [simp]:  $\text{is-closed } t \Longrightarrow \text{decr lev } t = t$   
 ⟨proof⟩

**lemma**  $\text{eta-reducible-imp-eta-step}$ :  $\text{eta-reducible } t \Longrightarrow \exists t'. t \rightarrow_\eta t'$   
 ⟨proof⟩

**lemma**  $\text{eta-step-imp-eta-reducible}$ :  $t \rightarrow_\eta t' \Longrightarrow \text{eta-reducible } t$

*<proof>*

**lemma** *eta-reds-appR*:  $s \rightarrow_{\eta}^* t \implies u \$ s \rightarrow_{\eta}^* u \$ t$   
*<proof>*

**lemma** *eta-reds-appL*:  $s \rightarrow_{\eta}^* t \implies s \$ u \rightarrow_{\eta}^* t \$ u$   
*<proof>*

**lemma** *eta-reds-abs*:  $s \rightarrow_{\eta}^* t \implies \text{Abs } T \ s \rightarrow_{\eta}^* \text{Abs } T \ t$   
*<proof>*

**lemma** *eta-norm-imp-eta-reds*: **assumes** *eta-norm*  $t = t'$  **shows**  $t \rightarrow_{\eta}^* t'$   
*<proof>*

**lemma** *rtrancl-eta-App*:  
 $s \rightarrow_{\eta}^* s' \implies t \rightarrow_{\eta}^* t' \implies s \$ t \rightarrow_{\eta}^* s' \$ t'$   
*<proof>*

**lemma** *eta-preserves-typ-of1*:  $t \rightarrow_{\eta} t' \implies \text{typ-of1 } Ts \ t = \text{Some } \tau \implies \text{typ-of1 } Ts \ t' = \text{Some } \tau$   
*<proof>*

**lemma** *eta-preserves-typ-of*:  $t \rightarrow_{\eta} t' \implies \text{typ-of } t = \text{Some } \tau \implies \text{typ-of } t' = \text{Some } \tau$   
*<proof>*

**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>*

**lemma** *eta-star-preserves-typ-of*:  $r \rightarrow_{\eta}^* s \implies \text{typ-of } r = \text{Some } T \implies \text{typ-of } s = \text{Some } T$   
*<proof>*

**lemma** *subst-bvs1'-decr*:  $\forall x \in \text{set } us. \text{is-closed } x \implies \neg \text{loose-bvar1 } t \ k \implies \text{subst-bvs1}' (\text{decr } k \ t) \ k \ us = \text{decr } k (\text{subst-bvs1}' t (\text{Suc } k) \ us)$   
*<proof>*

**lemma** *subst-bvs-decr*:  $\forall x \in \text{set } us. \text{is-closed } x \implies \neg \text{is-dependent } t \implies \text{subst-bvs } us (\text{decr } 0 \ t) = \text{decr } 0 (\text{subst-bvs1}' t \ 1 \ us)$   
*<proof>*

**end**

Facts about eta normalization involving theories

**theory** *EtaNormProof*

**imports** *EtaNorm Theory*

*BetaNormProof*

**begin**

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

**lemma** *eta-preserves-term-ok'*:  $term-ok' \Sigma r \implies r \rightarrow_{\eta} s \implies term-ok' \Sigma s$   
 ⟨proof⟩

**lemma** *eta-preserves-term-ok*:  $term-ok \Theta r \implies r \rightarrow_{\eta} s \implies term-ok \Theta s$   
 ⟨proof⟩

**lemma** *eta-star-preserves-term-ok'*:  $r \rightarrow_{\eta}^* s \implies term-ok' \Sigma r \implies term-ok' \Sigma s$   
 ⟨proof⟩

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

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

end

## 10 Logic

**theory** *Logic*

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

**term** *proves*

**abbreviation** *inst-ok*  $\Theta\ insts \equiv$

*distinct* (*map fst insts*) — No duplicates, makes stuff easier  
 $\wedge$  *list-all* (*typ-ok*  $\Theta$ ) (*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*:

$inst-ok\ \Theta\ insts \implies wf-inst\ \Theta\ (\lambda idn\ S . the-default\ (Tv\ idn\ S)\ (lookup\ (\lambda x . x=(idn, S))\ insts))$   
 ⟨proof⟩

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

**corollary** *term-ok-eta-norm*:  $term-ok\ thy\ t \implies term-ok\ thy\ (eta-norm\ t)$   
 ⟨proof⟩

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

**lemma** *beta-eta-norm*  $t = Some\ t' \implies \neg eta-reducible\ t'$   
 ⟨proof⟩

**lemma** *term-ok-beta-eta-norm*:  $\text{term-ok thy } t \implies \text{beta-eta-norm } t = \text{Some } t' \implies \text{term-ok thy } t'$

*<proof>*

**lemma** *typ-of-beta-eta-norm*:

$\text{typ-of } t = \text{Some } T \implies \text{beta-eta-norm } t = \text{Some } t' \implies \text{typ-of } t' = \text{Some } T$

*<proof>*

**lemma** *inst-ok-nil[simp]*:  $\text{inst-ok } \Theta \ [] \ \langle \text{proof} \rangle$

**lemma** *axiom-subst-typ'*:

**assumes** *wf-theory*  $\Theta$   $A \in \text{axioms } \Theta$  *inst-ok*  $\Theta$  *insts*

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

*<proof>*

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

*<proof>*

**lemma** *has-sort-Tv-refl*:  $\text{wf-osig } \text{oss} \implies \text{sort-ex } (\text{subclass } \text{oss}) S \implies \text{has-sort } \text{oss} (Tv \ v \ S) \ S$

*<proof>*

**lemma** *has-sort-Tv-refl'*:

$\text{wf-theory } \Theta \implies \text{typ-ok } \Theta (Tv \ v \ S) \implies \text{has-sort } (\text{osig } (\text{sig } \Theta)) (Tv \ v \ S) \ S$

*<proof>*

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

$\text{wf-theory } \Theta \implies \text{distinct } l \implies \forall (v, S) \in \text{set } l . \text{typ-ok } \Theta (Tv \ v \ S) \implies \text{wf-inst } \Theta \ \varrho$

$\implies \text{inst-ok } \Theta (\text{map } (\lambda(v, S) . ((v, S), \varrho \ v \ S)) \ l)$

*<proof>*

**lemma** *typs-of-fv-subset-Types*:  $\text{snd } ' \text{fv } t \subseteq \text{Types } t$

*<proof>*

**lemma** *osig-tvsT-subset-SortsT*:  $\text{snd } ' \text{tvsT } T \subseteq \text{SortsT } T$

*<proof>*

**lemma** *osig-tvs-subset-Sorts*:  $\text{snd } ' \text{tvs } t \subseteq \text{Sorts } t$

*<proof>*

**lemma** *term-ok-Types-imp-typ-ok-pre*:

$\text{is-std-sig } \Sigma \implies \text{term-ok}' \Sigma \ t \implies \tau \in \text{Types } t \implies \text{typ-ok-sig } \Sigma \ \tau$

*<proof>*

**lemma** *term-ok-Types-typ-ok*:  $\text{wf-theory } \Theta \implies \text{term-ok } \Theta \ t \implies \tau \in \text{Types } t \implies \text{typ-ok } \Theta \ \tau$

*<proof>*

**lemma** *term-ok-fv-imp-typ-ok-pre*:

$\text{is-std-sig } \Sigma \implies \text{term-ok}' \Sigma \ t \implies (x, \tau) \in \text{fv } t \implies \text{typ-ok-sig } \Sigma \ \tau$

*<proof>*

**lemma** *term-ok-vars-typ-ok*:  $wf\text{-theory } \Theta \implies term\text{-ok } \Theta t \implies (x, \tau) \in fv\ t \implies typ\text{-ok } \Theta \tau$   
*<proof>*

**lemma** *typ-ok-TFreesT-imp-sort-ok-pre*:  
 $is\text{-std-sig } \Sigma \implies typ\text{-ok-sig } \Sigma T \implies (x, S) \in tvsT\ T \implies wf\text{-sort } (subclass\ (osig\ \Sigma))\ S$   
*<proof>*

**lemma** *term-ok-TFrees-imp-sort-ok-pre*:  
 $is\text{-std-sig } \Sigma \implies term\text{-ok}'\ \Sigma t \implies (x, S) \in tvs\ t \implies wf\text{-sort } (subclass\ (osig\ \Sigma))\ S$   
*<proof>*

**lemma** *typ-ok-tvsT-imp-sort-ok-pre*:  
 $is\text{-std-sig } \Sigma \implies typ\text{-ok-sig } \Sigma T \implies (x, S) \in tvsT\ T \implies wf\text{-sort } (subclass\ (osig\ \Sigma))\ S$   
*<proof>*

**lemma** *term-ok-tvars-sort-ok*:  
**assumes**  $wf\text{-theory } \Theta\ term\text{-ok } \Theta t\ (x, S) \in tvs\ t$   
**shows**  $wf\text{-sort } (subclass\ (osig\ (sig\ \Theta)))\ S$   
*<proof>*

**lemma** *term-ok'-bind-fv2*:  
**assumes**  $term\text{-ok}'\ \Sigma t$   
**shows**  $term\text{-ok}'\ \Sigma\ (bind\text{-fv2 } (v, T)\ lev\ t)$   
*<proof>*

**lemma** *term-ok'-bind-fv*:  
**assumes**  $term\text{-ok}'\ \Sigma t$   
**shows**  $term\text{-ok}'\ \Sigma\ (bind\text{-fv } (v, \tau)\ t)$   
*<proof>*

**lemma** *term-ok'-Abs-fv*:  
**assumes**  $term\text{-ok}'\ \Sigma t\ typ\text{-ok-sig } \Sigma \tau$   
**shows**  $term\text{-ok}'\ \Sigma\ (Abs\ \tau\ (bind\text{-fv } (v, \tau)\ t))$   
*<proof>*

**lemma** *term-ok'-mk-all*:  
**assumes**  $wf\text{-theory } \Theta$  **and**  $term\text{-ok}'\ (sig\ \Theta)\ B$  **and**  $typ\text{-of } B = Some\ propT$   
**and**  $typ\text{-ok } \Theta \tau$   
**shows**  $term\text{-ok}'\ (sig\ \Theta)\ (mk\text{-all } x\ \tau\ B)$   
*<proof>*

**lemma** *term-ok-mk-all*:  
**assumes**  $wf\text{-theory } \Theta$  **and**  $term\text{-ok}'\ (sig\ \Theta)\ B$  **and**  $typ\text{-of } B = Some\ propT$  **and**  
 $typ\text{-ok } \Theta \tau$

**shows**  $term-ok \Theta (mk-all\ x\ \tau\ B)$   
 $\langle proof \rangle$

**lemma**  $term-ok'-incr-boundvars$ :  
 $term-ok' (sig\ \Theta)\ t \implies term-ok' (sig\ \Theta)\ (incr-boundvars\ lev\ t)$   
 $\langle proof \rangle$

**lemma**  $term-ok'-subst-bv1$ :  
**assumes**  $term-ok' (sig\ \Theta)\ f$  **and**  $term-ok' (sig\ \Theta)\ u$   
**shows**  $term-ok' (sig\ \Theta)\ (subst-bv1\ f\ lev\ u)$   
 $\langle proof \rangle$

**lemma**  $term-ok'-subst-bv$ :  
**assumes**  $term-ok' (sig\ \Theta)\ f$  **and**  $term-ok' (sig\ \Theta)\ u$   
**shows**  $term-ok' (sig\ \Theta)\ (subst-bv\ f\ u)$   
 $\langle proof \rangle$

**lemma**  $term-ok'-betapply$ :  
**assumes**  $term-ok' (sig\ \Theta)\ f$   $term-ok' (sig\ \Theta)\ u$   
**shows**  $term-ok' (sig\ \Theta)\ (f \cdot u)$   
 $\langle proof \rangle$

**lemma**  $term-ok-betapply$ :  
**assumes**  $term-ok\ \Theta\ f$   $term-ok\ \Theta\ u$   
**assumes**  $typ-of\ f = Some\ (uty \rightarrow tty)$   $typ-of\ u = Some\ uty$   
**shows**  $term-ok\ \Theta\ (f \cdot u)$   
 $\langle proof \rangle$

**lemma**  $typ-ok-sig-subst-typ$ :  
**assumes**  $is-std-sig\ \Sigma$  **and**  $typ-ok-sig\ \Sigma\ ty$  **and**  $distinct\ (map\ fst\ insts)$   
**and**  $list-all\ (typ-ok-sig\ \Sigma)\ (map\ snd\ insts)$   
**shows**  $typ-ok-sig\ \Sigma\ (subst-typ\ insts\ ty)$   
 $\langle proof \rangle$

**corollary**  $subst-typ-tinstT$ :  $tinstT\ (subst-typ\ insts\ ty)\ ty$   
 $\langle proof \rangle$

**lemma**  $tsubstT-trans$ :  $tsubstT\ ty\ \rho1 = ty1 \implies tsubstT\ ty1\ \rho2 = ty2$   
 $\implies tsubstT\ ty\ (\lambda idx\ s . case\ \rho1\ idx\ s\ of\ Tv\ idx'\ s' \Rightarrow \rho2\ idx'\ s'$   
 $| Ty\ s\ Ts \Rightarrow Ty\ s\ (map\ (\lambda T . tsubstT\ T\ \rho2)\ Ts)) = ty2$   
 $\langle proof \rangle$

**corollary**  $tinstT-trans$ :  $tinstT\ ty1\ ty \implies tinstT\ ty2\ ty1 \implies tinstT\ ty2\ ty$   
 $\langle proof \rangle$

**lemma**  $term-ok'-subst-typ'$ :  
**assumes**  $is-std-sig\ \Sigma$  **and**  $term-ok'\ \Sigma\ t$  **and**  $distinct\ (map\ fst\ insts)$   
**and**  $list-all\ (typ-ok-sig\ \Sigma)\ (map\ snd\ insts)$



**shows**  $term-ok' \Sigma (subst-typr insts t)$   
 $\langle proof \rangle$

**lemma**

*term-ok'-occs:*

*is-std-sig*  $\Sigma \implies term-ok' \Sigma t \implies occs u t \implies term-ok' \Sigma u$   
 $\langle proof \rangle$

**lemma** *typ-of1-tsubst:*

*typ-of1*  $Ts t = Some ty \implies typ-of1 (map (\lambda T . tsubstT T \varrho) Ts) (tsubst t \varrho) =$   
 $Some (tsubstT ty \varrho)$   
 $\langle proof \rangle$

**corollary** *typ-of1-tsubst-weak:*

**assumes** *typ-of1*  $Ts t = Some ty$

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

**shows**  $tsubstT ty \varrho = ty'$

$\langle proof \rangle$

**lemma** *tsubstT-no-change[simp]:*  $tsubstT T Tv = T$

$\langle proof \rangle$

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

**assumes** *wf-theory*  $\Theta$

**assumes** *term-ok*  $\Theta (mk-eq s t)$

**shows**  $typ-of s = typ-of t$

$\langle proof \rangle$

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

$\langle proof \rangle$

**lemma** *term-okI:*  $term-ok' (sig \Theta) t \implies typ-of t \neq None \implies term-ok \Theta t$

$\langle proof \rangle$

**lemma** *term-okD1:*  $term-ok \Theta t \implies term-ok' (sig \Theta) t$

$\langle proof \rangle$

**lemma** *term-okD2:*  $term-ok \Theta t \implies typ-of t \neq None$

$\langle proof \rangle$

**lemma** *term-ok-imp-typr-ok':* **assumes** *wf-theory*  $\Theta$  *term-ok*  $\Theta t$  **shows** *typr-ok*  $\Theta$   
(*the* (*typ-of*  $t$ ))

$\langle proof \rangle$

**lemma** *term-ok-mk-eqI:*

**assumes** *wf-theory*  $\Theta$  *term-ok*  $\Theta s$  *term-ok*  $\Theta t$   $typ-of s = typ-of t$

**shows** *term-ok*  $\Theta (mk-eq s t)$

$\langle proof \rangle$

**lemma** *typ-of1-decr'*:  $\neg \text{loose-bvar1 } t \ 0 \implies \text{typ-of1 } (T\#Ts) \ t = \text{Some } \tau \implies \text{typ-of1 } Ts \ (\text{decr } 0 \ t) = \text{Some } \tau$   
 ⟨proof⟩

**lemma** *typ-of1-eta-red-step-pre*:  $\neg \text{loose-bvar1 } t \ 0 \implies \text{typ-of1 } Ts \ (\text{Abs } \tau \ (t \ \$ \ Bv \ 0)) = \text{Some } (\tau \rightarrow \tau') \implies \text{typ-of1 } Ts \ (\text{decr } 0 \ t) = \text{Some } (\tau \rightarrow \tau')$   
 ⟨proof⟩

**lemma** *typ-of1-eta-red-step*:  $\neg \text{is-dependent } t \implies \text{typ-of } (\text{Abs } \tau \ (t \ \$ \ Bv \ 0)) = \text{Some } (\tau \rightarrow \tau') \implies \text{typ-of } (\text{decr } 0 \ t) = \text{Some } (\tau \rightarrow \tau')$   
 ⟨proof⟩

**lemma** *distinct-add-vars'*:  $\text{distinct } acc \implies \text{distinct } (\text{add-vars}' \ t \ acc)$   
 ⟨proof⟩

**lemma** *distinct-add-tvarsT'*:  $\text{distinct } acc \implies \text{distinct } (\text{add-tvars}T' \ T \ acc)$   
 ⟨proof⟩

**lemma** *distinct-add-tvars'*:  $\text{distinct } acc \implies \text{distinct } (\text{add-tvars}' \ t \ acc)$   
 ⟨proof⟩

**lemma** *proved-terms-well-formed-pre*:  $\Theta, \Gamma \vdash p \implies \text{typ-of } p = \text{Some } \text{prop}T \ \wedge \ \text{term-ok } \Theta \ p$   
 ⟨proof⟩

**corollary** *proved-terms-well-formed*:  
 assumes  $\Theta, \Gamma \vdash p$   
 shows  $\text{typ-of } p = \text{Some } \text{prop}T \ \text{term-ok } \Theta \ p$   
 ⟨proof⟩

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

**lemma** *term-ok-var[simp]*:  $\text{term-ok } \Theta \ (Fv \ \text{idn } \tau) = \text{typ-ok } \Theta \ \tau$   
 ⟨proof⟩

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

**lemma** *is-closed-Fv[simp]*:  $\text{is-closed } (Fv \ \text{idn } \tau)$  ⟨proof⟩

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

*<proof>*

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

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

*<proof>*

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

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

*<proof>*

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

*<proof>*

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

*<proof>*

**lemma** *forall-intro-vars*:

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

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

*<proof>*

**lemma** *mk-all-list'-preserves-term-ok-typ-of*:

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

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

*<proof>*

**corollary** *forall-intro-vars-preserves-term-ok-typ-of*:

**assumes** *wf-theory*  $\Theta$  *term-ok*  $\Theta$  *B typ-of*  $B = \text{Some propT}$

**shows**  $\text{term-ok } \Theta (\text{forall-intro-vars } B Hs) \wedge \text{typ-of } (\text{forall-intro-vars } B Hs) = \text{Some propT}$

*<proof>*

**lemma** *bind-fv-remove-var-from-fv*:  $\text{fv } (\text{bind-fv } (idn, \tau) t) = \text{fv } t - \{(idn, \tau)\}$

*<proof>*

**lemma** *forall-intro-vars-remove-fv[simp]*:  $\text{fv } (\text{forall-intro-vars } t []) = \{\}$

*<proof>*

**lemma** *term-ok-mk-all-list*:

**assumes** *wf-theory*  $\Theta$

**assumes** *term-ok*  $\Theta$  *B*

**assumes** *typ-of*  $B = \text{Some propT}$

**assumes**  $\forall (idn, \tau) \in \text{set } l . \text{typ-ok } \Theta \tau$

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

*<proof>*

**lemma** *tvs-bind-fv2*:  $\text{tvs } (\text{bind-fv2 } (v, T) \text{ lev } t) \cup \text{tvsT } T = \text{tvs } t \cup \text{tvsT } T$

$\langle \text{proof} \rangle$   
**lemma** *tvs-bind-fv*:  $tvs (\text{bind-fv } (v, T) t) \cup tvsT T = tvs t \cup tvsT T$   
 $\langle \text{proof} \rangle$

**lemma** *tvs-mk-all'*:  $tvs (\text{mk-all idn ty } B) = tvs B \cup tvsT ty$   
 $\langle \text{proof} \rangle$

**lemma** *tvs-mk-all-list*:  
 $tvs (\text{mk-all-list vs } B) = tvs B \cup tvsT\text{-Set } (\text{snd } \text{' set vs})$   
 $\langle \text{proof} \rangle$

**lemma** *tvs-occs*:  $\text{occs } v t \implies tvs v \subseteq tvs t$   
 $\langle \text{proof} \rangle$

**lemma** *tvs-forall-intro-vars*:  $tvs (\text{forall-intro-vars } B Hs) = tvs B$   
 $\langle \text{proof} \rangle$

**lemma** *strip-all-single-var*  $B = \text{Some } \tau \implies \text{strip-all-single-body } B \neq B$   
 $\langle \text{proof} \rangle$

**lemma** *strip-all-body-unchanged-iff-strip-all-single-body-unchanged*:  
 $\text{strip-all-body } B = B \iff \text{strip-all-single-body } B = B$   
 $\langle \text{proof} \rangle$

**lemma** *strip-all-body-unchanged-imp-strip-all-vars-no*:  
**assumes**  $\text{strip-all-body } B = B$   
**shows**  $\text{strip-all-vars } B = []$   
 $\langle \text{proof} \rangle$

**lemma** *strip-all-body-unchanged-imp-strip-all-single-body-unchanged*:  
 $\text{strip-all-body } B = B \implies \text{strip-all-single-body } B = B$   
 $\langle \text{proof} \rangle$

**lemma** *strip-all-single-body-unchanged-imp-strip-all-body-unchanged*:  
 $\text{strip-all-single-body } B = B \implies \text{strip-all-body } B = B$   
 $\langle \text{proof} \rangle$

**lemma** *strip-all-single-var-imp-strip-all-body-single-unchanged*:  
 $\text{strip-all-single-var } B = \text{None} \implies \text{strip-all-single-body } B = B$   
 $\langle \text{proof} \rangle$

**lemma** *strip-all-single-form*:  $\text{strip-all-single-var } B = \text{Some } \tau$   
 $\implies Ct STR \text{'Pure.all'' } ((\tau \rightarrow \text{prop } T) \rightarrow \text{prop } T) \$ Abs \tau (\text{strip-all-single-body } B) = B$   
 $\langle \text{proof} \rangle$

**lemma** *proves-strip-all-single*:  
**assumes**  $\Theta, \Gamma \vdash B \text{ strip-all-single-var } B = \text{Some } \tau$   
 $\text{typ-of } t = \text{Some } \tau \text{ term-ok } \Theta t$

**shows**  $\Theta, \Gamma \vdash \text{subst-bv } t \text{ (strip-all-single-body } B)$   
*<proof>*

**corollary** *proves-strip-all-single-Fv:*

**assumes**  $\Theta, \Gamma \vdash B \text{ strip-all-single-var } B = \text{Some } \tau$   
**shows**  $\Theta, \Gamma \vdash \text{subst-bv } (Fv \ x \ \tau) \text{ (strip-all-single-body } B)$   
*<proof>*

**lemma** *strip-all-vars-no-strip-all-body-unchanged[simp]:*

$\text{strip-all-vars } B = [] \implies \text{strip-all-body } B = B$   
*<proof>*

**lemma** *strip-all-vars*  $B = (\tau s@[ \tau ]) \implies \text{strip-all-body } B$

$= \text{strip-all-single-body } (Ct \ STR \ "Pure.all" \ ((\tau \rightarrow \text{prop}T) \rightarrow \text{prop}T) \ \$ \ Abs \ \tau$   
*(strip-all-body } B)*  
*<proof>*

**lemma** *strip-all-vars-incr-bv:*  $\text{strip-all-vars } (\text{incr-bv } inc \ lev \ t) = \text{strip-all-vars } t$   
*<proof>*

**lemma** *strip-all-vars-incr-boundvars:*  $\text{strip-all-vars } (\text{incr-boundvars } inc \ t) = \text{strip-all-vars } t$   
*<proof>*

**lemma** *strip-all-vars-subst-bv1-Fv:*

$\text{strip-all-vars } (\text{subst-bv1 } B \ lev \ (Fv \ x \ \tau)) = \text{strip-all-vars } B$   
*<proof>*

**lemma** *strip-all-vars-subst-bv-Fv:*

$\text{strip-all-vars } (\text{subst-bv } (Fv \ x \ \tau) \ B) = \text{strip-all-vars } B$   
*<proof>*

**lemma** *strip-all-single-var*  $B = \text{Some } \tau$

$\implies \text{strip-all-vars } (\text{subst-bv } (Fv \ x \ \tau) \text{ (strip-all-single-body } B)) = tl \ (\text{strip-all-vars } B)$   
*<proof>*

**corollary** *proves-strip-all-vars-Fv:*

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

**lemma** *trivial-pre-depr:*  $\text{term-ok } \Theta \ c \implies \text{typ-of } c = \text{Some } \text{prop}T \implies \Theta, \{c\} \vdash c$   
*<proof>*

**lemma** *trivial-pre:*

**assumes** *wf-theory*  $\Theta \ \text{term-ok } \Theta \ c \ \text{typ-of } c = \text{Some } \text{prop}T$   
**shows**  $\Theta, \{\} \vdash c \mapsto c$

$\langle proof \rangle$

**lemma** *inst-var*:

**assumes** *wf-theory*: *wf-theory*  $\Theta$

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

**assumes** *a-ok*: *term-ok*  $\Theta$  *a*

**assumes** *typ-a*: *typ-of* *a* = *Some*  $\tau$

**assumes** *free*:  $(x, \tau) \notin FV \Gamma$

**shows**  $\Theta, \Gamma \vdash \text{subst-term } [((x, \tau), a)] B$

$\langle proof \rangle$

**lemma** *subst-term-single-no-change*[*simp*]:

**assumes** *nvar*:  $(x, \tau) \notin fv B$

**shows**  $\text{subst-term } [((x, \tau), t)] B = B$

$\langle proof \rangle$

**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**  $fv (\text{subst-term } [((x, \tau), t)] B) = fv B - \{(x, \tau)\} \cup fv t$

$\langle proof \rangle$

**lemma** *inst-vars-pre*:

**assumes** *wf-theory*: *wf-theory*  $\Theta$

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

**assumes** *vars-ok*: *list-all* (*term-ok*  $\Theta$ ) (*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* = *Some*  $\tau$

**assumes** *distinct*: *distinct* (*map fst insts*)

**assumes** *no-overlap*:  $\bigwedge x . x \in (\bigcup t \in \text{snd } ' (set \text{ insts}) . fv t) \implies x \notin \text{fst } ' (set \text{ insts})$

**shows**  $\Theta, \Gamma \vdash \text{fold } (\lambda \text{single} . \text{subst-term } [\text{single}]) \text{ insts } B$

$\langle proof \rangle$

**lemma** *subterm-term-ok'*:

*is-std-sig*  $\Sigma \implies \text{term-ok}' \Sigma t \implies \text{is-closed } st \implies \text{occs } st t \implies \text{term-ok}' \Sigma st$

$\langle proof \rangle$

**lemma** *infinite-fv-UNIV*: *infinite* (*UNIV* :: (*indexname*  $\times$  *typ*) *set*)

$\langle proof \rangle$

**lemma** *implies-intro'-pre*:

**assumes** *wf-theory*  $\Theta$ ,  $\Gamma \vdash B$  *term-ok*  $\Theta$   $A$  *typ-of*  $A = \text{Some prop} T A \notin \Gamma$   
**shows**  $\Theta, \Gamma \vdash A \mapsto B$   
 $\langle \text{proof} \rangle$

**lemma** *implies-intro'-pre2*:

**assumes** *wf-theory*  $\Theta$ ,  $\Gamma \vdash B$  *term-ok*  $\Theta$   $A$  *typ-of*  $A = \text{Some prop} T A \in \Gamma$   
**shows**  $\Theta, \Gamma \vdash A \mapsto B$   
 $\langle \text{proof} \rangle$

**lemma** *subst-term-preserves-typ-of1[simp]*:

*typ-of1*  $Ts$  (*subst-term*  $[(x, \tau), Fv\ y\ \tau]$   $t$ ) = *typ-of1*  $Ts\ t$   
 $\langle \text{proof} \rangle$

**lemma** *subst-term-preserves-typ-of[simp]*:

*typ-of* (*subst-term*  $[(x, \tau), Fv\ y\ \tau]$   $t$ ) = *typ-of*  $t$   
 $\langle \text{proof} \rangle$

**lemma** *subst-term-preserves-term-ok'[simp]*:

*term-ok'*  $\Sigma$  (*subst-term*  $[(x, \tau), Fv\ y\ \tau]$   $t$ )  $\longleftrightarrow$  *term-ok'*  $\Sigma\ t$   
 $\langle \text{proof} \rangle$

**lemma** *subst-term-preserves-term-ok[simp]*:

*term-ok*  $\Theta$  (*subst-term*  $[(x, \tau), Fv\ y\ \tau]$   $A$ )  $\longleftrightarrow$  *term-ok*  $\Theta\ A$   
 $\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

**lemma** *subst-term-fv*:  $fv$  (*subst-term*  $[(x, \tau), Fv\ y\ \tau]$   $t$ )

= (if  $(x, \tau) \in fv\ t$  then *insert*  $(y, \tau)$  else *id*) ( $fv\ t - \{(x, \tau)\}$ )

$\langle \text{proof} \rangle$

**lemma** *rename-free*:

**assumes** *wf-theory*: *wf-theory*  $\Theta$

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

**assumes** *free*:  $(x, \tau) \notin FV\ \Gamma$

**shows**  $\Theta, \Gamma \vdash$  *subst-term*  $[(x, \tau), Fv\ y\ \tau]$   $B$

$\langle \text{proof} \rangle$

**lemma** *tvs-subst-term-single[simp]*:  $tvs$  (*subst-term*  $[(x, \tau), Fv\ y\ \tau]$   $A$ ) =  $tvs\ A$

$\langle \text{proof} \rangle$

**lemma** *weaken-proves'*:  $\Theta, \Gamma \vdash B \implies$  *term-ok*  $\Theta\ A \implies$  *typ-of*  $A = \text{Some prop} T$

$\implies A \notin \Gamma$

$\implies$  *finite*  $\Gamma$

$\implies \Theta, \text{insert } A\ \Gamma \vdash B$

$\langle \text{proof} \rangle$

**corollary** *weaken-proves*:  $\Theta, \Gamma \vdash B \implies \text{term-ok } \Theta A \implies \text{typ-of } A = \text{Some prop } T$   
 $\implies \text{finite } \Gamma$   
 $\implies \Theta, \text{insert } A \Gamma \vdash B$   
 $\langle \text{proof} \rangle$

**lemma** *weaken-proves-set*:  $\text{finite } \Gamma \implies \Theta, \Gamma \vdash B \implies \forall A \in \Gamma \implies \text{term-ok } \Theta A \implies$   
 $\forall A \in \Gamma \implies \text{typ-of } A = \text{Some prop } T$   
 $\implies \text{finite } \Gamma$   
 $\implies \Theta, \Gamma \cup \Gamma \implies B$   
 $\langle \text{proof} \rangle$

**lemma** *no-tvsT-imp-subst-typ-unchanged*:  $\text{tvs } T = \text{empty} \implies \text{subst-typ insts } T$   
 $= T$   
 $\langle \text{proof} \rangle$

**lemma** *subst-typ-fv*:  
**shows**  $\text{apsnd } (\text{subst-typ insts}) \text{ `fv } B = \text{fv } (\text{subst-typ' insts } B)$   
 $\langle \text{proof} \rangle$

**lemma** *subst-typ-fv-point*:  
**assumes**  $(x, \tau) \in \text{fv } B$   
**shows**  $(x, \text{subst-typ insts } \tau) \in \text{fv } (\text{subst-typ' insts } B)$   
 $\langle \text{proof} \rangle$

**lemma** *subst-typ-typ-ok*:  
**assumes**  $\text{typ-ok-sig } \Sigma \tau$   
**assumes**  $\text{list-all } (\text{typ-ok-sig } \Sigma) (\text{map snd insts})$   
**shows**  $\text{typ-ok-sig } \Sigma (\text{subst-typ insts } \tau)$   
 $\langle \text{proof} \rangle$

**lemma** *subst-typ-comp-single-left*:  $\text{subst-typ } [\text{single}] (\text{subst-typ insts } T)$   
 $= \text{subst-typ } (\text{map } (\text{apsnd } (\text{subst-typ } [\text{single}])) \text{ insts} @ [\text{single}]) T$   
 $\langle \text{proof} \rangle$

**lemma** *subst-typ-comp-single-left-stronger*:  $\text{subst-typ } [\text{single}] (\text{subst-typ insts } T)$   
 $= \text{subst-typ } (\text{map } (\text{apsnd } (\text{subst-typ } [\text{single}])) \text{ insts}$   
 $@ (\text{if fst single} \in \text{set } (\text{map fst insts}) \text{ then } [] \text{ else } [\text{single}])) T$   
 $\langle \text{proof} \rangle$

**lemma** *subst-typ'-comp-single-left*:  $\text{subst-typ' } [\text{single}] (\text{subst-typ' insts } t)$   
 $= \text{subst-typ' } (\text{map } (\text{apsnd } (\text{subst-typ } [\text{single}])) \text{ insts} @ [\text{single}]) t$   
 $\langle \text{proof} \rangle$

**lemma** *subst-typ'-comp-single-left-stronger*:  $\text{subst-typ' } [\text{single}] (\text{subst-typ' insts } t)$   
 $= \text{subst-typ' } (\text{map } (\text{apsnd } (\text{subst-typ } [\text{single}])) \text{ insts}$   
 $@ (\text{if fst single} \in \text{set } (\text{map fst insts}) \text{ then } [] \text{ else } [\text{single}])) t$   
 $\langle \text{proof} \rangle$



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

**assumes** *wf-theory*  $\Theta$   
**assumes** *typ-ok*  $\Theta$   $T$   
**assumes** *list-all* (*typ-ok*  $\Theta$ ) (*map snd insts*)  
**shows** *typ-ok*  $\Theta$  (*subst-typ insts*  $T$ )

*<proof>*

**lemma** *typ-ok-Ty[simp]*: *typ-ok*  $\Theta$  ( $Ty$   $n$   $Ts$ )  $\implies$  *list-all* (*typ-ok*  $\Theta$ )  $Ts$

*<proof>*

**lemma** *typ-ok-sig-Ty[simp]*: *typ-ok-sig*  $\Sigma$  ( $Ty$   $n$   $Ts$ )  $\implies$  *list-all* (*typ-ok-sig*  $\Sigma$ )  $Ts$

*<proof>*

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

*<proof>*

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

*b <proof>*

**lemma** *class-les-mgd*:

**assumes** *wf-osig*  $oss$   
**assumes** *tcsigs*  $oss$  *type* = *Some*  $mgd$   
**assumes** *mgd*  $C' =$  *Some*  $Ss'$   
**assumes** *class-les* (*subclass*  $oss$ )  $C' C$   
**shows** *mgd*  $C \neq$  *None*

*<proof>*

**lemma** *has-sort-sort-leq-osig*:

**assumes** *wf-osig* ( $sub, tcs$ ) *has-sort* ( $sub, tcs$ )  $T S$  *sort-leq*  $sub S S'$   
**shows** *has-sort* ( $sub, tcs$ )  $T S'$

*<proof>*

**lemma** *has-sort-sort-leq*: *wf-theory*  $\Theta \implies$  *has-sort* (*osig* (*sig*  $\Theta$ ))  $T S$

$\implies$  *sort-leq* (*subclass* (*osig* (*sig*  $\Theta$ )))  $S S'$

$\implies$  *has-sort* (*osig* (*sig*  $\Theta$ ))  $T S'$

*<proof>*

**lemma** *subst-typ-preserves-has-sort*:

**assumes** *wf-theory*  $\Theta$   
**assumes** *has-sort* (*osig* (*sig*  $\Theta$ ))  $T S$   
**assumes** *list-all* ( $\lambda(idn, S), T$ ). *has-sort* (*osig* (*sig*  $\Theta$ ))  $T S$  *insts*  
**shows** *has-sort* (*osig* (*sig*  $\Theta$ )) (*subst-typ insts*  $T$ )  $S$

*<proof>*

**lemma** *subst-typ-preserves-Some-typ-of1*:

**assumes** *typ-of1*  $Ts$   $t =$  *Some*  $T$   
**shows** *typ-of1* (*map* (*subst-typ insts*)  $Ts$ ) (*subst-typ' insts*  $t$ )  
 $=$  *Some* (*subst-typ insts*  $T$ )

*<proof>*

**corollary** *subst-typ-preserves-Some-typ-of:*

**assumes**  $\text{typ-of } t = \text{Some } T$   
**shows**  $\text{typ-of } (\text{subst-typ}' \text{ insts } t)$   
     $= \text{Some } (\text{subst-typ } \text{ insts } T)$   
*<proof>*

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

$\text{subst-typ}' \text{ insts } (\text{incr-bv } \text{inc } \text{lev } t) = \text{incr-bv } \text{inc } \text{lev } (\text{subst-typ}' \text{ insts } t)$   
*<proof>*

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

$\text{subst-typ}' \text{ insts } (\text{incr-boundvars } \text{lev } t) = \text{incr-boundvars } \text{lev } (\text{subst-typ}' \text{ insts } t)$   
*<proof>*

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

$= \text{subst-bv1 } (\text{subst-typ}' \text{ insts } t) \ n \ (\text{subst-typ}' \text{ insts } u)$   
*<proof>*

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

$= \text{subst-bv } (\text{subst-typ}' \text{ insts } t) \ (\text{subst-typ}' \text{ insts } u)$   
*<proof>*

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

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

**lemma** *subst-typ'-no-tvs-unchanged:*

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

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

**assumes** *wf-theory*  $\Theta$   
**assumes** *inst-ok*  $\Theta \ \text{insts}$   
**assumes** *term-ok'*  $(\text{sig } \Theta) \ t$   
**shows** *term-ok'*  $(\text{sig } \Theta) \ (\text{subst-typ}' \text{ insts } t)$   
*<proof>*

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

**assumes** *wf-theory*  $\Theta$   
**assumes** *inst-ok*  $\Theta \ \text{insts}$   
**assumes** *term-ok*  $\Theta \ t$   
**shows** *term-ok*  $\Theta \ (\text{subst-typ}' \text{ insts } t)$   
*<proof>*

**lemma** *subst-typ-rename-vars-cancel:*

**assumes**  $y \notin \text{fst } ' \text{tvs } T \ T$   
**shows**  $\text{subst-typ } [((y, S), \text{Tv } x \ S)] \ (\text{subst-typ } [((x, S), \text{Tv } y \ S)] \ T) = T$

$\langle \text{proof} \rangle$

**lemma** *subst-typ'-rename-tvars-cancel*:

**assumes**  $y \notin \text{fst } ' \text{tvs } t$  **assumes**  $y \notin \text{fst } ' \text{tvsT } \tau$

**shows**  $\text{subst-typ}' [(y, S), \text{Tv } x S] ((\text{bind-fv2 } (x, \text{subst-typ}' [(x, S), \text{Tv } y S]) \tau))$   
 $\text{lev } (\text{subst-typ}' [(x, S), \text{Tv } y S]) t)$

$= \text{bind-fv2 } (x, \tau) \text{ lev } t$

$\langle \text{proof} \rangle$

**lemma** *bind-fv2-renamed-var*:

**assumes**  $y \notin \text{fst } ' \text{fv } t$

**shows**  $\text{bind-fv2 } (y, \tau) i (\text{subst-term } [(x, \tau), \text{Fv } y \tau]) t)$

$= \text{bind-fv2 } (x, \tau) i t$

$\langle \text{proof} \rangle$

**lemma** *bind-fv-renamed-var*:

**assumes**  $y \notin \text{fst } ' \text{fv } t$

**shows**  $\text{bind-fv } (y, \tau) (\text{subst-term } [(x, \tau), \text{Fv } y \tau]) t)$

$= \text{bind-fv } (x, \tau) t$

$\langle \text{proof} \rangle$

**lemma** *subst-typ'-rename-tvar-bind-fv2*:

**assumes**  $y \notin \text{fst } ' \text{fv } t$

**assumes**  $(b, S) \notin \text{tvs } t$

**assumes**  $(b, S) \notin \text{tvsT } \tau$

**shows**  $\text{bind-fv2 } (y, \text{subst-typ}' [(a, S), \text{Tv } b S]) \tau) i$   
 $(\text{subst-typ}' [(a, S), \text{Tv } b S]) (\text{subst-term } [(x, \tau), \text{Fv } y \tau]) t)$   
 $= \text{subst-typ}' [(a, S), \text{Tv } b S] (\text{bind-fv2 } (x, \tau) i t)$

$\langle \text{proof} \rangle$

**lemma** *subst-typ'-rename-tvar-bind-fv*:

**assumes**  $y \notin \text{fst } ' \text{fv } t$

**assumes**  $(b, S) \notin \text{tvs } t$

**assumes**  $(b, S) \notin \text{tvsT } \tau$

**shows**  $\text{bind-fv } (y, \text{subst-typ}' [(a, S), \text{Tv } b S]) \tau)$   
 $(\text{subst-typ}' [(a, S), \text{Tv } b S]) (\text{subst-term } [(x, \tau), \text{Fv } y \tau]) t)$   
 $= \text{subst-typ}' [(a, S), \text{Tv } b S] (\text{bind-fv } (x, \tau) t)$

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

**lemma** *subst-typ'-rename-tvar-preserves-eq*:

$(y, S) \notin \text{tvs}T \ T \implies (y, S) \notin \text{tvs}T \ \tau \implies$   
 $\text{subst-typ} [((x, S), \text{Tv } y \ S)] \ T = \text{subst-typ} [((x, S), \text{Tv } y \ S)] \ \tau \implies T = \tau$   
 <proof>

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

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

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

$(y, S) \in \text{tvs}T \ \tau \implies (y, S) \notin \text{tvs} \ t \implies (a, \tau) \notin \text{fv} \ t$   
 <proof>

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

$\text{finite } \Gamma \implies (y, S) \in \text{tvs}T \ \tau \implies (y, S) \notin \text{tvs-Set } \Gamma \implies (a, \tau) \notin \text{FV } \Gamma$   
 <proof>

**lemma** *inst-var-multiple:*

**assumes** *wf-theory:*  $\text{wf-theory } \Theta$   
**assumes**  $B: \Theta, \Gamma \vdash B$   
**assumes** *vars:*  $\forall (x, \tau) \in \text{fst} \ ' \ \text{set insts} \ . \ \text{term-ok } \Theta \ (\text{Fv } x \ \tau)$   
**assumes** *a-ok:*  $\forall a \in \text{snd} \ ' \ \text{set insts} \ . \ \text{term-ok } \Theta \ a$   
**assumes** *typ-a:*  $\forall ((-, \tau), a) \in \text{set insts} \ . \ \text{typ-of } a = \text{Some } \tau$   
**assumes** *free:*  $\forall (v, -) \in \text{set insts} \ . \ v \notin \text{FV } \Gamma$   
**assumes** *distinct:*  $\text{distinct} \ (\text{map } \text{fst } \text{insts})$   
**assumes** *finite:*  $\text{finite } \Gamma$   
**shows**  $\Theta, \Gamma \vdash \text{subst-term insts } B$   
 <proof>

**lemma** *term-ok-eta-red-step:*

$\neg \text{is-dependent } t \implies \text{term-ok } \Theta \ (\text{Abs } T \ (t \ \$ \ Bv \ 0)) \implies \text{term-ok } \Theta \ (\text{decr } 0 \ t)$   
 <proof>

end

## 11 Derived rules on equality and normalization

**theory** *EqualityProof*

**imports** *Logic*

**begin**

**lemma** *proves-eq-reflexive-pre*:

**assumes** *wf-theory*  $\Theta$   
**assumes** *term-ok*  $\Theta$   $t$   
**shows**  $\Theta, \{\} \vdash \text{mk-eq } t \ t$   
*<proof>*

**lemma** *unsimp-context*:  $\Gamma = \{\} \cup \Gamma$   
*<proof>*

**lemma** *proves-eq-reflexive*:

**assumes** *wf-theory*  $\Theta$   
**assumes** *term-ok*  $\Theta$   $t$   
**assumes** *finite*  $\Gamma \ \forall A \in \Gamma. \text{term-ok } \Theta \ A \ \forall A \in \Gamma. \text{typ-of } A = \text{Some propT}$   
**shows**  $\Theta, \Gamma \vdash \text{mk-eq } t \ t$   
*<proof>*

**lemma** *proves-eq-symmetric-pre*:

**assumes** *wf-theory*  $\Theta$   
**assumes** *term-ok*  $\Theta$   $t$   
**assumes** *term-ok*  $\Theta$   $s$   
**assumes** *typ-of*  $s = \text{typ-of } t$   
**shows**  $\Theta, \{\} \vdash \text{mk-eq } s \ t \ \mapsto \ \text{mk-eq } t \ s$   
*<proof>*

**lemma** *proves-eq-symmetric*:

**assumes** *wf-theory*  $\Theta$   
**assumes** *term-ok*  $\Theta$   $t$   
**assumes** *term-ok*  $\Theta$   $s$   
**assumes** *typ-of*  $s = \text{typ-of } t$   
**assumes** *finite*  $\Gamma \ \forall A \in \Gamma. \text{term-ok } \Theta \ A \ \forall A \in \Gamma. \text{typ-of } A = \text{Some propT}$   
**shows**  $\Theta, \Gamma \vdash \text{mk-eq } s \ t \ \mapsto \ \text{mk-eq } t \ s$   
*<proof>*

**lemma** *proves-eq-symmetric2'*:

**assumes** *wf-theory*  $\Theta$   
**assumes** *term-ok*  $\Theta$   $(\text{mk-eq } s \ t)$   
**assumes** *finite*  $\Gamma \ \forall A \in \Gamma. \text{term-ok } \Theta \ A \ \forall A \in \Gamma. \text{typ-of } A = \text{Some propT}$   
**shows**  $\Theta, \Gamma \vdash \text{mk-eq } s \ t \ \mapsto \ \text{mk-eq } t \ s$   
*<proof>*

**lemma** *proves-eq-symmetric-rule*:

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

*<proof>*

**lemma** *proves-eq-transitive-pre:*

**assumes** *wf-theory*  $\Theta$   
**assumes** *term-ok*  $\Theta$   $s$   
**assumes** *term-ok*  $\Theta$   $t$   
**assumes** *term-ok*  $\Theta$   $u$   
**assumes**  $\text{typ-of } s = \text{typ-of } t \text{ typ-of } t = \text{typ-of } u$   
**shows**  $\Theta, \{\} \vdash \text{mk-eq } s \ t \mapsto \text{mk-eq } t \ u \mapsto \text{mk-eq } s \ u$

*<proof>*

**lemma** *proves-eq-transitive:*

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

*<proof>*

**lemma** *proves-eq-transitive2:*

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

*<proof>*

**lemma** *proves-eq-transitive-rule:*

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

*<proof>*

**lemma** *proves-eq-intr-pre:*

**assumes** *thy: wf-theory*  $\Theta$   
**assumes**  $A: \text{term-ok } \Theta \ A \ \text{typ-of } A = \text{Some propT}$   
**assumes**  $B: \text{term-ok } \Theta \ B \ \text{typ-of } B = \text{Some propT}$   
**shows**  $\Theta, \{\} \vdash (A \mapsto B) \mapsto (B \mapsto A) \mapsto \text{mk-eq } A \ B$

*<proof>*

**lemma** *proves-eq-intr:*

**assumes** *thy: wf-theory*  $\Theta$

**assumes**  $A$ : *term-ok*  $\Theta$   $A$  *typ-of*  $A = \text{Some prop}T$   
**assumes**  $B$ : *term-ok*  $\Theta$   $B$  *typ-of*  $B = \text{Some prop}T$   
**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**  $\Theta, \Gamma \vdash (A \mapsto B) \mapsto (B \mapsto A) \mapsto \text{mk-eq } A B$   
 $\langle \text{proof} \rangle$

**lemma** *proves-eq-intr-rule*:  
**assumes** *thy*: *wf-theory*  $\Theta$   
**assumes**  $A$ : *term-ok*  $\Theta$   $A$  *typ-of*  $A = \text{Some prop}T$   
**assumes**  $B$ : *term-ok*  $\Theta$   $B$  *typ-of*  $B = \text{Some prop}T$   
**assumes**  $\Theta, \Gamma \vdash (A \mapsto B) \Theta, \Gamma \vdash (B \mapsto A)$   
**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**  $\Theta, \Gamma \vdash \text{mk-eq } A B$   
 $\langle \text{proof} \rangle$

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

**lemma** *proves-eq-elim*:  
**assumes** *thy*: *wf-theory*  $\Theta$   
**assumes**  $A$ : *term-ok*  $\Theta$   $A$  *typ-of*  $A = \text{Some prop}T$   
**assumes**  $B$ : *term-ok*  $\Theta$   $B$  *typ-of*  $B = \text{Some prop}T$   
**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**  $\Theta, \Gamma \vdash \text{mk-eq } A B \mapsto A \mapsto B$   
 $\langle \text{proof} \rangle$

**lemma** *proves-eq-elim-rule*:  
**assumes** *thy*: *wf-theory*  $\Theta$   
**assumes**  $A$ : *term-ok*  $\Theta$   $A$  *typ-of*  $A = \text{Some prop}T$   
**assumes**  $B$ : *term-ok*  $\Theta$   $B$  *typ-of*  $B = \text{Some prop}T$   
**assumes**  $\Theta, \Gamma \vdash \text{mk-eq } A B$   
**assumes** *ctxt*: *finite*  $\Gamma \forall A \in \Gamma. \text{term-ok } \Theta A \forall A \in \Gamma. \text{typ-of } A = \text{Some prop}T$   
**shows**  $\Theta, \Gamma \vdash A \mapsto B$   
 $\langle \text{proof} \rangle$

**lemma** *proves-eq-elim2-rule*:  
**assumes** *thy*: *wf-theory*  $\Theta$   
**assumes**  $A$ : *term-ok*  $\Theta$   $A$  *typ-of*  $A = \text{Some prop}T$   
**assumes**  $B$ : *term-ok*  $\Theta$   $B$  *typ-of*  $B = \text{Some prop}T$   
**assumes**  $\Theta, \Gamma \vdash \text{mk-eq } A B$   
**assumes** *ctxt*: *finite*  $\Gamma \forall A \in \Gamma. \text{term-ok } \Theta A \forall A \in \Gamma. \text{typ-of } A = \text{Some prop}T$   
**shows**  $\Theta, \Gamma \vdash B \mapsto A$   
 $\langle \text{proof} \rangle$

**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**  $\Theta, \{\}$   $\vdash$  *mk-eq* *f g*  $\mapsto$  *mk-eq* *x y*  $\mapsto$  *mk-eq* (*f* \$ *x*) (*g* \$ *y*)  
*<proof>*

**lemma** *proves-eq-combination*:

**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*  $\tau$   
**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**  $\Theta, \Gamma \vdash$  *mk-eq* *f g*  $\mapsto$  *mk-eq* *x y*  $\mapsto$  *mk-eq* (*f* \$ *x*) (*g* \$ *y*)  
*<proof>*

**lemma** *proves-eq-combination-rule*:

**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*  $\tau$   
**assumes**  $\Theta, \Gamma \vdash$  *mk-eq* *f g*  $\Theta, \Gamma \vdash$  *mk-eq* *x y*  
**assumes** *ctxt*: *finite*  $\Gamma \forall A \in \Gamma. \text{term-ok } \Theta A \forall A \in \Gamma. \text{typ-of } A = \text{Some propT}$   
**shows**  $\Theta, \Gamma \vdash$  *mk-eq* (*f* \$ *x*) (*g* \$ *y*)  
*<proof>*

**lemma** *proves-eq-combination-rule-better*:

**assumes** *thy*: *wf-theory*  $\Theta$   
**assumes**  $\Theta, \Gamma \vdash$  *mk-eq* *f g*  $\Theta, \Gamma \vdash$  *mk-eq* *x y*  
**assumes** *f*: *typ-of* *f* = *Some* ( $\tau \rightarrow \tau'$ )  
**assumes** *x*: *typ-of* *x* = *Some*  $\tau$   
**assumes** *ctxt*: *finite*  $\Gamma \forall A \in \Gamma. \text{term-ok } \Theta A \forall A \in \Gamma. \text{typ-of } A = \text{Some propT}$   
**shows**  $\Theta, \Gamma \vdash$  *mk-eq* (*f* \$ *x*) (*g* \$ *y*)  
*<proof>*

**lemma** *proves-eq-mp-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** *eq*:  $\Theta, \Gamma \vdash$  *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>*



**lemma** *proves-eq-mp-rule-better:*

**assumes** *thy: wf-theory*  $\Theta$

**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>*

**lemma** *proves-subst-rule:*

**assumes** *thy: wf-theory*  $\Theta$

**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>*

**lemma** *proves-beta-step-rule:*

**assumes** *thy: wf-theory*  $\Theta$

**assumes** *abs:* *term-ok*  $\Theta \ (\text{Abs } T \ t) \ \Theta, \Gamma \vdash (\text{Abs } T \ t) \ \$ \ x$

**assumes** *x:* *term-ok*  $\Theta \ x \ \text{typ-of } x = \text{Some } T$

**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**  $\Theta, \Gamma \vdash \text{subst-bv } x \ t$

*<proof>*

**lemma** *proves-add-param-rule:*

**assumes** *thy: wf-theory*  $\Theta$

**assumes** *ctxt:* *finite*  $\Gamma$

**assumes** *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*  $\Gamma \ \forall A \in \Gamma. \text{term-ok } \Theta \ A \ \forall A \in \Gamma. \text{typ-of } A = \text{Some propT}$

**shows**  $\Theta, \Gamma \vdash (\text{Ct STR "Pure.all" } ((\tau \rightarrow \text{propT}) \rightarrow \text{propT}) \ \$$

$(\text{Abs } \tau \ (\text{mk-eq}' \ \tau' \ (f \ \$ \ \text{Bv } 0) \ (g \ \$ \ \text{Bv } 0))))$

*<proof>*

**lemma** *proves-add-abs-rule:*

**assumes** *thy: wf-theory*  $\Theta$

**assumes** *ctxt:* *finite*  $\Gamma$

**assumes** *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*  $\Gamma \ \forall A \in \Gamma. \text{term-ok } \Theta \ A \ \forall A \in \Gamma. \text{typ-of } A = \text{Some propT}$

**shows**  $\Theta, \Gamma \vdash \text{mk-eq } (\text{Abs } \tau \ (f \ \$ \ \text{Bv } 0)) \ (\text{Abs } \tau \ (g \ \$ \ \text{Bv } 0))$

*<proof>*

**lemma** *proves-inst-bound-rule:*

**assumes** *thy: wf-theory*  $\Theta$

**assumes** *ctxt*:  $\text{finite } \Gamma \forall A \in \Gamma . \text{term-ok } \Theta A \forall A \in \Gamma . \text{typ-of } A = \text{Some propT}$   
**assumes** *eq*:  $\Theta, \Gamma \vdash \text{mk-eq } (\text{Abs } \tau f) (\text{Abs } \tau g) \text{ typ-of } (\text{Abs } \tau f) = \text{Some } (\tau \rightarrow \tau')$   
**assumes** *x*:  $\text{term-ok } \Theta x \text{ typ-of } x = \text{Some } \tau$   
**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**  $\Theta, \Gamma \vdash \text{mk-eq } (\text{subst-bv } x f) (\text{subst-bv } x g)$   
*<proof>*

**lemma** *proves-descend-abs-rule*:

**assumes** *thy*: *wf-theory*  $\Theta$   
**assumes** *eq*:  $\Theta, \Gamma \vdash \text{mk-eq } (\text{Abs } \tau' (\text{bind-fv } (x, \tau') s)) (\text{Abs } \tau' (\text{bind-fv } (x, \tau') t))$   
*is-closed s is-closed t*  
**assumes** *x*:  $(x, \tau') \notin \text{FV } \Gamma \text{ typ-ok } \Theta \tau'$   
**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**  $\Theta, \Gamma \vdash \text{mk-eq } s t$   
*<proof>*

**lemma** *obtain-fresh-variable*:

**assumes** *finite*  $\Gamma$   
**obtains** *x* **where**  $(x, \tau) \notin \text{fv } t \cup \text{FV } \Gamma$   
*<proof>*

**lemma** *obtain-fresh-variable'*:

**assumes** *finite*  $\Gamma$   
**obtains** *x* **where**  $(x, \tau) \notin \text{fv } t \cup \text{fv } u \cup \text{FV } \Gamma$   
*<proof>*

**lemma** *proves-eq-abstract-rule-pre*:

**assumes** *thy*: *wf-theory*  $\Theta$   
**assumes** *A*:  $\text{term-ok } \Theta f \text{ typ-of } f = \text{Some } (\tau \rightarrow \tau')$   
**assumes** *B*:  $\text{term-ok } \Theta g \text{ typ-of } g = \text{Some } (\tau \rightarrow \tau')$   
**shows**  $\Theta, \{\} \vdash (\text{Ct STR "Pure.all" } ((\tau \rightarrow \text{propT}) \rightarrow \text{propT}) \$ \text{Abs } \tau (\text{mk-eq}' \tau' (f \$ \text{Bv } 0) (g \$ \text{Bv } 0)))$   
 $\mapsto \text{mk-eq } (\text{Abs } \tau (f \$ \text{Bv } 0)) (\text{Abs } \tau (g \$ \text{Bv } 0))$   
*<proof>*

**lemma** *proves-eq-abstract-rule*:

**assumes** *thy*: *wf-theory*  $\Theta$   
**assumes** *A*:  $\text{term-ok } \Theta f \text{ typ-of } f = \text{Some } (\tau \rightarrow \tau')$   
**assumes** *B*:  $\text{term-ok } \Theta g \text{ typ-of } g = \text{Some } (\tau \rightarrow \tau')$   
**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**  $\Theta, \Gamma \vdash (\text{Ct STR "Pure.all" } ((\tau \rightarrow \text{propT}) \rightarrow \text{propT}) \$ \text{Abs } \tau (\text{mk-eq}' \tau' (f \$ \text{Bv } 0) (g \$ \text{Bv } 0)))$   
 $\mapsto \text{mk-eq } (\text{Abs } \tau (f \$ \text{Bv } 0)) (\text{Abs } \tau (g \$ \text{Bv } 0))$   
*<proof>*

**lemma** *proves-eq-abstract-rule-rule*:

**assumes** *thy*: *wf-theory*  $\Theta$

**assumes**  $A$ :  $\text{term-ok } \Theta \ f \ \text{typ-of } f = \text{Some } (\tau \rightarrow \tau')$   
**assumes**  $B$ :  $\text{term-ok } \Theta \ g \ \text{typ-of } g = \text{Some } (\tau \rightarrow \tau')$   
**assumes**  $\Theta, \Gamma \vdash (\text{Ct STR "Pure.all" } ((\tau \rightarrow \text{propT}) \rightarrow \text{propT}) \ \$ \ \text{Abs } \tau \ (mk\text{-eq}' \ \tau' \ (f \ \$ \ Bv \ 0) \ (g \ \$ \ Bv \ 0)))$   
**assumes**  $\text{ctxt}$ :  $\text{finite } \Gamma \ \forall A \in \Gamma. \ \text{term-ok } \Theta \ A \ \forall A \in \Gamma. \ \text{typ-of } A = \text{Some propT}$   
**shows**  $\Theta, \Gamma \vdash mk\text{-eq } (\text{Abs } \tau \ (f \ \$ \ Bv \ 0)) \ (\text{Abs } \tau \ (g \ \$ \ Bv \ 0))$   
 $\langle \text{proof} \rangle$

**lemma** *proves-eq-ext-rule*:

**assumes**  $\text{thy}$ :  $\text{wf-theory } \Theta$   
**assumes**  $f$ :  $\text{term-ok } \Theta \ f \ \text{typ-of } f = \text{Some } (\tau \rightarrow \tau')$   
**assumes**  $g$ :  $\text{term-ok } \Theta \ g \ \text{typ-of } g = \text{Some } (\tau \rightarrow \tau')$   
**assumes**  $\text{prem}$ :  $\Theta, \Gamma \vdash \text{Ct STR "Pure.all" } ((\tau \rightarrow \text{propT}) \rightarrow \text{propT}) \ \$ \ \text{Abs } \tau \ (mk\text{-eq}' \ \tau' \ (f \ \$ \ Bv \ 0) \ (g \ \$ \ Bv \ 0))$   
**assumes**  $\text{ctxt}$ :  $\text{finite } \Gamma \ \forall A \in \Gamma. \ \text{term-ok } \Theta \ A \ \forall A \in \Gamma. \ \text{typ-of } A = \text{Some propT}$   
**shows**  $\Theta, \Gamma \vdash mk\text{-eq } f \ g$   
 $\langle \text{proof} \rangle$

**lemma** *bind-fv2-idem[simp]*:

$\text{bind-fv2 } (x, \tau) \ \text{lev1 } (\text{bind-fv2 } (x, \tau) \ \text{lev2 } t) = \text{bind-fv2 } (x, \tau) \ \text{lev2 } t$   
 $\langle \text{proof} \rangle$

**corollary** *bind-fv-idem[simp]*:

$\text{bind-fv } (x, \tau) \ (\text{bind-fv } (x, \tau) \ t) = \text{bind-fv } (x, \tau) \ t$   
 $\langle \text{proof} \rangle$

**corollary** *bind-fv-Abs-fv[simp]*:  $\text{bind-fv } (x, \tau) \ (\text{Abs-fv } x \ \tau \ t) = \text{Abs-fv } x \ \tau \ t$   
 $\langle \text{proof} \rangle$

**lemma** *bind-fv2*  $(x, \tau) \ \text{lev} \ (mk\text{-eq}' \ \tau' \ s \ t) = mk\text{-eq}' \ \tau' \ (\text{bind-fv2 } (x, \tau) \ \text{lev} \ s) \ (\text{bind-fv2 } (x, \tau) \ \text{lev} \ t)$   
 $\langle \text{proof} \rangle$

**lemma** *bind-fv*  $(x, \tau) \ (mk\text{-eq}' \ \tau' \ s \ t) = mk\text{-eq}' \ \tau' \ (\text{bind-fv } (x, \tau) \ s) \ (\text{bind-fv } (x, \tau) \ t)$   
 $\langle \text{proof} \rangle$

**lemma** *term-ok-Abs-fvI*:  $\text{term-ok } \Theta \ s \implies \text{typ-ok } \Theta \ \tau \implies \text{term-ok } \Theta \ (\text{Abs-fv } x \ \tau \ s)$   
 $\langle \text{proof} \rangle$

**lemma** *proves-eq-abstract-rule-derived-rule*:

**assumes**  $\text{thy}$ :  $\text{wf-theory } \Theta$   
**assumes**  $x$ :  $(x, \tau) \notin \text{FV } \Gamma \ \text{typ-ok } \Theta \ \tau$   
**assumes**  $\text{ctxt}$ :  $\text{finite } \Gamma \ \forall A \in \Gamma. \ \text{term-ok } \Theta \ A \ \forall A \in \Gamma. \ \text{typ-of } A = \text{Some propT}$   
**assumes**  $\text{eq}$ :  $\Theta, \Gamma \vdash mk\text{-eq } s \ t$   
**shows**  $\Theta, \Gamma \vdash mk\text{-eq } (\text{Abs } \tau \ (\text{bind-fv } (x, \tau) \ s)) \ (\text{Abs } \tau \ (\text{bind-fv } (x, \tau) \ t))$   
 $\langle \text{proof} \rangle$

**lemma** *proves-descend-abs-rule-iff*:

**assumes**  $\text{thy}$ :  $\text{wf-theory } \Theta$   
**assumes**  $\text{ok}$ :  $\text{is-closed } s \ \text{is-closed } t$

**assumes**  $x: (x, \tau') \notin FV \Gamma$  *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**  $\Theta, \Gamma \vdash mk-eq s t$   
 $\longleftrightarrow \Theta, \Gamma \vdash mk-eq (Abs \tau' (bind-fv (x, \tau') s)) (Abs \tau' (bind-fv (x, \tau') t))$   
*<proof>*

**lemma** *proves-descend-abs-rule'*:

**assumes** *thy*: *wf-theory*  $\Theta$   
**assumes** *eq*:  $\Theta, \Gamma \vdash mk-eq (Abs \tau' s) (Abs \tau' t)$   
**assumes**  $x: (x, \tau') \notin FV \Gamma$  *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**  $\Theta, \Gamma \vdash mk-eq (subst-bv (Fv x \tau') s) (subst-bv (Fv x \tau') t)$   
*<proof>*

**lemma** *proves-ascend-abs-rule'*:

**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** *eq*:  $\Theta, \Gamma \vdash mk-eq (subst-bv (Fv x \tau') s) (subst-bv (Fv x \tau') t)$   
**assumes** *ctxt*:  $finite \Gamma \forall A \in \Gamma. term-ok \Theta A \forall A \in \Gamma. typ-of A = Some propT$   
**shows**  $\Theta, \Gamma \vdash mk-eq (Abs \tau' s) (Abs \tau' t)$   
*<proof>*

**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**  $\Theta, \Gamma \vdash mk-eq (subst-bv (Fv x \tau') s) (subst-bv (Fv x \tau') t)$   
 $\longleftrightarrow \Theta, \Gamma \vdash mk-eq (Abs \tau' s) (Abs \tau' t)$   
*<proof>*

**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**  $\Theta, \Gamma \vdash mk-eq$   
 $(subst-bvs (map (case-prod Fv) vs) t)$   
 $(subst-bvs (map (case-prod Fv) vs) u)$   
*<proof>*

**lemma** *subst-bvs-empty[simp]*:  $subst-bvs [] t = t$   
*<proof>*

**lemma** *proves-beta-step*:

**assumes** *thy*: *wf-theory*  $\Theta$   
**assumes** *finite*:  $finite \Gamma$

**assumes** *term-ok*: *term-ok*  $\Theta$   $t$   
**assumes** *beta*:  $t \rightarrow_{\beta} u$   
**assumes** *ctxt*:  $\forall A \in \Gamma. \text{term-ok } \Theta \ A \ \forall A \in \Gamma. \text{typ-of } A = \text{Some propT}$   
**shows**  $\Theta, \Gamma \vdash \text{mk-eq } t \ u$   
*<proof>*

**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*:  $\forall A \in \Gamma. \text{term-ok } \Theta \ A \ \forall A \in \Gamma. \text{typ-of } A = \text{Some propT}$   
**shows**  $\Theta, \Gamma \vdash \text{mk-eq } t \ u$   
*<proof>*

**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*:  $\forall A \in \Gamma. \text{term-ok } \Theta \ A \ \forall A \in \Gamma. \text{typ-of } A = \text{Some propT}$   
**shows**  $\Theta, \Gamma \vdash \text{mk-eq } t \ u$   
*<proof>*

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

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

**lemma** *proves-eta-step*:  
**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**  $\Theta, \Gamma \vdash \text{mk-eq } t \ u$   
*<proof>*

**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**  $\Theta, \Gamma \vdash \text{mk-eq } t \ u$   
*<proof>*

**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**  $\Theta, \Gamma \vdash \text{mk-eq } t \ u$   
*<proof>*

**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**  $\Theta, \Gamma \vdash u$   
*<proof>*

**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**  $\Theta, \Gamma \vdash u$   
*<proof>*

**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*: *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>*  
**end**

## 12 Proof Terms and proof checker

```

theory ProofTerm
  imports Term Logic Term-Subst SortConstants EqualityProof
begin

type-synonym tyinst = (variable × sort) × typ
type-synonym tinst = (variable × typ) × term

datatype proofterm = PAxm term tyinst list
  | PBound nat
  | Abst typ proofterm
  | AbsP term proofterm
  | Appt proofterm term
  | AppP proofterm proofterm
  | OfClass typ class
  | Hyp term

fun depth :: proofterm ⇒ nat where
  depth (Abst - P) = Suc (depth P)
| depth (AbsP - P) = Suc (depth P)
| depth (Appt P -) = Suc (depth P)
| depth (AppP P1 P2) = Suc (max (depth P1) (depth P2))
| depth - = 1
fun size :: proofterm ⇒ nat where
  size (Abst - P) = Suc (size P)
| size (AbsP - P) = Suc (size P)
| size (Appt P -) = Suc (size P)
| size (AppP P1 P2) = Suc (size P1 + size P2)
| size - = 1

lemma depth P > 0
  ⟨proof⟩
lemma size P > 0
  ⟨proof⟩
lemma size P ≥ depth P
  ⟨proof⟩

fun partial-nth :: 'a list ⇒ nat ⇒ 'a option where
  partial-nth [] - = None
| partial-nth (x#xs) 0 = Some x
| partial-nth (x#xs) (Suc n) = partial-nth xs n

definition [simp]: partial-nth' xs n ≡ if n < length xs then Some (nth xs n) else
None

lemma partial-nth xs n ≡ partial-nth' xs n

```

*<proof>*

**lemma** *partial-nth-Some-imp-elem*:  $\text{partial-nth } l \ n = \text{Some } x \implies x \in \text{set } l$   
*<proof>*

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*)  
| *replay' thy vs ns Hs (Appt p t)* =  
  (*let rep = replay' thy vs ns Hs p in*  
    *let t' = subst-bus (map ( $\lambda(x,y)$  . Fv x y) vs) t in*  
      *case (rep, typ-of t')* of  
        (*Some (Ct s (Ty fun1 [Ty fun2 [ $\tau$ , Ty propT1 Nil], Ty propT2 Nil) \$ b),*  
          *Some  $\tau'$ )*  $\Rightarrow$   
          *if s = STR "Pure.all"  $\wedge$  fun1 = STR "fun"  $\wedge$  fun2 = STR "fun"*  
             $\wedge$  *propT1 = STR "prop"  $\wedge$  propT2 = STR "prop"*  
             $\wedge$   *$\tau = \tau' \wedge$  term-ok thy t'*  
            *then Some (b  $\cdot$  t') else None*  
        | -  $\Rightarrow$  *None*)  
  | *replay' thy vs ns Hs (AbsP t p)* =  
    (*let t' = subst-bus (map ( $\lambda(x,y)$  . Fv x y) vs) t in*  
      *let rep = replay' thy vs ns (t' # Hs) p in*  
      (*if typ-of t' = Some propT  $\wedge$  term-ok thy t' then map-option (mk-imp t') rep*  
      *else None*))  
  | *replay' thy vs ns Hs (AppP p1 p2)* =  
    (*let rep1 = Option.bind (replay' thy vs ns Hs p1) beta-eta-norm in*  
      *let rep2 = Option.bind (replay' thy vs ns Hs p2) beta-eta-norm in*  
      (*case (rep1, rep2) of*  
        (*Some (Ct imp (Ty fn1 [Ty prp1 [], Ty fn2 [Ty prp2 [], Ty prp3 []]) \$ A \$ B),*  
          *Some A')*  $\Rightarrow$   
          *if 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'*  
            *then Some B else None*  
        | -  $\Rightarrow$  *None*)  
    | *replay' thy vs ns Hs (OfClass ty c)* = (*if has-sort (osig (sig thy)) ty {c}*  
       $\wedge$  *typ-ok thy ty*  
      *then (case const-type (sig thy) (const-of-class c) of*  
        (*Some (Ty fun [Ty it [ity], Ty prop [])*  $\Rightarrow$   
          *if ity = tvariable STR "'a'"  $\wedge$  fun = STR "fun"  $\wedge$  prop = STR "prop"  $\wedge$  it*



= STR "itself"  
 then Some (mk-of-class ty c) else None | -  $\Rightarrow$  None) else None)  
 | replay' thy vs ns Hs (Hyp t) = (if t $\in$ set Hs then Some t else None)

**lemma** fv-subst-bv1:

fv (subst-bv1 t lev u) = fv t  $\cup$  (if loose-bvar1 t lev then fv u else {})  
 <proof>

**corollary** fv-subst-bvs-upper-bound:

assumes is-closed t

shows fv (subst-bvs us t)  $\subseteq$  fv t  $\cup$  ( $\bigcup$  x $\in$ set us . (fv x))  
 <proof>

**lemma** fv-subst-bvs1-upper-bound:

fv (subst-bvs1 t lev us)  $\subseteq$  fv t  $\cup$  ( $\bigcup$  x $\in$ set us . (fv x))  
 <proof>

**lemma** typ-of-axiom: wf-theory thy  $\Rightarrow$  t  $\in$  axioms thy  $\Rightarrow$  typ-of t = Some propT

<proof>

**fun** fv-Proof :: proofterm  $\Rightarrow$  (variable  $\times$  typ) set **where**

fv-Proof (PAxm t -) = fv t  
 | fv-Proof (PBound -) = empty  
 | fv-Proof (Abst - p) = fv-Proof p  
 | fv-Proof (AbsP t p) = fv t  $\cup$  fv-Proof p  
 | fv-Proof (Appt p t) = fv-Proof p  $\cup$  fv t  
 | fv-Proof (AppP p1 p2) = fv-Proof p1  $\cup$  fv-Proof p2  
 | fv-Proof (OfClass - -) = empty  
 | fv-Proof (Hyp t) = fv t

**lemma** typ-ok-Tv[simp]: typ-ok thy (Tv idn S) = wf-sort (subclass (osig (sig thy)))  
 S

<proof>

**lemma** typ-ok-contained-tvars-typ-ok: typ-ok thy ty  $\Rightarrow$  (idn, S)  $\in$  tvsT ty  $\Rightarrow$   
 typ-ok thy (Tv idn S)

<proof>

**lemma** typ-ok-sig-contained-tvars-typ-ok-sig:

typ-ok-sig  $\Sigma$  ty  $\Rightarrow$  (idn, S)  $\in$  tvsT ty  $\Rightarrow$  typ-ok-sig  $\Sigma$  (Tv idn S)  
 <proof>

**lemma** term-ok'-contained-tvars-typ-ok-sig:

term-ok'  $\Sigma$  t  $\Rightarrow$  (idn, S)  $\in$  tvs t  $\Rightarrow$  typ-ok-sig  $\Sigma$  (Tv idn S)

<proof>

**lemma** *term-ok-contained-tvars-typ-ok*:

$term-ok\ thy\ t \implies (idn, S) \in tvs\ t \implies typ-ok\ thy\ (Tv\ idn\ S)$

$\langle proof \rangle$

**lemma** *typ-ok-subst-typ*:

$typ-ok\ thy\ T \implies \forall (-, ty) \in set\ insts . typ-ok\ thy\ ty \implies typ-ok\ thy\ (subst-typ\ insts\ T)$

$\langle proof \rangle$

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

$typ-ok-sig\ \Sigma\ T \implies \forall (-, ty) \in set\ insts . typ-ok-sig\ \Sigma\ ty \implies typ-ok-sig\ \Sigma\ (subst-typ\ insts\ T)$

$\langle proof \rangle$

**lemma** *typ-ok-sig-imp-sortsT-ok-sig*:  $typ-ok-sig\ \Sigma\ T \implies S \in Sorts\ T\ T \implies wf-sort\ (subclass\ (osig\ \Sigma))\ S$

$\langle proof \rangle$

**lemma** *term-ok'-imp-Sorts-ok-sig*:  $term-ok'\ \Sigma\ t \implies S \in Sorts\ t \implies wf-sort\ (subclass\ (osig\ \Sigma))\ S$

$\langle proof \rangle$

**lemma** *replay'-sound-pre*:

**assumes** *thy*: *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*

$\langle proof \rangle$

**lemma** *finite-fv-Proof*: *finite (fv-Proof P)*

$\langle proof \rangle$

**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:*

$\bigwedge x. x \in \text{set } Hs \implies \text{term-ok } \text{thy } x$

$\bigwedge x. x \in \text{set } Hs \implies \text{typ-of } x = \text{Some propT}$

**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>*

**lemma**

**assumes** *wf-theory thy*

**assumes** *replay'' thy [] (fst ' fv-Proof P) [] P = Some res*

**shows** *thy, set []  $\vdash$  res*

*<proof>*

**fun** *hyps :: proofterm  $\Rightarrow$  term list where*

*hyps (Abst - p) = hyps p*

| *hyps (AbsP - p) = hyps p*

| *hyps (Appt p -) = hyps p*

| *hyps (AppP p1 p2) = List.union (hyps p1) (hyps p2)*

| *hyps (Hyp t) = [t]*

| *hyps - = []*

**lemma** *replay''-sound-pre-hyps:*

**assumes** *wf-theory thy*

**assumes**  $\bigwedge x. x \in \text{set } (\text{hyps } P) \implies \text{term-ok } \text{thy } x$

**assumes**  $\bigwedge x. x \in \text{set } (\text{hyps } P) \implies \text{typ-of } x = \text{Some propT}$

**assumes** *replay'' thy [] (fst ' (fv-Proof P  $\cup$  FV (set (hyps P)))) (hyps P) P = Some res*

**shows** *thy, set (hyps P)  $\vdash$  res*

*<proof>*

**definition** [*simp*]: *replay thy P  $\equiv$*

*(if  $\forall x \in \text{set } (\text{hyps } P) . \text{term-ok } \text{thy } x \wedge \text{typ-of } x = \text{Some propT}$  then*

*replay'' thy [] (fst ' (fv-Proof P  $\cup$  FV (set (hyps P)))) (hyps P) P else None)*

**lemma** *replay-sound-pre-hyps:*

**assumes** *wf-theory thy*  
**assumes** *replay thy P = Some res*  
**shows** *thy, set (hyps P) ⊢ res*  
 ⟨*proof*⟩

**definition** *check-proof thy P res ≡ wf-theory thy ∧ replay thy P = Some res*

**lemma** *check-proof-sound:*  
**shows** *check-proof thy P res ⇒ thy, set (hyps P) ⊢ res*  
 ⟨*proof*⟩

**lemma** *check-proof-really-sound:*  
**assumes** *check-proof thy P res*  
**shows** *thy, set (hyps P) ⊢ res*  
 ⟨*proof*⟩

**end**

## 13 Executable Sorts

**theory** *SortsExe*  
**imports** *Sorts*  
**begin**

**type-synonym** *exeosig = (class × class) list × (name × (class × sort list) list)*  
*list*

**abbreviation** *(input) execlasses ≡ fst*  
**abbreviation** *(input) exetcSIGs ≡ snd*

**abbreviation** *alist-conds :: ('k::linorder × 'v) list ⇒ bool* **where**  
*alist-conds al ≡ distinct (map fst al)*

**definition** *exe-ars-conds :: (name × (class × sort list) list) list ⇒ bool* **where**  
*exe-ars-conds arss ⟷ alist-conds arss ∧ (∀ ars ∈ snd ' set arss . alist-conds ars)*

**fun** *exe-ars-conds' :: (('k1::linorder) × (('k2::linorder) × 's list) list) list ⇒ bool*  
**where**  
*exe-ars-conds' arss ⟷ alist-conds arss ∧ (∀ ars ∈ snd ' set arss . alist-conds ars)*

**lemma** [*code*]: *exe-ars-conds arss ⟷ exe-ars-conds' arss*  
 ⟨*proof*⟩

**definition** *exe-class-conds :: (class × class) list ⇒ bool* **where**  
*exe-class-conds cs ≡ distinct cs*

**definition** *exe-osig-conds* :: *exeosig*  $\Rightarrow$  *bool* **where**

*exe-osig-conds* *a*  $\equiv$  *exe-class-conds* (*execlasses* *a*)  $\wedge$  *exe-ars-conds* (*exetcSIGs* *a*)

**fun** *translate-ars* :: (*name*  $\times$  (*class*  $\times$  *sort list*) *list*) *list*  $\Rightarrow$  *name*  $\rightarrow$  (*class*  $\rightarrow$  *sort list*) **where**

*translate-ars* *ars* = *map-of* (*map* (*apsnd map-of*) *ars*)

**abbreviation** *illformed-osig*  $\equiv$  ( $\{\}$ , *Map.empty*(*STR* "A"  $\mapsto$  *Map.empty*(*STR* "A"  $\mapsto$   $\{\{STR$  "A"\}\}))

**lemma** *illformed-osig-not-wf-osig*:  $\neg$  *wf-osig* *illformed-osig*  
(*proof*)

**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*<sub>1</sub>  $\in$  *snd* ' *set* *ars*.  $\forall$  *ss*<sub>2</sub>  $\in$  *snd* ' *set* *ars*. *length* *ss*<sub>1</sub> = *length* *ss*<sub>2</sub>)

**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*)  
(*proof*)

**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*  
(*proof*)

**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*  
(*proof*)

**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*  
(*proof*)

**lemma** *exe-consistent-length-tcsigs-imp-consistent-length-tcsigs*:  
**assumes** *exe-ars-conds* *arss* *exe-consistent-length-tcsigs* *arss*  
**shows** *consistent-length-tcsigs* (*translate-ars* *arss*)  
(*proof*)

**lemma** *consistent-length-tcsigs-iff-exe-consistent-length-tcsigs*:  
*exe-ars-conds* *arss*  $\Rightarrow$   
*consistent-length-tcsigs* (*translate-ars* *arss*)  $\longleftrightarrow$  *exe-consistent-length-tcsigs* *arss*

*<proof>*

**definition** *exe-complete-tcsigs cs arss*

$\equiv (\forall ars \in snd \text{ ' set arss .}$

$\forall (c_1, c_2) \in set\ cs . c_1 \in fst \text{ ' set ars} \longrightarrow c_2 \in fst \text{ ' set ars})$

**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>*

**lemma** *complete-tcsigs-imp-exe-complete-tcsigs: exe-ars-conds arss  $\implies$*

*complete-tcsigs (set cs) (translate-ars arss)  $\implies$  exe-complete-tcsigs cs arss*

*<proof>*

**lemma** *exe-complete-tcsigs-iff-complete-tcsigs:*

*exe-ars-conds arss  $\implies$*

*complete-tcsigs (set cs) (translate-ars arss)  $\longleftrightarrow$  exe-complete-tcsigs cs arss*

*<proof>*

**definition** *exe-coregular-tcsigs (cs :: (class  $\times$  class) list) arss*

$\equiv (\forall ars \in snd \text{ ' set arss .}$

$\forall c_1 \in fst \text{ ' set ars. } \forall c_2 \in fst \text{ ' set ars.}$

$(class\ leq (set\ cs) c_1 c_2 \longrightarrow$

$list\ all2 (sort\ leq (set\ cs)) (the (lookup (\lambda x. x=c_1) ars)) (the (lookup (\lambda x. x=c_2) ars))))$

**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>*

**lemma** *coregular-tcsigs-imp-exe-coregular-tcsigs:*

**assumes** *exe-ars-conds arss coregular-tcsigs (set cs) (translate-ars arss)*

**shows** *exe-coregular-tcsigs cs arss*

*<proof>*

**lemma** *coregular-tcsigs-iff-exe-coregular-tcsigs:*

*exe-ars-conds arss  $\implies$  coregular-tcsigs (set cs) (translate-ars arss)  $\longleftrightarrow$  exe-coregular-tcsigs cs arss*

*<proof>*

**lemma** *wf-subclass sub  $\implies$  Field sub = Domain sub*

*<proof>*

**definition** [*simp*]: *exefield rel = List.union (map fst rel) (map snd rel)*

**lemma** *Field-set-code: Field (set rel) = set (exefield rel)*

*<proof>*

**lemma** *class-ex-rec*:  $\text{finite } r \implies \text{class-ex } (\text{insert } (a,b) r) c = (a=c \vee b=c \vee \text{class-ex } r c)$

*<proof>*

**definition** [*simp*]:  $\text{execlass-ex rel } c = \text{List.member } (\text{exefield rel}) c$

**lemma** *execlass-ex-code*:  $\text{class-ex } (\text{set rel}) c = \text{execlass-ex rel } c$

*<proof>*

**definition** [*simp*]:  $\text{exesort-ex rel } S = (\forall x \in S . (\text{List.member } (\text{exefield rel}) x))$

**lemma** *sort-ex-code*:  $\text{sort-ex } (\text{set rel}) S = \text{exesort-ex rel } S$

*<proof>*

**definition** [*simp*]:  $\text{execlass-les } cs c1 c2 = (\text{List.member } cs (c1,c2) \wedge \neg \text{List.member } cs (c2,c1))$

**lemma** *execlass-les-code*:  $\text{class-les } (\text{set } cs) c1 c2 = \text{execlass-les } cs c1 c2$

*<proof>*

**definition** [*simp*]:  $\text{exenormalize-sort } cs (s::\text{sort})$

$= \{c \in s . \neg (\exists c' \in s . \text{execlass-les } cs c' c)\}$

**definition** [*simp*]:  $\text{exenormalized-sort } cs s \equiv (\text{exenormalize-sort } cs s) = s$

**lemma** *normalize-sort-code*[*code*]:  $\text{normalize-sort } (\text{set } cs) s = \text{exenormalize-sort } cs s$

*<proof>*

**lemma** *normalized-sort-code*[*code*]:  $\text{normalized-sort } (\text{set } cs) s = \text{exenormalized-sort } cs s$

*<proof>*

**definition** [*simp*]:  $\text{exewf-sort sub } S \equiv \text{exenormalized-sort sub } S \wedge \text{exesort-ex sub } S$

**lemma** *wf-sort-code*:

**assumes** *exe-class-conds sub*

**shows**  $\text{wf-sort } (\text{set sub}) S = \text{exewf-sort sub } S$

*<proof>*

**declare** *exewf-sort-def*[*code del*]

**lemma** [*code*]:  $\text{exewf-sort sub } S \equiv (S = \{\}) \vee \text{exenormalized-sort sub } S \wedge \text{exesort-ex sub } S$

*<proof>*

**definition** *exe-all-normalized-and-ex-tcsigs*  $cs arss$

$\equiv (\forall ars \in \text{snd } ' \text{set } arss . \forall ss \in \text{snd } ' \text{set } ars . \forall s \in \text{set } ss . \text{exewf-sort } cs s)$

**lemma** *all-normalized-and-ex-tcsigs-imp-exe-all-normalized-and-ex-tcsigs*:

**assumes** *exe-ars-conds arss all-normalized-and-ex-tcsigs (set cs) (translate-ars arss)*

**shows** *exe-all-normalized-and-ex-tcsigs*  $cs arss$

*<proof>*

**lemma** *exe-all-normalized-and-ex-tcsigs-imp-all-normalized-and-ex-tcsigs:*

**assumes** *exe-ars-conds arss exe-all-normalized-and-ex-tcsigs cs arss*

**shows** *all-normalized-and-ex-tcsigs (set cs) (translate-ars arss)*

*<proof>*

**lemma** *all-normalized-and-ex-tcsigs-iff-exe-all-normalized-and-ex-tcsigs:*

*exe-ars-conds arss  $\implies$  all-normalized-and-ex-tcsigs (set cs) (translate-ars arss)*

*$\longleftrightarrow$  exe-all-normalized-and-ex-tcsigs cs arss*

*<proof>*

**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*

*<proof>*

**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)*

*<proof>*

**lemma** *antisym-imp-exe-antisym: antisym (set l)  $\implies$  exe-antisym l*

*<proof>*

**lemma** *antisym-iff-exe-antisym: antisym (set l) = exe-antisym l*

*<proof>*

**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*

*<proof>*

**definition** [*simp*]: *exe-wf-osig oss  $\equiv$  exe-wf-subclass (execlases oss)*

*$\wedge$  exe-wf-tcsigs (execlases 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)*

*<proof>*

**lemma** *classes-translate: exe-osig-conds oss  $\implies$  subclass (translate-osig oss) = set (execlases oss)*



*<proof>*

**lemma** *tcsigs-translate: exe-osig-conds oss*  
 $\implies$  *tcsigs (translate-osig oss) = translate-ars (exetcsigs oss)*  
*<proof>*

**lemma** *wf-osig-translate-imp-exe-osig-conds:*  
*wf-osig (translate-osig oss)  $\implies$  exe-osig-conds oss*  
*<proof>*

**lemma** *wf-osig-imp-exe-wf-osig:*  
**assumes** *wf-osig (translate-osig oss)* **shows** *exe-wf-osig oss*  
*<proof>*

**lemma** *wf-osig-iff-exe-wf-osig: wf-osig (translate-osig oss)  $\longleftrightarrow$  exe-wf-osig oss*  
*<proof>*

**end**

## 14 Executable Instance Relations

**theory** *Instances*  
**imports** *Term*  
**begin**

**fun** *raw-match* :: *typ  $\Rightarrow$  typ  $\Rightarrow$  ((variable  $\times$  sort)  $\rightarrow$  typ)  $\Rightarrow$  ((variable  $\times$  sort)  $\rightarrow$  typ) option*

**and** *raw-matches* :: *typ list  $\Rightarrow$  typ list  $\Rightarrow$  ((variable  $\times$  sort)  $\rightarrow$  typ)  $\Rightarrow$  ((variable  $\times$  sort)  $\rightarrow$  typ) option*

**where**

*raw-match (Tv v S) T subs =*  
  (*case subs (v,S) of*  
    *None  $\Rightarrow$  Some (subs((v,S) := Some T)*  
    | *Some U  $\Rightarrow$  (if U = T then Some subs else None)*)

| *raw-match (Ty a Ts) (Ty b Us) subs =*  
  (*if a=b then raw-matches Ts Us subs else None*)

| *raw-match - - = None*

| *raw-matches (T#Ts) (U#Us) subs = Option.bind (raw-match T U subs) (raw-matches Ts Us)*

| *raw-matches [] [] subs = Some subs*

| *raw-matches - - subs = None*

**function** (*sequential*) *raw-match'*  
  :: *typ  $\Rightarrow$  typ  $\Rightarrow$  ((variable  $\times$  sort)  $\rightarrow$  typ)  $\Rightarrow$  ((variable  $\times$  sort)  $\rightarrow$  typ) option*

**where**

$raw-match' (Tv v S) T subs =$   
 $(case\ subs\ (v,S)\ of$   
 $\quad None \Rightarrow Some\ (subs((v,S) := Some\ T))$   
 $\quad | Some\ U \Rightarrow (if\ U = T\ then\ Some\ subs\ else\ None))$   
 $| raw-match' (Ty a Ts) (Ty b Us) subs =$   
 $\quad (if\ a=b \wedge length\ Ts = length\ Us$   
 $\quad\quad then\ fold\ (\lambda(T, U)\ subs . Option.bind\ subs\ (raw-match' T U))\ (zip\ Ts\ Us)$   
 $\quad (Some\ subs)$   
 $\quad\quad else\ None)$   
 $| raw-match' T U subs = (if\ T = U\ then\ Some\ subs\ else\ None)$   
 $\langle proof \rangle$   
**termination**  $\langle proof \rangle$

**lemma** *length-neq-imp-not-raw-matches*:  $length\ Ts \neq length\ Us \implies raw-matches\ Ts\ Us\ subs = None$   
 $\langle proof \rangle$

**lemma** *raw-match T U subs = raw-match' T U subs*  
 $\langle proof \rangle$

**lemma** *raw-match'-map-le*:  $raw-match' T U subs = Some\ subs' \implies map-le\ subs\ subs'$   
 $\langle proof \rangle$

**lemma** *fold-matches-first-step-not-None*:

**assumes**  
 $fold\ (\lambda(T, U)\ subs . Option.bind\ subs\ (raw-match' T U))\ (zip\ (x\#xs)\ (y\#ys))$   
 $(Some\ subs) = Some\ subs'$   
**obtains point where**  
 $raw-match' x y subs = Some\ point$   
 $fold\ (\lambda(T, U)\ subs . Option.bind\ subs\ (raw-match' T U))\ (zip\ (xs)\ (ys))\ (Some\ point) = Some\ subs'$   
 $\langle proof \rangle$

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

**assumes**  
 $length\ xs = length\ ys$   
 $fold\ (\lambda(T, U)\ subs . Option.bind\ subs\ (raw-match' T U))\ (zip\ (xs@[x])\ (ys@[y]))$   
 $(Some\ subs) = Some\ subs'$   
**obtains point where**  
 $fold\ (\lambda(T, U)\ subs . Option.bind\ subs\ (raw-match' T U))\ (zip\ (xs)\ (ys))\ (Some\ subs) = Some\ point$   
 $raw-match' x y point = Some\ subs'$   
 $\langle proof \rangle$

**corollary** *raw-match'-Type-conds*:

**assumes**  $raw-match' (Ty a Ts) (Ty b Us) subs = Some\ subs'$   
**shows**  $a=b\ length\ Ts = length\ Us$

*<proof>*

**corollary** *fold-matches-first-step-not-None'*:

**assumes**  $\text{length } xs = \text{length } ys$   
 $\text{fold } (\lambda(T, U) \text{ subs} . \text{Option.bind } \text{subs} (\text{raw-match}' T U)) (\text{zip } (x\#xs) (y\#ys))$   
 $(\text{Some } \text{subs}) = \text{Some } \text{subs}'$   
**shows**  $\text{raw-match}' x y \text{subs} \sim = \text{None}$   
*<proof>*

**corollary** *raw-match'-hd-raw-match'*:

**assumes**  $\text{raw-match}' (Ty a (T\#Ts)) (Ty b (U\#Us)) \text{subs} = \text{Some } \text{subs}'$   
**shows**  $\text{raw-match}' T U \text{subs} \sim = \text{None}$   
*<proof>*

**corollary** *raw-match'-eq-Some-at-point-not-None'*:

**assumes**  $\text{length } Ts = \text{length } Us$   
**assumes**  $\text{raw-match}' (Ty a (Ts@Ts')) (Ty b (Us@Us')) \text{subs} = \text{Some } \text{subs}'$   
**shows**  $\text{raw-match}' (Ty a (Ts)) (Ty b (Us)) \text{subs} \sim = \text{None}$   
*<proof>*

**lemma** *raw-match'-tvsT-subset-dom-res*:  $\text{raw-match}' T U \text{subs} = \text{Some } \text{subs}' \implies$   
 $\text{tvsT } T \subseteq \text{dom } \text{subs}'$   
*<proof>*

**lemma** *raw-match'-dom-res-subset-tvsT*:

$\text{raw-match}' T U \text{subs} = \text{Some } \text{subs}' \implies \text{dom } \text{subs}' \subseteq \text{tvsT } T \cup \text{dom } \text{subs}$   
*<proof>*

**corollary** *raw-match'-dom-res-eq-tvsT*:

$\text{raw-match}' T U \text{subs} = \text{Some } \text{subs}' \implies \text{dom } \text{subs}' = \text{tvsT } T \cup \text{dom } \text{subs}$   
*<proof>*

**corollary** *raw-match'-dom-res-eq-tvsT-empty*:

$\text{raw-match}' T U (\lambda x. \text{None}) = \text{Some } \text{subs}' \implies \text{dom } \text{subs}' = \text{tvsT } T$   
*<proof>*

**lemma** *raw-match'-map-defined*:  $\text{raw-match}' T U \text{subs} = \text{Some } \text{subs}' \implies p \in \text{tvsT } T$   
 $\implies \text{subs}' p \sim = \text{None}$   
*<proof>*

**lemma** *raw-match'-extend-map-preserve*:

$\text{raw-match}' T U \text{subs} = \text{Some } \text{subs}' \implies \text{map-le } \text{subs}' \text{subs}'' \implies p \in \text{tvsT } T \implies$   
 $\text{subs}'' p = \text{subs}' p$   
*<proof>*

**abbreviation** *convert-subs*  $\text{subs} \equiv (\lambda v S . \text{the-default } (Tv v S) (\text{subs } (v, S)))$

**lemma** *map-eq-on-tvsT-imp-map-eq-on-tyt*:

$(\bigwedge p . p \in \text{tvs}T \ T \implies \text{subs } p = \text{subs}' p)$   
 $\implies \text{tsubst}T \ T \ (\text{convert-subs } \text{subs})$   
 $= \text{tsubst}T \ T \ (\text{convert-subs } \text{subs}')$   
*<proof>*

**lemma** *raw-match'-extend-map-preserve'*:

**assumes** *raw-match' T U subs = Some subs' map-le subs' subs''*  
**shows** *tsubstT T (convert-subs subs')*  
 $= \text{tsubst}T \ T \ (\text{convert-subs } \text{subs}'')$   
*<proof>*

**lemma** *raw-match'-produces-matcher*:

*raw-match' T U subs = Some subs'*  
 $\implies \text{tsubst}T \ T \ (\text{convert-subs } \text{subs}') = U$   
*<proof>*

**lemma** *tsubstT-matcher-imp-raw-match'-unchanged*:

*tsubstT T q = U  $\implies$  raw-match' T U ( $\lambda(\text{idx}, S). \text{Some } (q \ \text{idx } S)$ ) = Some*  
*( $\lambda(\text{idx}, S). \text{Some } (q \ \text{idx } S)$ )*  
*<proof>*

**lemma** *raw-match'-imp-raw-match'-on-map-le*:

**assumes** *raw-match' T U subs = Some subs'*  
**assumes** *map-le lesubs subs*  
**shows**  $\exists \text{lesubs}' . \text{raw-match}' \ T \ U \ \text{lesubs} = \text{Some } \text{lesubs}' \wedge \text{map-le } \text{lesubs}' \ \text{subs}'$   
*<proof>*

**lemma** *map-le-same-dom-imp-same-map*: *dom f = dom g  $\implies$  map-le f g  $\implies$  f =*

*g*  
*<proof>*

**corollary** *map-le-produces-same-raw-match'*:

**assumes** *raw-match' T U subs = Some subs'*  
**assumes** *dom subs  $\subseteq$  tvsT T*  
**assumes** *map-le lesubs subs*  
**shows** *raw-match' T U lesubs = Some subs'*  
*<proof>*

**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'*  
*<proof>*

**lemma** *raw-match'-restriction*:

**assumes** *raw-match' T U subs = Some subs'*  
**assumes** *tvsT T  $\subseteq$  restriction*  
**shows** *raw-match' T U (subs|'restriction) = Some (subs'|'restriction)*  
*<proof>*

**corollary** *raw-match'-restriction-on-tvsT*:

**assumes** *raw-match' T U subs = Some subs'*

**shows** *raw-match' T U (subs|'tvsT T) = Some (subs|'tvsT T)*

*<proof>*

**lemma** *tinstT-imp-ex-raw-match'*:

**assumes** *tinstT T1 T2*

**shows**  $\exists \text{subs. } \text{raw-match}' T2 T1 (\lambda p . \text{None}) = \text{Some subs}$

*<proof>*

**lemma** *ex-raw-match'-imp-tinstT*:

**assumes**  $\exists \text{subs. } \text{raw-match}' T2 T1 (\lambda p . \text{None}) = \text{Some subs}$

**shows** *tinstT T1 T2*

*<proof>*

**corollary** *tinstT-iff-ex-raw-match'*:

*tinstT T1 T2  $\longleftrightarrow$  ( $\exists \text{subs. } \text{raw-match}' T2 T1 (\lambda p . \text{None}) = \text{Some subs}$ )*

*<proof>*

**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 =*

*Option.bind (raw-match' T U subs) (raw-match-term t u)*

*| raw-match-term (f \$ u) (f' \$ u') subs = Option.bind (raw-match-term f f' subs) (raw-match-term u u')*

*| raw-match-term - - - = None*

*<proof>*

**termination** *<proof>*

**lemma** *raw-match-term-map-le*: *raw-match-term t u subs = Some subs'  $\Longrightarrow$  map-le subs subs'*

*<proof>*

**lemma** *raw-match-term-tvs-subset-dom-res*:

*raw-match-term t u subs = Some subs'  $\Longrightarrow$  tvs t  $\subseteq$  dom subs'*

*<proof>*

**lemma** *raw-match-term-dom-res-subset-tvs*:

*raw-match-term t u subs = Some subs'  $\Longrightarrow$  dom subs'  $\subseteq$  tvs t  $\cup$  dom subs*

*<proof>*

**corollary** *raw-match-term-dom-res-eq-tvs:*

$raw-match-term\ t\ u\ subs = Some\ subs' \implies dom\ subs' = tvs\ t \cup dom\ subs$   
*<proof>*

**lemma** *raw-match-term-extend-map-preserve:*

$raw-match-term\ t\ u\ subs = Some\ subs' \implies map-le\ subs'\ subs'' \implies p \in tvs\ t \implies$   
 $subs''\ p = subs'\ p$   
*<proof>*

**lemma** *map-eq-on-tvs-imp-map-eq-on-term:*

$(\bigwedge p . p \in tvs\ t \implies subs\ p = subs'\ p)$   
 $\implies tsubst\ t\ (convert-sub\ subs)$   
 $= tsubst\ t\ (convert-sub\ subs')$   
*<proof>*

**lemma** *raw-match-extend-map-preserve':*

**assumes**  $raw-match-term\ t\ u\ subs = Some\ subs'\ map-le\ subs'\ subs''$   
**shows**  $tsubst\ t\ (convert-sub\ subs')$   
 $= tsubst\ t\ (convert-sub\ subs'')$   
*<proof>*

**lemma** *raw-match-term-produces-matcher:*

$raw-match-term\ t\ u\ subs = Some\ subs'$   
 $\implies tsubst\ t\ (convert-sub\ subs') = u$   
*<proof>*

**lemma** *ex-raw-match-term-imp-tinst:*

**assumes**  $\exists\ subs.\ raw-match-term\ t2\ t1\ (\lambda p . None) = Some\ subs$   
**shows**  $tinst\ t1\ t2$   
*<proof>*

**lemma** *tsubst-matcher-imp-raw-match-term-unchanged:*

$tsubst\ t\ \varrho = u \implies raw-match-term\ t\ u\ (\lambda(idx, S). Some\ (\varrho\ idx\ S)) = Some$   
 $(\lambda(idx, S). Some\ (\varrho\ idx\ S))$   
*<proof>*

**lemma** *raw-match-term-restriction:*

**assumes**  $raw-match-term\ t\ u\ subs = Some\ subs'$   
**assumes**  $tvs\ t \subseteq restriction$   
**shows**  $raw-match-term\ t\ u\ (subs|'restriction) = Some\ (subs'|'restriction)$   
*<proof>*

**corollary** *raw-match-term-restriction-on-tvs:*

**assumes**  $raw-match-term\ t\ u\ subs = Some\ subs'$   
**shows**  $raw-match-term\ t\ u\ (subs|'tvs\ t) = Some\ (subs'|'tvs\ t)$   
*<proof>*

**lemma** *raw-match-term-imp-raw-match-term-on-map-le:*

**assumes**  $raw-match-term\ t\ u\ subs = Some\ subs'$

**assumes** *map-le lesubs subs*  
**shows**  $\exists \text{lesubs}' . \text{raw-match-term } t \ u \ \text{lesubs} = \text{Some } \text{lesubs}' \wedge \text{map-le } \text{lesubs}' \ \text{subs}'$   
 $\langle \text{proof} \rangle$

**corollary** *map-le-produces-same-raw-match-term:*  
**assumes** *raw-match-term t u subs = Some subs'*  
**assumes** *dom subs  $\subseteq$  tvs t*  
**assumes** *map-le lesubs subs*  
**shows** *raw-match-term t u lesubs = Some subs'*  
 $\langle \text{proof} \rangle$

**lemma** *tinst-imp-ex-raw-match-term:*  
**assumes** *tinst t1 t2*  
**shows**  $\exists \text{subs} . \text{raw-match-term } t2 \ t1 \ (\lambda p . \text{None}) = \text{Some } \text{subs}$   
 $\langle \text{proof} \rangle$

**corollary** *tinst-iff-ex-raw-match-term:*  
 $tinst \ t1 \ t2 \longleftrightarrow (\exists \text{subs} . \text{raw-match-term } t2 \ t1 \ (\lambda p . \text{None}) = \text{Some } \text{subs})$   
 $\langle \text{proof} \rangle$

**function** *(sequential) assoc-match*  
 $:: \text{typ} \Rightarrow \text{typ} \Rightarrow ((\text{variable} \times \text{sort}) \times \text{typ}) \ \text{list} \Rightarrow ((\text{variable} \times \text{sort}) \times \text{typ}) \ \text{list}$   
*option where*  
 $\text{assoc-match } (Tv \ v \ S) \ T \ \text{subs} =$   
 $(\text{case } \text{lookup } (\lambda x . x=(v,S)) \ \text{subs} \ \text{of}$   
 $\quad \text{None} \Rightarrow \text{Some } ((v,S), T) \ \# \ \text{subs}$   
 $\quad | \ \text{Some } U \Rightarrow (\text{if } U = T \ \text{then } \text{Some } \text{subs} \ \text{else } \text{None}))$   
 $| \ \text{assoc-match } (Ty \ a \ Ts) \ (Ty \ b \ Us) \ \text{subs} =$   
 $(\text{if } a=b \wedge \text{length } Ts = \text{length } Us$   
 $\quad \text{then } \text{fold } (\lambda(T, U) \ \text{subs} . \text{Option.bind } \text{subs} \ (\text{assoc-match } T \ U)) \ (\text{zip } Ts \ Us)$   
 $(\text{Some } \text{subs})$   
 $\quad \text{else } \text{None})$   
 $| \ \text{assoc-match } T \ U \ \text{subs} = (\text{if } T = U \ \text{then } \text{Some } \text{subs} \ \text{else } \text{None})$   
 $\langle \text{proof} \rangle$

**termination**  $\langle \text{proof} \rangle$

**corollary** *assoc-match-Type-conds:*  
**assumes** *assoc-match (Ty a Ts) (Ty b Us) subs = Some subs'*  
**shows** *a=b length Ts = length Us*  
 $\langle \text{proof} \rangle$

**lemma** *fold-assoc-matches-first-step-not-None:*  
**assumes**  
 $\text{fold } (\lambda(T, U) \ \text{subs} . \text{Option.bind } \text{subs} \ (\text{assoc-match } T \ U)) \ (\text{zip } (x\#xs) \ (y\#ys))$   
 $(\text{Some } \text{subs}) = \text{Some } \text{subs}'$   
**obtains point where**  
 $\text{assoc-match } x \ y \ \text{subs} = \text{Some } \text{point}$

*fold* ( $\lambda(T, U) \text{ subs} . \text{Option.bind } \text{subs} (\text{assoc-match } T \ U) (\text{zip } (xs) \ (ys)) (\text{Some } \text{point}) = \text{Some } \text{subs}'$ )  
 ⟨*proof*⟩

**lemma** *assoc-match-subset*:  $\text{assoc-match } T \ U \ \text{subs} = \text{Some } \text{subs}' \implies \text{set } \text{subs} \subseteq \text{set } \text{subs}'$   
 ⟨*proof*⟩

**lemma** *assoc-match-distinct*:  $\text{assoc-match } T \ U \ \text{subs} = \text{Some } \text{subs}' \implies \text{distinct } (\text{map } \text{fst } \text{subs}) \implies \text{distinct } (\text{map } \text{fst } \text{subs}')$   
 ⟨*proof*⟩

**lemma** *lookup-eq-map-of-ap*:  
**shows**  $\text{lookup } (\lambda x. x=k) \ \text{subs} = \text{map-of } \text{subs} \ k$   
 ⟨*proof*⟩

**lemma** *raw-match'-assoc-match*:  
**shows**  $\text{raw-match}' \ T \ U \ (\text{map-of } \text{subs}) = \text{map-option } \text{map-of } (\text{assoc-match } T \ U \ \text{subs})$   
 ⟨*proof*⟩

**lemma** *dom-eq-and-eq-on-dom-imp-eq*:  $\text{dom } m = \text{dom } m' \implies \forall x \in \text{dom } m . m \ x = m' \ x \implies m = m'$   
 ⟨*proof*⟩

**lemma** *list-of-map*:  
**assumes** *finite* (*dom* *subs*)  
**shows**  $\exists l. \text{map-of } l = \text{subs}$   
 ⟨*proof*⟩

**corollary** *tinstT-iff-assoc-match*[code]:  $\text{tinstT } T1 \ T2 \longleftrightarrow \text{assoc-match } T2 \ T1 \ \square$   
 $\sim = \text{None}$   
 ⟨*proof*⟩

**function** (*sequential*) *assoc-match-term*  
 :: *term*  $\Rightarrow$  *term*  $\Rightarrow$   $((\text{variable} \times \text{sort}) \times \text{typ}) \ \text{list} \Rightarrow ((\text{variable} \times \text{sort}) \times \text{typ}) \ \text{list}$   
*option*  
**where**  
 $\text{assoc-match-term } (\text{Ct } a \ T) \ (\text{Ct } b \ U) \ \text{subs} = (\text{if } a = b \ \text{then } \text{assoc-match } T \ U \ \text{subs} \ \text{else } \text{None})$   
 $|\ \text{assoc-match-term } (\text{Fv } a \ T) \ (\text{Fv } b \ U) \ \text{subs} = (\text{if } a = b \ \text{then } \text{assoc-match } T \ U \ \text{subs} \ \text{else } \text{None})$   
 $|\ \text{assoc-match-term } (\text{Bv } i) \ (\text{Bv } j) \ \text{subs} = (\text{if } i = j \ \text{then } \text{Some } \text{subs} \ \text{else } \text{None})$   
 $|\ \text{assoc-match-term } (\text{Abs } T \ t) \ (\text{Abs } U \ u) \ \text{subs} =$   
 $\text{Option.bind } (\text{assoc-match } T \ U \ \text{subs}) \ (\text{assoc-match-term } t \ u)$



| *assoc-match-term* (f \$ u) (f' \$ u') subs = *Option.bind* (*assoc-match-term* f f'  
 subs) (*assoc-match-term* u u')  
 | *assoc-match-term* - - - = *None*  
 ⟨*proof*⟩  
**termination** ⟨*proof*⟩

**lemma** *raw-match-term-assoc-match-term*:  
*raw-match-term* t u (*map-of* subs) = *map-option map-of* (*assoc-match-term* t u  
 subs)  
 ⟨*proof*⟩

**corollary** *tinst-iff-assoc-match-term*[code]: *tinst* t1 t2  $\longleftrightarrow$  *assoc-match-term* t2 t1  
 []  $\neq$  *None*  
 ⟨*proof*⟩

**hide-fact** *fold-matches-first-step-not-None fold-matches-last-step-not-None*

**end**

## 15 Executable Signature and Theory

**theory** *TheoryExe*  
**imports** *SortsExe Theory Instances*  
**begin**

**datatype** *exesignature* = *ExeSignature*  
 (*execonst-type-of*: (name  $\times$  typ) list)  
 (*exetyp-arity-of*: (name  $\times$  nat) list)  
 (*exesorts*: *exeosig*)

**lemma** *exe-const-type-of-ok*:  
*alist-conds* cto  $\implies$   
 ( $\forall$  ty  $\in$  *Map.ran* (*map-of* cto) . *typ-ok-sig* (*map-of* cto, ta, sa) ty)  
 $\longleftrightarrow$  ( $\forall$  ty  $\in$  *snd* ' *set* cto . *typ-ok-sig* (*map-of* cto, ta, sa) ty)  
 ⟨*proof*⟩

**fun** *exe-wf-sig* **where**  
*exe-wf-sig* (*ExeSignature* cto tao sa) = (*exe-wf-osig* sa  $\wedge$   
*fst* ' *set* (*exetcsigs* sa) = *fst* ' *set* tao  
 $\wedge$  ( $\forall$  type  $\in$  *fst* ' *set* (*exetcsigs* sa).  
 ( $\forall$  ars  $\in$  *snd* ' *set* (*the* (*lookup* ( $\lambda$ k. k=*type*) (*exetcsigs* sa))) .  
*the* (*lookup* ( $\lambda$ k. k=*type*) tao) = *length* ars))  
 $\wedge$  ( $\forall$  ty  $\in$  *snd* ' *set* cto . *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  
 $\wedge$  *fst* ' *set* (*exetcsigs* sa) = *fst* ' *set* tao  
 $\wedge$  ( $\forall$  type  $\in$  *fst* ' *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**  $wf\text{-sig (map-of cto, map-of tao, translate-osig sa)$   
 $\langle proof \rangle$

**lemma** *wf-sig-imp-exe-wf-sig*:

**assumes**  $alist\text{-conds cto alist-conds tao exe-osig-conds sa}$   
 $wf\text{-sig (map-of cto, map-of tao, translate-osig sa)}$   
**shows**  $(exe\text{-wf-osig sa}$   
 $\wedge fst \text{ ' set (exetcsigs sa) = fst ' set tao}$   
 $\wedge (\forall type \in fst \text{ ' set (exetcsigs sa).$   
 $(\forall ars \in snd \text{ ' set (the (lookup } (\lambda k. k=type) (exetcsigs sa))) .$   
 $the (lookup (\lambda k. k=type) tao) = length ars))$   
 $\wedge (\forall ty \in snd \text{ ' set cto . typ-ok-sig (map-of cto, map-of tao, translate-osig sa)$   
 $ty)$   
 $\langle proof \rangle$

**lemma** *wf-sig-iff-exe-wf-sig-pre*:  $alist\text{-conds cto} \implies alist\text{-conds tao} \implies exe\text{-osig-conds sa}$

$\implies wf\text{-sig (map-of cto, map-of tao, translate-osig sa) = (exe\text{-wf-osig sa}$   
 $\wedge fst \text{ ' set (exetcsigs sa) = fst ' set tao}$   
 $\wedge (\forall type \in fst \text{ ' set (exetcsigs sa).$   
 $(\forall ars \in snd \text{ ' set (the (lookup } (\lambda k. k=type) (exetcsigs sa))) .$   
 $the (lookup (\lambda k. k=type) tao) = length ars))$   
 $\wedge (\forall ty \in snd \text{ ' set cto . typ-ok-sig (map-of cto, map-of tao, translate-osig sa) ty))$   
 $\langle proof \rangle$

**lemma** *wf-sig-iff-exe-wf-sig*:  $alist\text{-conds cto} \implies alist\text{-conds tao} \implies exe\text{-osig-conds sa}$

$\implies wf\text{-sig (map-of cto, map-of tao, translate-osig sa)}$   
 $\longleftrightarrow exe\text{-wf-sig (ExeSignature cto tao sa)}$   
 $\langle proof \rangle$

**fun** *translate-signature* ::  $exesignature \Rightarrow signature$  **where**

$translate\text{-signature (ExeSignature cto tao sa)}$   
 $= (map\text{-of cto, map-of tao, translate-osig sa)}$

**fun** *exetyp-ok-sig* ::  $exesignature \Rightarrow typ \Rightarrow bool$  **where**

$exetyp\text{-ok-sig } \Sigma (Ty \ c \ Ts) = (case \text{lookup } (\lambda k. k=c) (exetyp\text{-arity-of } \Sigma) \text{ of}$   
 $None \Rightarrow False$   
 $| Some \ ar \Rightarrow length \ Ts = ar \wedge list\text{-all (exetyp-ok-sig } \Sigma) \ Ts)$   
 $| exetyp\text{-ok-sig } \Sigma (Tv \ - \ S) = exe\text{wf-sort (exe\text{classes (exesorts } \Sigma)) \ S}$

**thm** *exewf-sort-def*

**definition** [*simp*]:  $exesort\text{-ok-sig } \Sigma \ S \equiv exesort\text{-ex (exe\text{classes (exesorts } \Sigma)) \ S}$   
 $\wedge exenormalized\text{-sort (exe\text{classes (exesorts } \Sigma)) \ S}$

**lemma** *typ-arity-lookup-code*:  $type\text{-arity (translate-signature } \Sigma) \ n = lookup (\lambda k. k$

= n) (exetyp-arity-of  $\Sigma$ )  
 ⟨proof⟩

**lemma** *typ-ok-sig-code*:

**assumes** *exe-osig-conds* (exesorts  $\Sigma$ )  
**shows** *typ-ok-sig* (translate-signature  $\Sigma$ ) *ty* = *exetyp-ok-sig*  $\Sigma$  *ty*  
 ⟨proof⟩

**fun** *exe-wf-sig'* **where**

*exe-wf-sig'* (ExeSignature *cto tao sa*) = (*exe-wf-osig sa*  $\wedge$   
*fst* ' *set* (*exetcsigs sa*) = *fst* ' *set tao*  
 $\wedge$  ( $\forall$  *type*  $\in$  *fst* ' *set* (*exetcsigs sa*).  
 ( $\forall$  *ars*  $\in$  *snd* ' *set* (*the* (*lookup* ( $\lambda k. k=type$ ) (*exetcsigs sa*))) .  
*the* (*lookup* ( $\lambda k. k=type$ ) *tao*) = *length ars*)  
 $\wedge$  ( $\forall$  *ty*  $\in$  *snd* ' *set cto* . *exetyp-ok-sig* (ExeSignature *cto tao sa*) *ty*))

**lemma** *exe-wf-sig-code*[*code*]: *exe-wf-sig*  $\Sigma$  = *exe-wf-sig'*  $\Sigma$   
 ⟨proof⟩

**fun** *exeterm-ok'* :: *exesignature*  $\Rightarrow$  *term*  $\Rightarrow$  *bool* **where**

*exeterm-ok'*  $\Sigma$  (*Fv* - *T*) = *exetyp-ok-sig*  $\Sigma$  *T*  
 | *exeterm-ok'*  $\Sigma$  (*Bv* -) = *True*  
 | *exeterm-ok'*  $\Sigma$  (*Ct s T*) = (*case lookup* ( $\lambda k. k=s$ ) (*execonst-type-of*  $\Sigma$ ) of  
   *None*  $\Rightarrow$  *False*  
   | *Some ty*  $\Rightarrow$  *exetyp-ok-sig*  $\Sigma$  *T*  $\wedge$  *tinstT T ty*)  
 | *exeterm-ok'*  $\Sigma$  (*t* \$ *u*)  $\longleftrightarrow$  *exeterm-ok'*  $\Sigma$  *t*  $\wedge$  *exeterm-ok'*  $\Sigma$  *u*  
 | *exeterm-ok'*  $\Sigma$  (*Abs T t*)  $\longleftrightarrow$  *exetyp-ok-sig*  $\Sigma$  *T*  $\wedge$  *exeterm-ok'*  $\Sigma$  *t*

**lemma** *const-type-of-lookup-code*: *const-type* (translate-signature  $\Sigma$ ) *n* = *lookup*  
 ( $\lambda k. k = n$ ) (*execonst-type-of*  $\Sigma$ )  
 ⟨proof⟩

**lemma** *wt-term-code*:

**assumes** *exe-osig-conds* (exesorts  $\Sigma$ )  
**shows** *term-ok'* (translate-signature  $\Sigma$ ) *t* = *exeterm-ok'*  $\Sigma$  *t*  
 ⟨proof⟩

**datatype** *exetheory* = *ExeTheory* (*exesig*: *exesignature*) (*exeaxioms-of*: *term list*)

**lemma** *exetheory-full-exhaust*: ( $\bigwedge$  *const-type typ-arity sorts axioms*.

$\Theta =$  (*ExeTheory* (*ExeSignature const-type typ-arity sorts*) *axioms*)  $\Longrightarrow$  *P*)  
 $\Longrightarrow$  *P*  
 ⟨proof⟩

**definition** *exe-sig-conds*  $\Sigma \equiv$  *alist-conds* (*execonst-type-of*  $\Sigma$ )  $\wedge$  *alist-conds* (*exetyp-arity-of*  
 $\Sigma$ )  
 $\wedge$  *exe-osig-conds* (exesorts  $\Sigma$ )

**abbreviation** *illformed-theory*  $\equiv$  ((*Map.empty*, *Map.empty*, *illformed-osig*), {})

**lemma** *illformed-theory-not-wf-theory*:  $\neg$  *wf-theory illformed-theory*  
 ⟨*proof*⟩

**fun** *translate-theory* :: *exetheory*  $\Rightarrow$  *theory* **where**  
*translate-theory* (*ExeTheory*  $\Sigma$  *ax*) = (if *exe-sig-conds*  $\Sigma$  then  
 (*translate-signature*  $\Sigma$ , *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*)  $\wedge$   
 ( $\forall p \in \text{set } ax . \text{term-ok } (\text{translate-theory } (\text{ExeTheory } (\text{ExeSignature } \text{cto tao sa}) \text{ ax})) p \wedge \text{typ-of } p = \text{Some propT}$ )  
 $\wedge$  *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*  $\Sigma$ )  $\longleftrightarrow$   
*exe-wf-sig*  $\Sigma$   
 ⟨*proof*⟩

**lemma** *wf-sig-imp-exe-wf-sig'*: *exe-sig-conds*  $\Sigma \Longrightarrow$   
*wf-sig* (*translate-signature*  $\Sigma$ )  $\Longrightarrow$   
*exe-wf-sig*  $\Sigma$   
 ⟨*proof*⟩

**lemma** *exe-wf-sig-imp-wf-sig'*: *exe-sig-conds*  $\Sigma \Longrightarrow$   
*exe-wf-sig*  $\Sigma$   
 $\Longrightarrow$  *wf-sig* (*translate-signature*  $\Sigma$ )  
 ⟨*proof*⟩

**lemma** *wf-theory-translate-imp-exe-wf-theory*:  
**assumes** *wf-theory* (*translate-theory* *a*) **shows** *exe-wf-theory* *a*  
 ⟨*proof*⟩

**lemma** *exe-wf-theory-translate-imp-wf-theory*:  
**assumes** *exe-wf-theory* *a* **shows** *wf-theory* (*translate-theory* *a*)  
 ⟨*proof*⟩

**lemma** *wf-theory-translate-iff-exe-wf-theory*:  
*wf-theory* (*translate-theory* *a*)  $\longleftrightarrow$  *exe-wf-theory* *a*  
 ⟨*proof*⟩

**fun** *exeis-std-sig* **where** *exeis-std-sig* (*ExeSignature* *cto tao sorts*)  $\longleftrightarrow$   
*lookup* ( $\lambda k . k = \text{STR "fun"}$ ) *tao* = *Some 2*  $\wedge$  *lookup* ( $\lambda k . k = \text{STR "prop"}$ ) *tao*  
 = *Some 0*  
 $\wedge$  *lookup* ( $\lambda k . k = \text{STR "itself"}$ ) *tao* = *Some 1*  
 $\wedge$  *lookup* ( $\lambda k . k = \text{STR "Pure.eq"}$ ) *cto*

$$= \text{Some } ((\text{Tv } (\text{Var } (\text{STR } \text{"a"}, 0)) \text{ full-sort}) \rightarrow ((\text{Tv } (\text{Var } (\text{STR } \text{"a"}, 0)) \text{ full-sort}) \rightarrow \text{propT}))$$

$$\wedge \text{lookup } (\lambda k. k = \text{STR } \text{"Pure.all"}) \text{ cto} = \text{Some } ((\text{Tv } (\text{Var } (\text{STR } \text{"a"}, 0)) \text{ full-sort}) \rightarrow \text{propT}) \rightarrow \text{propT})$$

$$\wedge \text{lookup } (\lambda k. k = \text{STR } \text{"Pure.imp"}) \text{ cto} = \text{Some } (\text{propT} \rightarrow (\text{propT} \rightarrow \text{propT}))$$

$$\wedge \text{lookup } (\lambda k. k = \text{STR } \text{"Pure.type"}) \text{ cto} = \text{Some } (\text{itselfT } (\text{Tv } (\text{Var } (\text{STR } \text{"a"}, 0)) \text{ full-sort}))$$

**lemma** *is-std-sig-code*:  $\text{is-std-sig } (\text{translate-signature } \Sigma) = \text{exeis-std-sig } \Sigma$   
 ⟨proof⟩

**fun** *exe-wf-theory'* **where** *exe-wf-theory'* (*ExeTheory* (*ExeSignature* *cto tao sa*) *ax*)  
 $\longleftrightarrow$   
*exe-sig-conds* (*ExeSignature* *cto tao sa*)  $\wedge$   
 $(\forall p \in \text{set } ax . \text{exeterm-ok}' (\text{ExeSignature } cto \text{ tao } sa) p \wedge \text{typ-of } p = \text{Some } \text{propT})$   
 $\wedge$  *exeis-std-sig* (*ExeSignature* *cto tao sa*)  
 $\wedge$  *exe-wf-sig* (*ExeSignature* *cto tao sa*)  
 $\wedge$  *eq-axs*  $\subseteq$  *set* *ax*

**lemma** *term-ok'-code*:

**assumes** *exe-osig-conds* (*exesorts* (*ExeSignature* *cto tao sa*)  
**shows** (*term-ok'* (*translate-signature* (*ExeSignature* *cto tao sa*)) *p*  $\wedge$  *typ-of* *p* = *Some propT*)  
 = (*exeterm-ok'* (*ExeSignature* *cto tao sa*) *p*  $\wedge$  *typ-of* *p* = *Some propT*)  
 ⟨proof⟩

**lemma** *term-ok-translate-code-step*:

**assumes** *exe-sig-conds* (*ExeSignature* *cto tao sa*)  
**shows** (*term-ok* (*translate-theory* (*ExeTheory* (*ExeSignature* *cto tao sa*) *ax*)) *p*  $\wedge$  *typ-of* *p* = *Some propT*)  
 = (*term-ok'* (*translate-signature* (*ExeSignature* *cto tao sa*)) *p*  $\wedge$  *typ-of* *p* = *Some propT*)  
 ⟨proof⟩

**lemma** *term-ok-theory-cond-code*:

**assumes** *exe-sig-conds* (*ExeSignature* *cto tao sa*)  
**shows** ( $\forall p \in \text{set } ax . \text{term-ok} (\text{translate-theory} (\text{ExeTheory} (\text{ExeSignature } cto \text{ tao } sa) \text{ ax})) p \wedge \text{typ-of } p = \text{Some } \text{propT}$ )  
 = ( $\forall p \in \text{set } ax . \text{exeterm-ok}' (\text{ExeSignature } cto \text{ tao } sa) p \wedge \text{typ-of } p = \text{Some } \text{propT}$ )  
 ⟨proof⟩

**lemma** *exe-wf-theory-code[code]*:  $\text{exe-wf-theory } \Theta = \text{exe-wf-theory}' \Theta$   
 ⟨proof⟩

**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*  
 $\langle \text{proof} \rangle$

**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*  
 $\langle \text{proof} \rangle$

**definition** *exesort-leq sub s1 s2* =  $(\forall c_2 \in s_2 . \exists c_1 \in s_1. \text{execlass-leq sub } c_1 c_2)$

**lemma** *exesort-leq-code*: *sort-leq (set cs) c1 c2* = *exesort-leq cs c1 c2*  
 $\langle \text{proof} \rangle$

**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*  
 $\langle \text{proof} \rangle$

**lemma** *has-sort-imp-exehas-sort*:

**assumes** *exe-osig-conds (sub, tcs)*  
**shows** *has-sort (translate-osig (sub, tcs)) T S*  $\Longrightarrow$  *exehas-sort (sub, tcs) T S*  
 $\langle \text{proof} \rangle$

**lemma** *has-sort-code*:

**assumes** *exe-osig-conds oss*  
**shows** *has-sort (translate-osig oss) T S* = *exehas-sort oss T S*  
 $\langle \text{proof} \rangle$

**lemma** *has-sort-code'*:

**assumes** *exe-wf-theory'*  $\Theta$   
**shows** *has-sort (osig (sig (translate-theory  $\Theta$ ))) T S*  
 = *exehas-sort (exesorts (exesig  $\Theta$ )) T S*  
 $\langle \text{proof} \rangle$

**abbreviation** *exeinst-ok*  $\Theta$  *insts*  $\equiv$   
*distinct* (*map fst insts*)  
 $\wedge$  *list-all* (*exetyp-ok*  $\Theta$ ) (*map snd insts*)  
 $\wedge$  *list-all* ( $\lambda((idn, S), T) . \text{exehas-sort } (\text{exesorts } (\text{exesig } \Theta)) T S$ ) *insts*

**lemma** *inst-ok-code1*:  
**assumes** *exe-wf-theory'*  $\Theta$   
**shows** *list-all* (*exetyp-ok*  $\Theta$ ) (*map snd insts*) = *list-all* (*typ-ok* (*translate-theory*  $\Theta$ )) (*map snd insts*)  
 $\langle$ *proof* $\rangle$

**lemma** *inst-ok-code2*:  
**assumes** *exe-wf-theory'*  $\Theta$   
**shows** *list-all* ( $\lambda((idn, S), T) . \text{has-sort } (\text{osig } (\text{sig } (\text{translate-theory } \Theta))) T S$ ) *insts*  
= *list-all* ( $\lambda((idn, S), T) . \text{exehas-sort } (\text{exesorts } (\text{exesig } \Theta)) T S$ ) *insts*  
 $\langle$ *proof* $\rangle$

**lemma** *inst-ok-code*:  
**assumes** *exe-wf-theory'*  $\Theta$   
**shows** *inst-ok* (*translate-theory*  $\Theta$ ) *insts* = *exeinst-ok*  $\Theta$  *insts*  
 $\langle$ *proof* $\rangle$

**definition** [*simp*]: *exeterm-ok*  $\Theta$  *t*  $\equiv$  *exeterm-ok'* (*exesig*  $\Theta$ ) *t*  $\wedge$  *typ-of* *t*  $\neq$  *None*

**lemma** *term-ok-code*:  
**assumes** *exe-wf-theory'*  $\Theta$   
**shows** *term-ok* (*translate-theory*  $\Theta$ ) *t* = *exeterm-ok*  $\Theta$  *t*  
 $\langle$ *proof* $\rangle$

**fun** *exereplay'* :: *exetheory*  $\Rightarrow$  (*variable*  $\times$  *typ*) *list*  $\Rightarrow$  *variable set*  
 $\Rightarrow$  *term list*  $\Rightarrow$  *proofterm*  $\Rightarrow$  *term option* **where**  
*exereplay'* *thy* - - *Hs* (*PA $\lambda$ 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* (*PBound n*) = *partial-nth* *Hs n*  
| *exereplay'* *thy vs ns Hs* (*Abst T p*) = (*if* *exetyp-ok* *thy T*  
*then* (*let* (*s', ns'*) = *variant-variable* (*Free STR "default"*) *ns in*  
*map-option* (*mk-all s' T*) (*exereplay'* *thy* ((*s', T*) # *vs*) *ns' Hs p*)  
*else* *None*)  
| *exereplay'* *thy vs ns Hs* (*Appt p t*) =  
(*let* *rep* = *exereplay'* *thy vs ns Hs p in*  
*let* *t'* = *subst-bvs* (*map* ( $\lambda(x,y) . \text{Fv } x y$ ) *vs*) *t in*  
*case* (*rep, typ-of t'*) *of*  
(*Some* (*Ct s* (*Ty fun1* [*Ty fun2* [ $\tau$ , *Ty propT1 Nil*], *Ty propT2 Nil*) \$ *b*),  
*Some*  $\tau'$ )  $\Rightarrow$   
*if* *s* = *STR "Pure.all"*  $\wedge$  *fun1* = *STR "fun"*  $\wedge$  *fun2* = *STR "fun"*  
 $\wedge$  *propT1* = *STR "prop"*  $\wedge$  *propT2* = *STR "prop"*

$\wedge \tau = \tau' \wedge \text{exeterm-ok } \text{thy } t'$   
then *Some* ( $b \cdot t'$ ) else *None*  
| -  $\Rightarrow$  *None*)  
| *exereplay'* *thy vs ns Hs* (*AbsP*  $t$   $p$ ) =  
(let  $t' = \text{subst-bvs } (\text{map } (\lambda(x,y) . \text{Fv } x \ y) \ \text{vs}) \ t$  in  
let  $\text{rep} = \text{exereplay}' \ \text{thy vs ns } (t' \# \text{Hs}) \ p$  in  
(if  $\text{typ-of } t' = \text{Some propT} \wedge \text{exeterm-ok } \text{thy } t'$  then  $\text{map-option } (\text{mk-imp } t')$   
 $\text{rep}$  else *None*))  
| *exereplay'* *thy vs ns Hs* (*AppP*  $p1$   $p2$ ) =  
(let  $\text{rep1} = \text{Option.bind } (\text{exereplay}' \ \text{thy vs ns } \text{Hs } p1) \ \text{beta-eta-norm}$  in  
let  $\text{rep2} = \text{Option.bind } (\text{exereplay}' \ \text{thy vs ns } \text{Hs } p2) \ \text{beta-eta-norm}$  in  
(case ( $\text{rep1}, \text{rep2}$ ) of (  
*Some* (*Ct imp* (*Ty fn1* [*Ty prp1* [], *Ty fn2* [*Ty prp2* [], *Ty prp3* []]) \$  $A$  \$  $B$ ),  
*Some*  $A'$ )  $\Rightarrow$   
if  $\text{imp} = \text{STR "Pure.imp"} \wedge \text{fn1} = \text{STR "fun"} \wedge \text{fn2} = \text{STR "fun"}$   
 $\wedge \text{prp1} = \text{STR "prop"} \wedge \text{prp2} = \text{STR "prop"} \wedge \text{prp3} = \text{STR "prop"} \wedge$   
 $A = A'$   
then *Some*  $B$  else *None*  
| -  $\Rightarrow$  *None*))  
| *exereplay'* *thy vs ns Hs* (*OfClass* *ty c*) = (if  $\text{exehas-sort } (\text{exesorts } (\text{exesig } \text{thy})) \ \text{ty}$   
 $\{c\}$   
 $\wedge \text{exetyp-ok } \text{thy } \text{ty}$   
then (case  $\text{lookup } (\lambda k. k = \text{const-of-class } c) (\text{execonst-type-of } (\text{exesig } \text{thy}))$  of  
*Some* (*Ty fun* [*Ty it* [*ity*], *Ty prop* []])  $\Rightarrow$   
if  $\text{ity} = \text{variable STR "a"} \wedge \text{fun} = \text{STR "fun"} \wedge \text{prop} = \text{STR "prop"} \wedge \text{it}$   
 $= \text{STR "itself"}$   
then *Some* (*mk-of-class* *ty c*) else *None* | -  $\Rightarrow$  *None*) else *None*)  
| *exereplay'* *thy vs ns Hs* (*Hyp*  $t$ ) = (if  $t \in \text{set } \text{Hs}$  then *Some*  $t$  else *None*)

**lemma** *of-class-code1*:

**assumes** *exe-wf-theory'* *thy*  
**shows** ( $\text{has-sort } (\text{osig } (\text{sig } (\text{translate-theory } \text{thy}))) \ \text{ty } \{c\} \wedge \text{typ-ok } (\text{translate-theory } \text{thy}) \ \text{ty}$ )  
= ( $\text{exehas-sort } (\text{exesorts } (\text{exesig } \text{thy})) \ \text{ty } \{c\} \wedge \text{exetyp-ok } \text{thy } \text{ty}$ )  
⟨*proof*⟩

**lemma** *of-class-code2*:

**assumes** *exe-wf-theory'* *thy*  
**shows**  $\text{const-type } (\text{sig } (\text{translate-theory } \text{thy})) (\text{const-of-class } c)$   
=  $\text{lookup } (\lambda k. k = \text{const-of-class } c) (\text{execonst-type-of } (\text{exesig } \text{thy}))$   
⟨*proof*⟩

**lemma** *replay'-code*:

**assumes** *exe-wf-theory'* *thy*  
**shows**  $\text{replay}' (\text{translate-theory } \text{thy}) \ \text{vs ns Hs } P = \text{exereplay}' \ \text{thy vs ns Hs } P$   
⟨*proof*⟩

**abbreviation**  $\text{exereplay}'' \ \text{thy vs ns Hs } P \equiv \text{Option.bind } (\text{exereplay}' \ \text{thy vs ns Hs } P) \ \text{beta-eta-norm}$



**lemma** *replay''-code*:  
**assumes** *exe-wf-theory' thy*  
**shows** *replay'' (translate-theory thy) vs ns Hs P = exereplay'' thy vs ns Hs P*  
*<proof>*

**definition** [*simp*]: *exereplay thy P*  $\equiv$   
*(if  $\forall x \in \text{set (hyps P)}$  . *exeterm-ok thy x*  $\wedge$  *typ-of x = Some propT* then*  
*exereplay'' thy [] (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*  
*<proof>*

**definition** *exe-replay' e P = exereplay'' e [] (fst ' fv-Proof P) [] 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*  
*<proof>*

**lemma** *check-proof-sound*:  
**shows** *exe-check-proof e P res  $\implies$  translate-theory e, set (hyps P)  $\vdash$  res*  
*<proof>*

**lemma** *check-proof-really-sound*:  
**shows** *exe-check-proof e P res  $\implies$  translate-theory e, set (hyps P)  $\Vdash$  res*  
*<proof>*

**end**

## 16 Code Generation

**theory** *CodeGen*  
**imports** *ProofTerm TheoryExe CheckerExe Instances*  
*HOL-Library.Code-Target-Int*  
*HOL-Library.Code-Target-Nat*  
**begin**

**declare** *typ-of-def[code]*

**export-code** *exe-check-proof exereplay exe-wf-theory*  
*Bv PBound Tv Free ExeTheory ExeSignature*  
**in** *SML module-name ExportCheck file-prefix export*

end

## References

- [1] S. Berghofer and T. Nipkow. Proof terms for simply typed higher order logic. In J. Harrison and M. Aagaard, editors, *Theorem Proving in Higher Order Logics*, volume 1869 of *Lect. Notes in Comp. Sci.*, pages 38–52. Springer, 2000.
- [2] T. Nipkow and S. Roßkopf. Isabelle’s metalogic: Formalization and proof checker. In G. S. A. Platzer, editor, *28th International Conference on Automated Deduction (CADE-28)*, *Lect. Notes in Comp. Sci.* Springer, 2021.
- [3] L. C. Paulson. The foundation of a generic theorem prover. *J. Automated Reasoning*, 5:363–397, 1989.
- [4] M. Wenzel. The isabelle/isar implementation. <https://isabelle.in.tum.de/doc/implementation.pdf>.
- [5] M. Wenzel. Type classes and overloading in higher-order logic. In E. L. Gunter and A. P. Felty, editors, *Theorem Proving in Higher Order Logics, TPHOLs’97*, volume 1275 of *Lect. Notes in Comp. Sci.*, pages 307–322. Springer, 1997.