

Gradual Typing: Isabelle/Isar Formalization

Jeremy Siek and Walid Taha

April 7, 2006

theory *GradualTyping* = *Main* + *LaTeXsugar* + *OptionalSugar*:

datatype *base-ty*
= *IntT* (*int*)
| *BoolT* (*bool*)

datatype *ty*
= *Arrow ty ty* (**infixl** \rightarrow 60)
| *BaseT base-ty* (-)
| *UnknownT* (?)

datatype *const*
= *IntC int* (-)
| *BoolC bool* (-)
| *Succ* (**succ**)
| *Prev* (**prev**)
| *IsZero* (**iszero**)

datatype *expr*
= *Var nat* (- [101] 100)
| *Const const* (- [101] 100)
| *Lam nat ty expr* (λ -- -- [53,53,53] 52)
| *App expr expr* (- - [53,53] 52)
| *Cast ty expr* (\langle -) - [53,53] 52)

types *env* = *nat* \Rightarrow *ty option*

consts *typeof* :: *const* \Rightarrow *ty*

primrec

tyint: *typeof* (*IntC n*) = (*BaseT IntT*)
tybool: *typeof* (*BoolC b*) = (*BaseT BoolT*)
tysucc: *typeof Succ* = ((*BaseT IntT*) \rightarrow (*BaseT IntT*))
typrev: *typeof Prev* = ((*BaseT IntT*) \rightarrow (*BaseT IntT*))
tyiszero: *typeof IsZero* = ((*BaseT IntT*) \rightarrow (*BaseT BoolT*))

syntax *typeof* :: *const* \Rightarrow *ty* \Rightarrow *bool* (Σ - = - [100,100] 101)

translations *typeof*- *c* τ == *typeof c* = τ

consts *STLC-type* :: *ty set*

inductive *STLC-type intros*

STBase[*intro!*]: *BaseT b* \in *STLC-type*
STArrow[*intro!*]: $\llbracket \tau_1 \in \text{STLC-type}; \tau_2 \in \text{STLC-type} \rrbracket \Longrightarrow$
 $(\tau_1 \rightarrow \tau_2) \in \text{STLC-type}$

inductive-cases *STLC-arrow-inv*[*elim!*]:

$(\tau_1 \rightarrow \tau_2) \in \text{STLC-type}$

inductive-cases *STLC-top-inv*[*elim!*]:

? \in *STLC-type*

consts *STLC-term* :: *expr set*

inductive STLC-term intros

STTVar[intro!]: $\text{Var } x \in \text{STLC-term}$

STTConst[intro!]: $\text{Const } c \in \text{STLC-term}$

STTLam[intro!]: $\llbracket \tau \in \text{STLC-type}; e \in \text{STLC-term} \rrbracket \implies$

$\text{Lam } x \tau e \in \text{STLC-term}$

STTApp[intro!]: $\llbracket e_1 \in \text{STLC-term}; e_2 \in \text{STLC-term} \rrbracket \implies$

$(\text{App } e_1 e_2) \in \text{STLC-term}$

inductive-cases STLC-lam-inv[elim!]:

$\text{Lam } x \tau e \in \text{STLC-term}$

inductive-cases STLC-app-inv[elim!]:

$\text{App } e_1 e_2 \in \text{STLC-term}$

inductive-cases STLC-down-inv[elim!]:

$\text{Cast } \tau e \in \text{STLC-term}$

lemma STBool: $\text{BaseT } (\text{BoolT}) \in \text{STLC-type}$ **by auto**

lemma STInt: $\text{BaseT } (\text{IntT}) \in \text{STLC-type}$ **by auto**

syntax just :: 'a \Rightarrow 'a option ($\llbracket _ \rrbracket$ [52] 501)

translations $\llbracket \tau \rrbracket == \text{Some } \tau$

consts compatible :: (ty \times ty) set

syntax compatible :: ty \Rightarrow ty \Rightarrow bool (**infix** \sim 51)

translations $\tau_1 \sim \tau_2 == (\tau_1, \tau_2) \in \text{compatible}$

inductive compatible intros

CRef[intro!]: $\tau \sim \tau$

CFun[intro!]: $\llbracket \sigma_1 \sim \tau_1; \sigma_2 \sim \tau_2 \rrbracket \implies (\sigma_1 \rightarrow \sigma_2) \sim (\tau_1 \rightarrow \tau_2)$

CUnR[intro!]: $\tau \sim ?$

CUnL[intro!]: $? \sim \tau$

inductive-cases compat-un-l[elim!]:

$? \sim \tau$

inductive-cases compat-fun-r[elim!]:

$\tau' \sim (\sigma \rightarrow \tau)$

inductive-cases compat-fun-l[elim!]:

$(\sigma \rightarrow \tau) \sim \tau'$

inductive-cases compat-fun[elim!]:

$(s_1 \rightarrow s_2) \sim (t_1 \rightarrow t_2)$

inductive-cases compat-int[elim!]:

$\tau \sim (\text{BaseT } \text{IntT})$

inductive-cases compat-base-l[elim!]:

$\text{BaseT } b \sim \tau$

inductive-cases compat-base-r[elim!]:

$\tau \sim \text{BaseT } b$

lemma compatible-symmetric: $\sigma \sim \tau \implies \tau \sim \sigma$

apply (induct rule: compatible.induct) **by auto**

lemma compatible-reflexive: $\tau \sim \tau$

apply (cases τ) **by auto**

lemma compatible-not-trans:

$\neg (\forall \tau_1 \tau_2 \tau_3. \tau_1 \sim \tau_2 \wedge \tau_2 \sim \tau_3 \longrightarrow \tau_1 \sim \tau_3)$
proof –
have $A: \text{BaseT IntT} \sim ?$ **by auto**
have $B: ? \sim \text{BaseT BoolT}$ **by auto**
have $C: \neg (\text{BaseT IntT} \sim \text{BaseT BoolT})$ **by auto**
from $A B C$ **show** $?thesis$ **by auto**
qed

lemma compatible-stlc-noteq:
 $\sigma \sim \tau \implies \sigma \in \text{STLC-type} \wedge \sigma \neq \tau \longrightarrow \tau \notin \text{STLC-type}$
apply (*induct rule: compatible.induct*) **by auto**

lemma compatible-stlc-eq:
 $\tau \sim \tau' \implies \tau \in \text{STLC-type} \wedge \tau' \in \text{STLC-type} \longrightarrow \tau = \tau'$
apply (*erule compatible.induct*) **by auto**

consts gradual-typing :: (*env* × *expr* × *ty*) *set*
syntax gradual-typing :: *env* ⇒ *expr* ⇒ *ty* ⇒ *bool* ($-\vdash_G - : -$ [52,52,52] 51)
translations
 $\Gamma \vdash_G e : \tau \iff (\Gamma, e, \tau) \in \text{gradual-typing}$
inductive gradual-typing intros
 $GVar[\text{intro!}]: \Gamma x = [\tau] \implies \Gamma \vdash_G \text{Var } x : \tau$
 $GConst[\text{intro!}]: \text{typeof- } c \tau \implies \Gamma \vdash_G \text{Const } c : \tau$
 $GLam[\text{intro!}]: \Gamma(x \mapsto \sigma) \vdash_G e : \tau \implies$
 $\Gamma \vdash_G \text{Lam } x \sigma e : (\sigma \rightarrow \tau)$
 $GApp1[\text{intro!}]: [\Gamma \vdash_G e_1 : ?; \Gamma \vdash_G e_2 : \tau_2] \implies$
 $\Gamma \vdash_G (\text{App } e_1 e_2) : ?$
 $GApp2[\text{intro!}]: [\Gamma \vdash_G e_1 : (\tau \rightarrow \tau'); \Gamma \vdash_G e_2 : \tau_2; \tau_2 \sim \tau] \implies$
 $\Gamma \vdash_G (\text{App } e_1 e_2) : \tau'$
 $GCast[\text{intro!}]: [\Gamma \vdash_G e : \sigma; \sigma \sim \tau] \implies$
 $\Gamma \vdash_G \text{Cast } \tau e : \tau$

consts STLC-wt :: (*env* × *expr* × *ty*) *set*
syntax STLC-wt :: *env* ⇒ *expr* ⇒ *ty* ⇒ *bool* ($-\vdash_{\rightarrow} - : -$ [52,52,52] 51)
translations
 $\Gamma \vdash_{\rightarrow} e : \tau \iff (\Gamma, e, \tau) \in \text{STLC-wt}$
inductive STLC-wt intros
 $STVar[\text{intro!}]: \Gamma(x) = [\tau] \implies \Gamma \vdash_{\rightarrow} \text{Var } x : \tau$
 $STConst[\text{intro!}]: \text{typeof- } c \tau \implies \Gamma \vdash_{\rightarrow} \text{Const } c : \tau$
 $STLam[\text{intro!}]: \Gamma(x \mapsto \sigma) \vdash_{\rightarrow} e : \tau \implies \Gamma \vdash_{\rightarrow} (\lambda x:\sigma. e) : (\sigma \rightarrow \tau)$
 $STApp[\text{intro!}]: [\Gamma \vdash_{\rightarrow} e_1 : (\sigma \rightarrow \tau); \Gamma \vdash_{\rightarrow} e_2 : \sigma] \implies$
 $\Gamma \vdash_{\rightarrow} \text{App } e_1 e_2 : \tau$

lemma gradual-complete-stlc: $\Gamma \vdash_{\rightarrow} e : \tau \implies \Gamma \vdash_G e : \tau$
proof (*induct rule: STLC-wt.induct*)
fix $\Gamma::\text{env}$ **and** τ **and** x
assume $\Gamma x = [\tau]$ **thus** $\Gamma \vdash_G \text{Var } x : \tau$ **by auto**
next
fix $\Gamma::\text{env}$ **and** τ c **assume** *typeof- c* τ **thus** $\Gamma \vdash_G \text{Const } c : \tau$ **by auto**
next

fix $\Gamma::env$ **and** σ **and** τ **and** e **and** x
assume $\Gamma(x \mapsto \sigma) \vdash_G e : \tau$
thus $\Gamma \vdash_G Lam\ x\ \sigma\ e : (\sigma \rightarrow \tau)$ **by** *auto*
next
fix $\Gamma::env$ **and** $\tau 1$ **and** $\tau 1'$ **and** $e 1$ **and** $e 2$
assume $\Gamma \vdash_G e 1 : (\tau 1 \rightarrow \tau 1')$ **and** $\Gamma \vdash_G e 2 : \tau 1$
thus $\Gamma \vdash_G App\ e 1\ e 2 : \tau 1'$ **by** *auto*
qed

lemma *constant-stlc*: *typeof* $c \in STLC\text{-type}$ **by** (*cases* c , *auto*)

lemma *gradual-soundness-stlc*:

$\Gamma \vdash_G e : \tau \implies$
 $e \in STLC\text{-term} \wedge (\forall x\ \tau. \Gamma\ x = [\tau] \implies \tau \in STLC\text{-type}) \implies$
 $\Gamma \vdash_{\rightarrow} e : \tau \wedge \tau \in STLC\text{-type}$
(is $\Gamma \vdash_G e : \tau \implies ?P\ \Gamma\ e\ \tau$ *)*

proof (*induct* rule: *gradual-typing.induct*)

fix $\Gamma::env$ **and** τ **and** x
assume $\Gamma\ x = [\tau]$ **thus** $?P\ \Gamma\ (Var\ x)\ \tau$ **by** *auto*

next

fix $\Gamma::env$ **and** τ **and** c **assume** *typeof*- $c\ \tau$
thus $?P\ \Gamma\ (Const\ c)\ \tau$
using *constant-stlc* **by** *auto*

next

fix $\Gamma::env$ **and** σ **and** τ **and** e **and** x
assume $IH: ?P\ (\Gamma(x \mapsto \sigma))\ e\ \tau$
show $?P\ \Gamma\ (\lambda\ x:\sigma. e)\ (\sigma \rightarrow \tau)$
apply (*rule* *impI*) **apply** (*erule* *conjE*)⁺

proof –

assume $L2: Lam\ x\ \sigma\ e \in STLC\text{-term}$
and $G: \forall\ x\ \tau. \Gamma\ x = [\tau] \implies \tau \in STLC\text{-type}$
from $L2$ **have** $E: e \in STLC\text{-term}$ **by** *auto*
from $L2$ **have** $S: \sigma \in STLC\text{-type}$ **by** *auto*
from $G\ S$ **have** $G2: \forall\ y\ \tau. (\Gamma(x \mapsto \sigma))\ y = [\tau] \implies \tau \in STLC\text{-type}$ **by** *auto*
from $E\ G2\ IH$
have $STLC\text{-wt}\ (\Gamma(x \mapsto \sigma))\ e\ \tau \wedge \tau \in STLC\text{-type}$ **by** *blast*
with S
show $(\Gamma, Lam\ x\ \sigma\ e, \sigma \rightarrow \tau) \in STLC\text{-wt}$
 $\wedge (\sigma \rightarrow \tau) \in STLC\text{-type}$ **by** *auto*

qed

next

fix $\Gamma::env$ **and** $\tau 2\ e 1$ **and** $e 2$
assume $IH: ?P\ \Gamma\ e 1\ ?$
thus $?P\ \Gamma\ (App\ e 1\ e 2)\ ?$ **by** *auto*

next

fix $\Gamma\ \tau\ \tau'\ \tau 2\ e 1\ e 2$
assume $IH1: ?P\ \Gamma\ e 1\ (\tau \rightarrow \tau')$
and $IH2: ?P\ \Gamma\ e 2\ \tau 2$
and $T2T: \tau 2 \sim \tau$
show $?P\ \Gamma\ (App\ e 1\ e 2)\ \tau'$

```

apply (rule impI) apply (erule conjE)+
proof -
  assume A: App e1 e2 ∈ STLC-term
  and G:  $\forall x \tau. \Gamma x = \lfloor \tau \rfloor \longrightarrow \tau \in \text{STLC-type}$ 
  from A have e1 ∈ STLC-term by auto
  with G IH1 have E1:  $(\Gamma, e1, \tau \rightarrow \tau') \in \text{STLC-wt}$ 
   $\wedge (\tau \rightarrow \tau') \in \text{STLC-type}$  by blast
  from A have e2 ∈ STLC-term by auto
  with G IH2 have E2:  $(\Gamma, e2, \tau2) \in \text{STLC-wt}$ 
   $\wedge \tau2 \in \text{STLC-type}$  by blast
  from E1 have tst:  $\tau \in \text{STLC-type}$  by auto
  from E2 have t2st:  $\tau2 \in \text{STLC-type}$  by auto
  from T2T tst t2st T2T have tt:  $\tau = \tau2$  using compatible-stlc-eq by blast
  from tt E1 E2
  show  $(\Gamma, \text{App } e1 e2, \tau') \in \text{STLC-wt}$ 
   $\wedge \tau' \in \text{STLC-type}$ 
  by blast
qed
next
  fix  $\Gamma \sigma \tau e$ 
  show  $?P \Gamma (\text{Cast } \tau e) \tau$  by auto
qed
lemma gradual-soundness-stlc:
   $\llbracket \Gamma \vdash_G e : \tau; e \in \text{STLC-term}; (\forall x \tau. \Gamma x = \lfloor \tau \rfloor \longrightarrow \tau \in \text{STLC-type}) \rrbracket$ 
   $\Longrightarrow \Gamma \vdash_{\rightarrow} e : \tau \wedge \tau \in \text{STLC-type}$ 
  using gradual-soundness-stlc apply simp done

theorem equiv-stlc:
   $e \in \text{STLC-term} \Longrightarrow \text{empty} \vdash_G e : \tau = \text{empty} \vdash_{\rightarrow} e : \tau$ 
  apply auto
  using gradual-soundness-stlc apply blast
  using gradual-complete-stlc apply blast
done

consts compile ::  $(\text{env} \times \text{expr} \times \text{expr} \times \text{ty}) \text{ set}$ 
syntax compile ::  $\text{env} \Rightarrow \text{expr} \Rightarrow \text{expr} \Rightarrow \text{ty} \Rightarrow \text{bool} \ (- \vdash - \Rightarrow - : - [52,52,52,52] 51)$ 
translations
   $\Gamma \vdash e \Rightarrow e' : \tau == (\Gamma, e, e', \tau) \in \text{compile}$ 
inductive compile intros
  CVar[intro!]:  $\Gamma x = \lfloor \tau \rfloor \Longrightarrow \Gamma \vdash \text{Var } x \Rightarrow \text{Var } x : \tau$ 
  CConst[intro!]: typeof- c  $\tau \Longrightarrow \Gamma \vdash \text{Const } c \Rightarrow \text{Const } c : \tau$ 
  CLam[intro!]:  $\Gamma(x \mapsto \sigma) \vdash e \Rightarrow e' : \tau \Longrightarrow$ 
   $\Gamma \vdash \text{Lam } x \sigma e \Rightarrow \text{Lam } x \sigma e' : (\sigma \rightarrow \tau)$ 
  CApp1[intro!]:  $\llbracket \Gamma \vdash e_1 \Rightarrow e'_1 : ?; \Gamma \vdash e_2 \Rightarrow e'_2 : \tau_2 \rrbracket \Longrightarrow$ 
   $\Gamma \vdash (\text{App } e_1 e_2) \Rightarrow (\text{App } (\text{Cast } (\tau_2 \rightarrow ?) e'_1) e'_2) : ?$ 
  CApp2[intro!]:  $\llbracket \Gamma \vdash e_1 \Rightarrow e'_1 : (\tau \rightarrow \tau');$ 
   $\Gamma \vdash e_2 \Rightarrow e'_2 : \tau_2; \tau_2 \neq \tau; \tau_2 \sim \tau \rrbracket \Longrightarrow$ 
   $\Gamma \vdash (\text{App } e_1 e_2) \Rightarrow (\text{App } e'_1 (\text{Cast } \tau e'_2)) : \tau'$ 
  CApp3[intro!]:  $\llbracket \Gamma \vdash e_1 \Rightarrow e'_1 : (\tau \rightarrow \tau');$ 
   $\Gamma \vdash e_2 \Rightarrow e'_2 : \tau \rrbracket \Longrightarrow$ 

```

$\Gamma \vdash (App\ e_1\ e_2) \Rightarrow (App\ e'_1\ e'_2) : \tau'$
 $CCast[intro!]: \llbracket \Gamma \vdash e \Rightarrow e' : \sigma; \sigma \sim \tau \rrbracket \Longrightarrow$
 $\Gamma \vdash Cast\ \tau\ e \Rightarrow Cast\ \tau\ e' : \tau$

inductive-cases *compile-var-inv*[*elim!*]:
 $\Gamma \vdash Var\ x \Rightarrow e : \tau$

inductive-cases *compile-const-inv*[*elim!*]:
 $\Gamma \vdash Const\ c \Rightarrow e : \tau$

inductive-cases *compile-lam-inv*[*elim!*]:
 $\Gamma \vdash Lam\ x\ \sigma\ e \Rightarrow e' : \tau$

inductive-cases *compile-app-inv*[*elim!*]:
 $\Gamma \vdash App\ e1\ e2 \Rightarrow e : \tau$

theorem *complete-stlc*: $\Gamma \vdash_{\rightarrow} e : \tau \Longrightarrow \Gamma \vdash e \Rightarrow e : \tau$
proof (*induct rule: STLC-wt.induct*)
fix $\Gamma::env$ **and** τ **and** x
assume $\Gamma\ x = [\tau]$ **thus** $\Gamma \vdash Var\ x \Rightarrow Var\ x : \tau$ **by** *auto*
next
fix $\Gamma::env$ **and** τ **and** c **assume** *typeof- c* τ
thus $\Gamma \vdash Const\ c \Rightarrow Const\ c : \tau$ **by** *auto*
next
fix $\Gamma::env$ **and** σ **and** τ **and** e **and** x
assume $(\Gamma(x \mapsto \sigma), e, e, \tau) \in compile$
thus $(\Gamma, Lam\ x\ \sigma\ e, Lam\ x\ \sigma\ e, (\sigma \rightarrow \tau)) \in compile$ **by** *auto*
next
fix $\Gamma::env$ **and** $\tau1$ **and** $\tau1'$ **and** $e1$ **and** $e2$
assume $(\Gamma, e1, e1, (\tau1 \rightarrow \tau1')) \in compile$
and $(\Gamma, e2, e2, \tau1) \in compile$
thus $(\Gamma, App\ e1\ e2, App\ e1\ e2, \tau1') \in compile$ **by** *auto*
qed

lemma *soundness-stlc-*:
 $\Gamma \vdash e \Rightarrow e' : \tau \Longrightarrow$
 $e \in STLC-term \wedge (\forall x\ \tau. \Gamma\ x = [\tau] \longrightarrow \tau \in STLC-type) \longrightarrow$
 $e = e' \wedge \Gamma \vdash_{\rightarrow} e : \tau \wedge \tau \in STLC-type$
(is $\Gamma \vdash e \Rightarrow e' : \tau \Longrightarrow ?P\ \Gamma\ e\ e'\ \tau)$

proof (*induct rule: compile.induct*)
fix $\Gamma::env$ **and** τ **and** x
assume $\Gamma\ x = [\tau]$ **thus** $?P\ \Gamma\ (Var\ x)\ (Var\ x)\ \tau$ **by** *auto*
next
fix $\Gamma::env$ **and** τ **and** c **assume** *typeof- c* τ
thus $?P\ \Gamma\ (Const\ c)\ (Const\ c)\ \tau$
using *constant-stlc* **by** *auto*
next
fix $\Gamma::env$ **and** σ **and** τ **and** e **and** e' **and** x
assume *IH*: $?P\ (\Gamma(x \mapsto \sigma))\ e\ e'\ \tau$
show $?P\ \Gamma\ (\lambda\ x:\sigma. e)\ (\lambda\ x:\sigma. e')\ (\sigma \rightarrow \tau)$
apply (*rule impI*) **apply** (*erule conjE*)
proof –
assume *L2*: $Lam\ x\ \sigma\ e \in STLC-term$
and *G*: $\forall x\ \tau. \Gamma\ x = [\tau] \longrightarrow \tau \in STLC-type$

from $L2$ **have** $E: e \in \text{STLC-term}$ **by** *auto*
from $L2$ **have** $S: \sigma \in \text{STLC-type}$ **by** *auto*
from $G S$ **have** $G2: \forall y \tau. (\Gamma(x \mapsto \sigma)) y = \lfloor \tau \rfloor \longrightarrow \tau \in \text{STLC-type}$ **by** *auto*
from $E G2 IH$
have $e = e' \wedge \text{STLC-wt} (\Gamma(x \mapsto \sigma)) e \tau \wedge \tau \in \text{STLC-type}$ **by** *blast*
with S
show $\text{Lam } x \sigma e = \text{Lam } x \sigma e' \wedge (\Gamma, \text{Lam } x \sigma e, \sigma \rightarrow \tau) \in \text{STLC-wt}$
 $\wedge (\sigma \rightarrow \tau) \in \text{STLC-type}$ **by** *auto*
qed
next
fix $\Gamma::\text{env}$ **and** $\tau2$ **and** $e'1$ **and** $e'2$ **and** $e1$ **and** $e2$
assume $IH: ?P \Gamma e1 e'1 ?$
thus $?P \Gamma (\text{App } e1 e2) (\text{App} (\text{Cast } (\tau2 \rightarrow ?) e'1) e'2) ?$ **by** *auto*
next
fix $\Gamma \tau \tau' \tau2 e'1 e'2 e1 e2$
assume $IH1: ?P \Gamma e1 e'1 (\tau \rightarrow \tau')$
and $IH2: ?P \Gamma e2 e'2 \tau2$
and $\text{NOTT2T}: \tau2 \neq \tau$
and $\text{T2T}: \tau2 \sim \tau$
show $?P \Gamma (\text{App } e1 e2) (\text{App } e'1 (\text{Cast } \tau e'2)) \tau'$
apply (*rule impI*) **apply** (*erule conjE*)
proof –
assume $A: \text{App } e1 e2 \in \text{STLC-term}$
and $G: \forall x \tau. \Gamma x = \lfloor \tau \rfloor \longrightarrow \tau \in \text{STLC-type}$
from A **have** $e1 \in \text{STLC-term}$ **by** *auto*
with $G IH1$ **have** $E1: e1 = e'1 \wedge (\Gamma, e1, \tau \rightarrow \tau') \in \text{STLC-wt}$
 $\wedge (\tau \rightarrow \tau') \in \text{STLC-type}$ **by** *blast*
from A **have** $e2 \in \text{STLC-term}$ **by** *auto*
with $G IH2$ **have** $E2: e2 = e'2 \wedge (\Gamma, e2, \tau2) \in \text{STLC-wt}$
 $\wedge \tau2 \in \text{STLC-type}$ **by** *blast*
from $E1$ **have** $tst: \tau \in \text{STLC-type}$ **by** *auto*
from $E2$ **have** $t2st: \tau2 \in \text{STLC-type}$ **by** *auto*
from $\text{T2T } tst \text{ t2st } \text{T2T}$ **have** $tt: \tau = \tau2$ **using** *compatible-stlc-eq* **by** *blast*
with NOTT2T **have** *False* **by** *simp*
thus $\text{App } e1 e2 = \text{App } e'1 (\text{Cast } \tau e'2) \wedge (\Gamma, \text{App } e1 e2, \tau') \in \text{STLC-wt}$
 $\wedge \tau' \in \text{STLC-type}$ **by** *simp*
qed
next
fix $\Gamma \tau \tau' e'1 e'2 e1 e2$
assume $IH1: ?P \Gamma e1 e'1 (\tau \rightarrow \tau')$
and $IH2: ?P \Gamma e2 e'2 \tau$
thus $?P \Gamma (\text{App } e1 e2) (\text{App } e'1 e'2) \tau'$ **by** *blast*
next
fix $\Gamma \sigma \tau e e'$
show $?P \Gamma (\text{Cast } \tau e) (\text{Cast } \tau e') \tau$ **by** *auto*
qed
theorem *soundness-stlc*:
 $\llbracket \Gamma \vdash e \Rightarrow e' : \tau; e \in \text{STLC-term}; (\forall x \tau. \Gamma x = \lfloor \tau \rfloor \longrightarrow \tau \in \text{STLC-type}) \rrbracket$
 $\Longrightarrow e = e' \wedge \Gamma \vdash_{\rightarrow} e : \tau \wedge \tau \in \text{STLC-type}$
using *soundness-stlc*- **apply** *simp* **done**

consts *welltyped* :: (env × expr × ty) set
syntax *welltyped* :: env ⇒ expr ⇒ ty ⇒ bool (- ⊢ - : - [52,52,52] 51)
translations $\Gamma \vdash e : \tau == (\Gamma, e, \tau) \in \text{welltyped}$
inductive welltyped intros
WTVar[intro!]: $\Gamma(x) = [\tau] \implies \Gamma \vdash \text{Var } x : \tau$
WTConst[intro!]: *typeof*- c $\tau \implies \Gamma \vdash \text{Const } c : \tau$
WTLam[intro!]: $\Gamma(x:\sigma) \vdash e : \tau \implies \Gamma \vdash (\lambda x:\sigma. e) : (\sigma \rightarrow \tau)$
WTApp[intro!]: $\llbracket \Gamma \vdash e_1 : (\tau \rightarrow \tau'); \Gamma \vdash e_2 : \tau \rrbracket \implies$
 $\Gamma \vdash \text{App } e_1 e_2 : \tau'$
WTCast[intro!]: $\llbracket \Gamma \vdash e : \sigma; \sigma \sim \tau \rrbracket \implies \Gamma \vdash \text{Cast } \tau e : \tau$

inductive-cases *welltyped-fun*:
 $\Gamma \vdash e : (\sigma \rightarrow \tau)$
inductive-cases *welltyped-int*:
 $\Gamma \vdash e : (\text{BaseT IntT})$
inductive-cases *welltyped-bool*:
 $\Gamma \vdash e : (\text{BaseT BoolT})$
inductive-cases *welltyped-lam-inv*:
 $\Gamma \vdash (\lambda x:\tau'. e) : \tau$
inductive-cases *welltyped-fun2*:
 $\Gamma \vdash (\lambda x:\tau'. e) : (\sigma \rightarrow \tau)$
inductive-cases *welltyped-var-inv*:
 $\Gamma \vdash (\text{Var } x) : \tau$
inductive-cases *welltyped-var2*:
 $\Gamma \vdash (\text{Var } x) : (\sigma \rightarrow \tau)$
inductive-cases *welltyped-var3*:
 $(\lambda x. \text{None}) \vdash (\text{Var } x) : (\sigma \rightarrow \tau)$
inductive-cases *welltyped-const-inv*:
 $\Gamma \vdash (\text{Const } c) : \tau$
inductive-cases *welltyped-app-inv*:
 $\Gamma \vdash (\text{App } e_1 e_2) : \tau$
inductive-cases *welltyped-cast-inv*:
 $\Gamma \vdash (\text{Cast } \tau e) : \tau'$
lemma *welltyped-var-inv2*: $\Gamma \vdash (\text{Var } x) : \tau \implies \Gamma x = [\tau]$
apply (*rule welltyped-var-inv*) **by** *auto*
lemma *welltyped-const-inv2*: $\Gamma \vdash (\text{Const } c) : \tau \implies \text{typeof- } c \tau$
apply (*rule welltyped-const-inv*) **by** *auto*
lemma *welltyped-lam-inv2*: $\Gamma \vdash (\lambda x:\sigma. e) : \tau \implies (\exists \tau'. \tau = (\sigma \rightarrow \tau'))$
apply (*rule welltyped-lam-inv*) **by** *auto*
lemma *welltyped-app-inv2*: $\Gamma \vdash (\text{App } e_1 e_2) : \tau' \implies$
 $(\exists \tau \tau_2. \Gamma \vdash e_1 : (\tau \rightarrow \tau') \wedge \Gamma \vdash e_2 : \tau)$
apply (*rule welltyped-app-inv*) **by** *auto*
lemma *welltyped-cast-inv2*: $\Gamma \vdash (\text{Cast } \sigma e) : \tau \implies$
 $(\exists \tau'. \Gamma \vdash e : \tau' \wedge \sigma = \tau \wedge \tau' \sim \sigma)$
apply (*rule welltyped-cast-inv*) **by** *auto*

lemma *unique-typing*[*rule-format*]:
 $\Gamma \vdash e : \tau \implies (\forall \tau'. \Gamma \vdash e : \tau' \longrightarrow \tau = \tau')$ (**is** $\Gamma \vdash e : \tau \implies ?P \Gamma e \tau$)
apply (*induct rule: welltyped.induct*)

```

apply (rule allI) apply (rule impI)
  apply (erule welltyped-var-inv) apply simp
apply (rule allI) apply (rule impI)
  apply (erule welltyped-const-inv) apply simp
apply (rule allI) apply (rule impI)
  apply (erule welltyped-lam-inv) apply blast
apply (rule allI) apply (rule impI)
  apply (erule welltyped-app-inv)
  apply (erule-tac x= $\tau \rightarrow \tau'$ a in allE)
  apply (erule-tac x= $\tau''$  in allE)
  apply simp
apply (rule allI) apply (rule impI)
  apply (erule welltyped-cast-inv)
  apply force
done

```

theorem *compilation-sound*: $\Gamma \vdash e \Rightarrow e' : \tau \Longrightarrow \Gamma \vdash e' : \tau$

proof (*induct rule: compile.induct*)

fix $\Gamma::env$ **and** τ **and** x

assume $\Gamma x = [\tau]$

thus $(\Gamma, Var\ x, \tau) \in welltyped$ **by** *auto*

next

fix $\Gamma::env$ **and** τ **and** c **assume** *typeof- c* τ

thus $(\Gamma, Const\ c, \tau) \in welltyped$ **by** *auto*

next

fix $\Gamma::env$ **and** σ **and** τ **and** e **and** e' **and** x

assume $(\Gamma(x \mapsto \sigma), e, \tau) \in welltyped$

thus $(\Gamma, Lam\ x\ \sigma\ e, \sigma \rightarrow \tau) \in welltyped$ **by** *auto*

next

fix $\Gamma\ \tau_2\ e'_1\ e'_2\ e_1\ e_2$

assume $E1: \Gamma \vdash e'_1 : ?$ **and** $E2: \Gamma \vdash e'_2 : \tau_2$

from $E1$ **have** $\Gamma \vdash (Cast\ (\tau_2 \rightarrow ?)\ e'_1) : (\tau_2 \rightarrow ?)$

using *WTCast* **apply** *auto* **done**

with $E2$ **show** $welltyped\ \Gamma\ (App\ (Cast\ (\tau_2 \rightarrow ?)\ e'_1)\ e'_2) ?$

using *WTApp* **by** *simp*

next

fix $\Gamma\ \tau\ \tau'\ \tau_2\ e'_1\ e'_2\ e_1\ e_2$

assume $E1: welltyped\ \Gamma\ e'_1\ (\tau \rightarrow \tau')$ **and** $E2: welltyped\ \Gamma\ e'_2\ \tau_2$ **and** $t2t: \tau_2 \sim \tau$

from $E2\ t2t$ **have** $\Gamma \vdash Cast\ \tau\ e'_2 : \tau$ **using** *WTCast* **by** *auto*

with $E1$ **show** $\Gamma \vdash App\ e'_1\ (Cast\ \tau\ e'_2) : \tau'$ **using** *WTApp* **by** *simp*

next

fix $\Gamma\ \tau\ \tau'\ e'_1\ e'_2\ e_1\ e_2$

assume $E1: welltyped\ \Gamma\ e'_1\ (\tau \rightarrow \tau')$ **and** $E2: welltyped\ \Gamma\ e'_2\ \tau$

from $E1\ E2$ **show** $\Gamma \vdash App\ e'_1\ e'_2 : \tau'$ **using** *WTApp* **by** *simp*

next

fix $\Gamma\ \sigma\ \tau\ e\ e'$

assume $welltyped\ \Gamma\ e'\ \sigma$ **and** $\sigma \sim \tau$

thus $welltyped\ \Gamma\ (Cast\ \tau\ e')\ \tau$ **using** *WTCast* **by** *simp*

qed

consts *maxv* :: *expr* \Rightarrow *nat*

primrec

maxv-var: *maxv* (Var *x*) = *x*
maxv-lam: *maxv* (λ *x*: τ . *e*) = *max* (*maxv e*) *x*
maxv-app: *maxv* (App *e*₁ *e*₂) = *max* (*maxv e*₁) (*maxv e*₂)
maxv-const: *maxv* (Const *c*) = 0
maxv-cast: *maxv* (Cast τ *e*) = *maxv e*

consts *subst* :: (*expr* \times *nat* \times *expr*) \Rightarrow *expr*

syntax *subst* :: *nat* \Rightarrow *expr* \Rightarrow *expr* \Rightarrow *expr* ([:=]- [100,100,100] 101)

translations

$[x:=e]e == \text{subst}(e,x,e')$
recdef (**permissive**) *subst measure* (λ *p*. *size* (*fst p*))
svar: $[x:=e](\text{Var } y) = (\text{if } y = x \text{ then } e \text{ else } \text{Var } y)$
slam: $[x:=e](\lambda y:\tau. e')$ =
 (*let* *z* = (*max* (*max* (*maxv e'*) *x*) (*maxv e*)) + 1 *in*
 $\lambda z:\tau. [x:=e]([y:=\text{Var } z]e')$)
sapp: $[x:=e](\text{App } e_1 e_2) = \text{App } ([x:=e]e_1) ([x:=e]e_2)$
sconst: $[x:=e](\text{Const } c) = (\text{Const } c)$
scast: $[x:=e](\text{Cast } \tau e') = (\text{Cast } \tau ([x:=e]e'))$

lemma *subst-size[simp]*:

$\forall x w e. \text{size } e = n \longrightarrow \text{size } (\text{subst}(e, x, \text{Var } w)) = n$

proof (*induct rule: nat-less-induct*)

fix *n*

assume *IH*: $\forall m < n. \forall x w e. \text{size } e = m \longrightarrow \text{size } (\text{subst}(e, x, \text{Var } w)) = m$

show $\forall x w e. \text{size } e = n \longrightarrow \text{size } (\text{subst}(e, x, \text{Var } w)) = n$

apply (*rule allI*) + **apply** (*rule impI*)

proof –

fix *x* **and** *w* **and** *e*:*expr* **assume** *se*: *size e* = *n*

show *size* (*subst* (*e*, *x*, *Var w*)) = *n*

apply (*cases e*)

using *se* **apply** (*simp add: svar*)

using *se* **apply** (*simp add: sconst*)

prefer 3 **using** *se* **apply** (*simp add: scast*) **using** *IH* **apply** *force*

proof –

fix *x'* τ *e'*

assume *E*: *e* = $\lambda x':\tau. e'$

let *?W* = (*max* (*max* (*maxv e'*) *x*) *w*) + 1

from *E se* **have** *Suc* (*size e'*) = *n* **by** *simp*

with *IH* **have**

EP: *Suc* (*size* (*subst*(*e'*, *x'*, *Var ?W*))) = *n* **by** *auto*

from *se EP E* **have** *EP2*:

size (*subst* (*e'*, *x'*, *Var ?W*)) < *Suc* (*size e'*) **by** *auto*

from *EP IH* **have**

Suc (*size* (*subst*(*subst*(*e'*, *x'*, *Var ?W*), *x*, *Var w*))) = *n* **by** *auto*

with *E EP2* **show** *size* (*subst* (*e*, *x*, *Var w*)) = *n*

by (*simp add: slam*)

next

fix *e1 e2* **assume** *AP*: *e* = App *e1 e2*

```

from AP se have (size e1) < n by auto
with IH have E1: size (subst(e1,x,Var w)) = size e1 by auto
from AP se have (size e2) < n by auto
with IH have E2: size (subst(e2,x,Var w)) = size e2 by auto
from AP E1 E2 have size (subst (e, x, Var w)) = size e
  by (simp add: sapp)
with se
show size (subst (e, x, Var w)) = n by simp
qed
qed
recdef-tc subst (1) by simp
lemmas subst-simps[simp] = subst.simps[simplified]
lemmas subst-lam[simp] = subst-simps(2)
lemma subst-lam2:
  z = (max (max (maxv e') x) (maxv e)) + 1  $\implies$ 
  [x:=e]( $\lambda$  y: $\tau$ . e') = ( $\lambda$  z: $\tau$ . [x:=e][y:=Var z]e') by simp

consts Vars :: expr  $\Rightarrow$  nat set
primrec
  Vars (Var x) = {x}
  Vars ( $\lambda$  x: $\tau$ . e) = insert x (Vars e)
  Vars (App e1 e2) = (Vars e1)  $\cup$  (Vars e2)
  Vars (Const c) = {}
  Vars (Cast  $\tau$  e) = Vars e

lemma expand-env[rule-format]:
   $\Gamma \vdash e : \tau \implies (\forall x \sigma. x \notin \text{Vars } e \longrightarrow \Gamma(x \mapsto \sigma) \vdash e : \tau)$ 
  (is  $\Gamma \vdash e : \tau \implies ?P \Gamma e \tau$ )
  apply (induct rule: welltyped.induct)
  apply force
  apply force
  prefer 2 apply force
  prefer 2 apply force
proof -
  fix  $\Gamma \sigma \tau e x$  assume IH: ?P ( $\Gamma(x \mapsto \sigma)$ ) e  $\tau$ 
  show ?P  $\Gamma (\lambda x:\sigma. e) (\sigma \rightarrow \tau)$  apply (rule allI)+ apply (rule impI)
proof -
  fix x'  $\sigma'$ 
  assume XP: x'  $\notin$  Vars ( $\lambda x:\sigma. e$ )
  from XP have X: x'  $\neq$  x by simp
  from XP have x'  $\notin$  Vars e by simp
  with IH have ET:  $\Gamma(x \mapsto \sigma, x' \mapsto \sigma') \vdash e : \tau$  by blast
  from X have  $\forall y. ((\Gamma(x \mapsto \sigma, x' \mapsto \sigma')) y) = ((\Gamma(x' \mapsto \sigma', x \mapsto \sigma)) y)$ 
  by auto
  hence  $\Gamma(x \mapsto \sigma, x' \mapsto \sigma') = \Gamma(x' \mapsto \sigma', x \mapsto \sigma)$  using ext by blast
  with ET have  $\Gamma(x' \mapsto \sigma', x \mapsto \sigma) \vdash e : \tau$  by simp
  thus  $\Gamma(x' \mapsto \sigma') \vdash (\lambda x:\sigma. e) : (\sigma \rightarrow \tau)$  by (rule WTLam)
qed
qed

```

lemma contract-env-:
 $\Gamma' \vdash e : \tau \implies (\forall \Gamma y \nu. \Gamma' = \Gamma(y \mapsto \nu) \wedge y \notin \text{Vars } e \longrightarrow \Gamma \vdash e : \tau)$
(is $\Gamma' \vdash e : \tau \implies ?P \Gamma' e \tau$)
apply (*induct rule: welltyped.induct*)
apply *force*
apply *force*
prefer 2 **apply** *force*
prefer 2 **apply** *force*
proof –
fix $\Gamma \sigma \tau e x$
assume *ET: welltyped* ($\Gamma(x \mapsto \sigma)$) $e \tau$
and *IH:* $?P (\Gamma(x \mapsto \sigma)) e \tau$
show $?P \Gamma (\lambda x:\sigma. e) (\sigma \rightarrow \tau)$
apply (*rule allI*)**+** **apply** (*rule impI*) **apply** (*erule conjE*)
proof –
fix $\Gamma' y \nu$
assume $G: \Gamma = \Gamma'(y \mapsto \nu)$ **and** $X: y \notin \text{Vars } (\lambda x:\sigma. e)$
from X **have** $YE: y \notin \text{Vars } e$ **by** *simp*
from X **have** $XY: x \neq y$ **by** *simp*
have $GP: \Gamma(x \mapsto \sigma) = \Gamma'(x \mapsto \sigma)(y \mapsto \nu)$
proof –
{ **fix** z **from** $G XY$ **have** $(\Gamma(x \mapsto \sigma)) z = (\Gamma'(x \mapsto \sigma)(y \mapsto \nu)) z$ **by** *simp* }
thus $\Gamma(x \mapsto \sigma) = \Gamma'(x \mapsto \sigma)(y \mapsto \nu)$ **by** (*rule ext*)
qed
with $YE IH$ **have** $XE: \Gamma'(x \mapsto \sigma) \vdash e : \tau$ **by** *blast*
thus $\Gamma' \vdash (\lambda x:\sigma. e) : (\sigma \rightarrow \tau)$ **by** (*rule WTLam*)
qed
qed

lemma contract-env-:
 $\llbracket \Gamma(y \mapsto \nu) \vdash e : \tau; y \notin \text{Vars } e \rrbracket \implies \Gamma \vdash e : \tau$
apply (*frule contract-env-*) **by** *blast*

lemma maxv-ge-vars: $\forall x. x \in \text{Vars } e \longrightarrow x < \text{Suc } (\text{maxv } e)$
apply (*induct e*)
apply *simp*
apply *simp*
apply *auto*
apply *arith*
apply (*erule-tac x=x in allE*)
apply *simp*
apply *arith*
apply (*erule-tac x=x in allE*)
apply *simp*
apply *arith*
apply (*erule-tac x=x in allE*)
apply (*erule-tac x=x in allE*)
apply *simp*
apply *arith*

done

lemma *maxv-vars*: $\text{maxv } e < x \implies x \notin \text{Vars } e$

proof –

assume *EX*: $\text{maxv } e < x$
have $x \in \text{Vars } e \vee x \notin \text{Vars } e$ by *simp*
moreover { assume $x \in \text{Vars } e$
with *maxv-ge-vars* have $x < \text{Suc } (\text{maxv } e)$ by *simp*
with *EX* have *False* by *simp* hence $x \notin \text{Vars } e$ by *simp*
} moreover { assume $x \notin \text{Vars } e$ hence $x \notin \text{Vars } e$. }
ultimately show $x \notin \text{Vars } e$ by *blast*

qed

lemma *substitution-impl*:

$\forall \Gamma e \tau e' \sigma x.$
 $\text{size } e = n \wedge \Gamma(x \mapsto \sigma) \vdash e : \tau \wedge \Gamma \vdash e' : \sigma \longrightarrow$
 $\Gamma \vdash ([x := e']e) : \tau$
(is *?P n*)

proof (*induct rule: nat-less-induct*)

fix *n* assume *IH*: $\forall m < n. ?P m$

show *?P n* apply (*rule allI*) + apply (*rule impI*) apply (*erule conjE*) +

proof –

fix $\Gamma e \tau e' \sigma x$

assume *N*: $\text{size } e = n$ and *ET*: $\Gamma(x \mapsto \sigma) \vdash e : \tau$

and *EP*: $\Gamma \vdash e' : \sigma$

show $\Gamma \vdash ([x := e']e) : \tau$

proof (*cases e*)

fix *y* assume *E*: $e = \text{Var } y$

show $\Gamma \vdash ([x := e']e) : \tau$

proof (*cases x = y*)

assume *XY*: $x = y$

from *XY* have $[x := e'](\text{Var } y) = e'$ by *simp*

with *EP* have *SYT*: $\Gamma \vdash [x := e'](\text{Var } y) : \sigma$ by *simp*

from *ET E XY* have *ST*: $\sigma = \tau$ by (*cases rule: welltyped.cases, auto*)

from *E EP ST XY* show *?thesis* by *auto*

next

assume *XY*: $x \neq y$

from *ET E XY* have *GYT*: $\Gamma(y) = [\tau]$

using *welltyped-var-inv*[of $\Gamma(x \mapsto \sigma) y \tau$] by *auto*

from *GYT* have $\Gamma \vdash (\text{Var } y) : \tau$ by (*rule WTVar*)

with *XY E* have *YT*: $\Gamma \vdash [x := e']e : \tau$ by *simp*

from *YT* show *?thesis* by *blast*

qed

next

fix *c* assume *E*: $e = \text{Const } c$

from *ET E* have *CT*: *typeof*- *c* τ using *welltyped-const-inv* by *auto*

from *CT* have *WT*: $\Gamma \vdash \text{Const } c : \tau$ by *auto*

with *E* show $\Gamma \vdash ([x := e']e) : \tau$ by *auto*

next

fix *y* $\sigma'' e''$ assume *E*: $e = (\lambda y : \sigma''. e'')$

from $ET\ E$ **have** $X: \exists \tau'. (\Gamma(x \mapsto \sigma)(y \mapsto \sigma')) \vdash e'' : \tau' \wedge \tau = (\sigma'' \rightarrow \tau')$
using *welltyped-lam-inv*[of $\Gamma(x \mapsto \sigma)\ y\ \sigma''\ e''\ \tau$] **by** *blast*
from X **obtain** τ' **where** $ETXY: (\Gamma(x \mapsto \sigma)(y \mapsto \sigma')) \vdash e'' : \tau'$
and $TST: \tau = (\sigma'' \rightarrow \tau')$ **by** *blast*
let $?W = \text{Suc}(\text{max}(\text{max}(\text{maxv } e'')\ x)\ (\text{maxv } e'))$
let $?M = \text{size } e''$
have $M: \text{size } e'' = ?M$ **by** *simp*
from $E\ N$ **have** $MN: ?M < n$ **by** *simp*

have $\text{maxv } e'' < ?W$ **by** *arith*
hence $WV: ?W \notin \text{Vars } e''$ **by** (*rule maxv-vars*)
from $ETXY\ WV$ **have** $ETXYW: \Gamma(x \mapsto \sigma, y \mapsto \sigma'', ?W \mapsto \sigma') \vdash e'' : \tau'$
by (*rule expand-env*)
have $\Gamma(x \mapsto \sigma, y \mapsto \sigma'', ?W \mapsto \sigma') = \Gamma(?W \mapsto \sigma'', x \mapsto \sigma, y \mapsto \sigma')$
proof –
{ **fix** z
have $(\Gamma(x \mapsto \sigma, y \mapsto \sigma'', ?W \mapsto \sigma'))\ z = (\Gamma(?W \mapsto \sigma'', x \mapsto \sigma, y \mapsto \sigma'))\ z$
apply *auto* **apply** *arith* **apply** *arith* **done** }
thus $\Gamma(x \mapsto \sigma, y \mapsto \sigma'', ?W \mapsto \sigma') = \Gamma(?W \mapsto \sigma'', x \mapsto \sigma, y \mapsto \sigma')$
by (*rule ext*)
qed

with $ETXYW$ **have** $ET2: \Gamma(?W \mapsto \sigma'', x \mapsto \sigma, y \mapsto \sigma') \vdash e'' : \tau'$ **by** *simp*
from $WTVar$ **have** $WT: \Gamma(?W \mapsto \sigma'', x \mapsto \sigma) \vdash \text{Var } ?W : \sigma''$ **by** (*auto, arith*)
from $MN\ IH\ M\ ET2\ WT$
obtain τ'' **where** $YE: \Gamma(?W \mapsto \sigma'', x \mapsto \sigma) \vdash [y := \text{Var } ?W]e'' : \tau''$
and $tpptp: \tau'' = \tau'$ **by** *blast*
have $\text{maxv } e' < ?W$ **by** *arith*
hence $WV: ?W \notin \text{Vars } e'$ **by** (*rule maxv-vars*)
from $EP\ WV$ **have** $EP2: \Gamma(?W \mapsto \sigma'') \vdash e' : \sigma$ **by** (*rule expand-env*)
let $?M2 = \text{size}([y := \text{Var } ?W]e'')$
from $E\ N$ **have** $MN2: ?M2 < n$ **by** *auto*
from $MN2\ IH\ YE\ EP2$
obtain $\tau 1$ **where** $EPPT1: \Gamma(?W \mapsto \sigma'') \vdash [x := e'] [y := \text{Var } ?W]e'' : \tau 1$
and $T1TP: \tau 1 = \tau''$ **by** *blast*
from $EPPT1$ **have**
 $LT1: \Gamma \vdash (\lambda ?W: \sigma''. [x := e'] [y := \text{Var } ?W]e'') : (\sigma'' \rightarrow \tau 1)$
by (*rule WTLam*)
from E **have** $[x := e']e = (\lambda ?W: \sigma''. [x := e'] [y := \text{Var } ?W]e'')$ **by** *simp*
with $LT1$ **have** $EST: \Gamma \vdash ([x := e']e) : (\sigma'' \rightarrow \tau 1)$ **by** *simp*
from $T1TP\ tpptp$ **have** $t1tp: \tau 1 = \tau'$ **by** *simp*
from $t1tp\ TST$ **have** $SPTT: (\sigma'' \rightarrow \tau 1) = \tau$ **by** *auto*
from $EST\ SPTT$ **show** *?thesis* **by** *blast*

next
fix $e1\ e2$ **assume** $E: e = \text{App } e1\ e2$
from $ET\ E$ **have** $AT: \Gamma(x \mapsto \sigma) \vdash \text{App } e1\ e2 : \tau$ **by** *simp*
from AT **obtain** t **where** $E1T: \Gamma(x \mapsto \sigma) \vdash e1 : (t \rightarrow \tau)$
and $E2T: \Gamma(x \mapsto \sigma) \vdash e2 : t$ **by** (*rule welltyped-app-inv*)
let $?M1 = \text{size } e1$ **have** $M1: \text{size } e1 = ?M1$ **by** *simp*
from $E\ N$ **have** $M1N: ?M1 < n$ **by** *simp*
from $M1N\ M1\ IH\ E1T\ EP$

```

obtain  $\tau 1$  where  $E1T1: \Gamma \vdash [x:=e]e1 : \tau 1$  and  $T1TP: \tau 1 = (t \rightarrow \tau)$ 
  by blast
let  $?M2 = \text{size } e2$  have  $M2: \text{size } e2 = ?M2$  by simp
from  $E N$  have  $M2N: ?M2 < n$  by simp
from  $M2N M2 IH E2T EP$  have  $E2T2: \Gamma \vdash [x:=e]e2 : t$  by blast
from  $E1T1 T1TP E2T2 E$ 
show  $\Gamma \vdash [x:=e]e : \tau$  using WTAApp by auto
next
fix  $\tau' e''$  assume  $E: e = \text{Cast } \tau' e''$ 
from  $ET E$  obtain  $\tau''$  where  $eppt: \Gamma(x \mapsto \sigma) \vdash e'' : \tau''$  and  $tpt: \tau' = \tau$ 
  and  $tpptp: \tau'' \sim \tau'$  using welltyped-cast-inv by blast
let  $?M = \text{size } e''$ 
have  $M: \text{size } e'' = ?M$  by simp
from  $E N$  have  $MN: ?M < n$  by simp
from  $MN IH M eppt EP$  have  $eppt1: \Gamma \vdash [x:=e]e'' : \tau''$  by blast
from  $eppt1 tpptp E$  have  $et: \Gamma \vdash [x:=e]e : \tau'$  using WTCast by auto
with  $tpt$  show ?thesis by auto
qed
qed
qed

```

lemma *substitution*:

```

 $\llbracket \Gamma(x \mapsto \sigma) \vdash e : \tau; \Gamma \vdash e' : \sigma \rrbracket \implies \Gamma \vdash ([x:=e]e) : \tau$ 
using substitution-impl by blast

```

constdefs *ConstFun* :: *const set*

ConstFun $\equiv \{ \text{Succ}, \text{Prev}, \text{IsZero} \}$

declare *ConstFun-def*[*simp*]

consts *SimpleFunVal* :: *expr set*

inductive *SimpleFunVal* **intros**

SFLam[*intro!*]: $(\lambda x:\tau. e) \in \text{SimpleFunVal}$

SFConst[*intro!*]: $c \in \text{ConstFun} \implies (\text{Const } c) \in \text{SimpleFunVal}$

inductive-cases *app-sfval*[*elim!*]:

App $e e' \in \text{SimpleFunVal}$

inductive-cases *cast-sfval*[*elim!*]:

Cast $\tau e \in \text{SimpleFunVal}$

consts *SimpleValues* :: *expr set*

inductive *SimpleValues* **intros**

SVar[*intro!*]: $\text{Var } x \in \text{SimpleValues}$

SConst[*intro!*]: $\text{Const } c \in \text{SimpleValues}$

SLam[*intro!*]: $(\lambda x:\tau. e) \in \text{SimpleValues}$

inductive-cases *app-sval*[*elim!*]:

App $e e' \in \text{SimpleValues}$

inductive-cases *cast-sval*[*elim!*]:

Cast $\tau e \in \text{SimpleValues}$

consts *FunVal* :: *expr set*

inductive *FunVal* **intros**

$FSimple[intro!]: v \in SimpleFunVal \implies v \in FunVal$
 $FCast[intro!]: v \in SimpleFunVal \implies (Cast ? v) \in FunVal$
inductive-cases *app-fval[elim!]*:
 $App e e' \in FunVal$
inductive-cases *cast-fval[elim!]*:
 $Cast \tau e \in FunVal$

consts *Values* :: *expr set*
inductive *Values intros*
 $VSimple[intro!]: v \in SimpleValues \implies v \in Values$
 $VCastFun[intro!]: v \in SimpleValues \implies (Cast ? v) \in Values$
inductive-cases *app-val[elim!]*:
 $App e e' \in Values$
inductive-cases *cast-val[elim!]*:
 $Cast \tau e \in Values$

inductive-cases *welltyped-dyn-inv*: $\Gamma \vdash e : ?$

lemma *canonical-form-int*:
 $\llbracket empty \vdash v : BaseT IntT; v \in Values \rrbracket \implies (\exists n. v = Const (IntC n))$
apply (*erule welltyped-int*)
apply *auto*
apply (*case-tac c*) **apply** *auto*
done

lemma *canonical-form-bool*:
 $\llbracket empty \vdash v : BaseT BoolT; v \in Values \rrbracket \implies (\exists b. v = Const (BoolC b))$
apply (*erule welltyped-bool*)
apply *auto*
apply (*case-tac c*) **apply** *auto*
done

lemma *canonical-form-fun*:
 $\llbracket empty \vdash v : (\tau \rightarrow \tau'); v \in Values \rrbracket \implies v \in SimpleFunVal$
apply (*erule welltyped-fun*)
apply *auto*
apply (*case-tac c*)
apply *auto*
done

lemma *canonical-form-dyn*:
 $\llbracket empty \vdash v : ?; v \in Values \rrbracket \implies \exists v'. v = Cast ? v' \wedge v' \in SimpleValues$
apply (*cases rule: Values.cases*)
apply *auto*
apply (*case-tac v*) **apply** *auto*
apply (*erule welltyped-dyn-inv*) **apply** *auto*
apply (*erule welltyped-dyn-inv*) **apply** *auto* **apply** (*case-tac c*) **apply** *auto*
apply (*erule welltyped-lam-inv*) **apply** *auto*
done

— Some example values and non-values

lemma $(\lambda x:\tau. e) \in \text{Values}$ **by auto**

lemma $\text{Cast } ? (\lambda x:\tau. e) \in \text{Values}$ **apply** (rule *VCastFun*) **by auto**

lemma $\text{Cast } (? \rightarrow ?) (\lambda x:\tau. e) \notin \text{Values}$ **by auto**

lemma $\text{Cast } ? (\text{Const Succ}) \in \text{Values}$ **by auto**

lemma $\text{Const } (\text{IntC } n) \in \text{Values}$ **by auto**

lemma $\text{Const } (\text{BoolC } b) \in \text{Values}$ **by auto**

lemma $\text{Cast } (\text{BaseT IntT}) (\text{Const } (\text{IntC } n)) \notin \text{Values}$ **by auto**

lemma $\text{Cast } (? \rightarrow ?) (\text{Const Succ}) \notin \text{Values}$ **by auto**

lemma $\text{Cast } (\text{BaseT IntT} \rightarrow \text{BaseT IntT}) (\text{Const Succ}) \notin \text{Values}$ **by auto**

datatype *result*

= *Val expr* (- 100)

| *Error error* (- 100)

and *error*

= *TypeE* (*TypeError*)

| *CastE* (*CastError*)

| *KillE* (*KillError*)

lemma *typeof-succ-int*: $\text{BaseT IntT} = \text{typeof Succ} \implies \text{False}$ **by auto**

lemma *typeof-prev-int*: $\text{BaseT IntT} = \text{typeof Prev} \implies \text{False}$ **by auto**

lemma *typeof-zero-int*: $\text{BaseT IntT} = \text{typeof IsZero} \implies \text{False}$ **by auto**

lemma *typeof-int-fun*: $\sigma \rightarrow \tau = \text{typeof } (\text{IntC } n) \implies \text{False}$ **by auto**

lemma *typeof-bool-fun*: $\sigma \rightarrow \tau = \text{typeof } (\text{BoolC } b) \implies \text{False}$ **by auto**

consts *delta* :: *const* \Rightarrow *const* \Rightarrow *result* (δ)

primrec

delta-int: $\text{delta } (\text{IntC } n) c = \text{Error TypeE}$

delta-bool: $\text{delta } (\text{BoolC } b) c = \text{Error TypeE}$

delta Succ $c =$

(*case c of*

IntC i \Rightarrow *Val* (*Const* (*IntC* ($i + 1$)))

| *BoolC b* \Rightarrow *Error TypeE*

| *Succ* \Rightarrow *Error TypeE*

| *Prev* \Rightarrow *Error TypeE*

| *IsZero* \Rightarrow *Error TypeE*)

delta Prev $c =$

(*case c of*

IntC i \Rightarrow *Val* (*Const* (*IntC* ($i - 1$)))

| *BoolC b* \Rightarrow *Error TypeE*

| *Succ* \Rightarrow *Error TypeE*

| *Prev* \Rightarrow *Error TypeE*

| *IsZero* \Rightarrow *Error TypeE*)

delta IsZero $c =$

(*case c of*

IntC i \Rightarrow *Val* (*Const* (*BoolC* ($i = 0$)))

| *BoolC b* \Rightarrow *Error TypeE*

| *Succ* \Rightarrow *Error TypeE*

| *Prev* \Rightarrow *Error TypeE*)

| $IsZero \Rightarrow Error\ TypeE$)

lemma *delta-succ-int*:

$delta\ Succ\ (IntC\ n) = Val\ (Const\ (IntC\ (n + 1)))$ **by** *simp*

lemma *delta-succ-other*:

$\neg(\exists\ n.\ c = (IntC\ n)) \Longrightarrow delta\ Succ\ c = Error\ TypeE$

apply (*case-tac c*) **by** *auto*

lemma *delta-prev-int*:

$delta\ Prev\ (IntC\ n) = Val\ (Const\ (IntC\ (n - 1)))$ **by** *simp*

lemma *delta-prev-other*:

$\neg(\exists\ n.\ c = (IntC\ n)) \Longrightarrow delta\ Prev\ c = Error\ TypeE$

apply (*case-tac c*) **by** *auto*

lemma *delta-zero-int*:

$delta\ IsZero\ (IntC\ n) = Val\ (Const\ (BoolC\ (n = 0)))$ **by** *simp*

lemma *delta-zero-other*:

$\neg(\exists\ n.\ c = (IntC\ n)) \Longrightarrow delta\ IsZero\ c = Error\ TypeE$

apply (*case-tac c*) **by** *auto*

consts *unbox* :: *expr* \Rightarrow *expr*

primrec

$unbox\ (Var\ x) = (Var\ x)$

$unbox\ (Const\ c) = (Const\ c)$

$unbox\ (\lambda\ x:\tau.\ e) = (\lambda\ x:\tau.\ e)$

$unbox\ (App\ e_1\ e_2) = (App\ e_1\ e_2)$

$unbox\ (Cast\ \tau\ e) = e$

consts *eval* :: (*expr* \times *nat* \times *result*) *set*

syntax *eval* :: *expr* \Rightarrow *nat* \Rightarrow *result* \Rightarrow *bool* ($- \hookrightarrow_n - [50,50,50] 51$)

translations $e \hookrightarrow_n r == (e, n, r) \in eval$

inductive *eval* **intros**

ELam: $0 < n \Longrightarrow (\lambda\ x:\tau.\ e) \hookrightarrow_n Val\ (\lambda\ x:\tau.\ e)$

EApp: $\llbracket e_1 \hookrightarrow_n Val\ (\lambda\ x:\tau.\ e_3); e_2 \hookrightarrow_n Val\ v_2; [x:=v_2]e_3 \hookrightarrow_n Val\ v_3 \rrbracket$
 $\Longrightarrow App\ e_1\ e_2 \hookrightarrow_{Suc\ n}\ Val\ v_3$

ECastG: $\llbracket e \hookrightarrow_n Val\ v; empty \vdash unbox\ v : (BaseT\ b) \rrbracket$
 $\Longrightarrow (Cast\ (BaseT\ b)\ e) \hookrightarrow_{(Suc\ n)} Val\ (unbox\ v)$

ECastU: $\llbracket e \hookrightarrow_n Val\ v \rrbracket$
 $\Longrightarrow (Cast\ ?\ e) \hookrightarrow_{(Suc\ n)} Val\ (Cast\ ?\ (unbox\ v))$

ECastF: $\llbracket e \hookrightarrow_n Val\ v; empty \vdash unbox\ v : \tau \rightarrow \tau'; (\tau \rightarrow \tau') \sim (\sigma \rightarrow \sigma');$
 $z = maxv\ v + 1 \rrbracket$
 $\Longrightarrow (Cast\ (\sigma \rightarrow \sigma')\ e) \hookrightarrow_{(Suc\ n)} Val\ (\lambda\ z:\sigma.\ (Cast\ \sigma'\ (App\ (unbox\ v)\ (Cast\ \tau\ (Var\ z))))))$

EConst: $0 < n \Longrightarrow Const\ c \hookrightarrow_n Val\ (Const\ c)$

EDelta: $\llbracket e_1 \hookrightarrow_n Val\ (Const\ c_1);$

$$e_2 \hookrightarrow_n \text{Val } (\text{Const } c_2) \text{]} \\ \implies \text{App } e_1 \ e_2 \hookrightarrow_{\text{Suc } n} \delta \ c_1 \ c_2$$

$$\text{EAppP1: } e_1 \hookrightarrow_n \text{Error } \varepsilon \implies \text{App } e_1 \ e_2 \hookrightarrow_{\text{Suc } n} \text{Error } \varepsilon \\ \text{EAppP2: } \llbracket e_1 \hookrightarrow_n \text{Val } v_1; v_1 \in \text{FunVal}; e_2 \hookrightarrow_n \text{Error } \varepsilon \rrbracket \implies \\ \text{App } e_1 \ e_2 \hookrightarrow_{\text{Suc } n} \text{Error } \varepsilon$$

$$\text{EAppT: } \llbracket e_1 \hookrightarrow_n \text{Val } v_1; v_1 \notin \text{FunVal} \rrbracket \\ \implies \text{App } e_1 \ e_2 \hookrightarrow_{\text{Suc } n} \text{Error TypeE} \\ \text{EAppP3: } \llbracket e_1 \hookrightarrow_n \text{Val } (\lambda x:\tau. e_3); e_2 \hookrightarrow_n \text{Val } v_2; [x:=v_2]e_3 \hookrightarrow_n \text{Error } \varepsilon \rrbracket \\ \implies \text{App } e_1 \ e_2 \hookrightarrow_{\text{Suc } n} \text{Error } \varepsilon$$

$$\text{ECastE: } \llbracket e \hookrightarrow_n \text{Val } v; \text{empty} \vdash \text{unbox } v : \sigma; \neg(\sigma \sim \tau) \rrbracket \implies \\ \text{Cast } \tau \ e \hookrightarrow_{\text{Suc } n} \text{Error CastE} \\ \text{ECastP: } e \hookrightarrow_n \text{Error } \varepsilon \implies \text{Cast } \tau \ e \hookrightarrow_{\text{Suc } n} \text{Error } \varepsilon$$

$$\text{EKill: } e \hookrightarrow_0 \text{Error KillE} \\ \text{EVarT: } 0 < n \implies \text{Var } x \hookrightarrow_n \text{Error TypeE}$$

lemma *unbox-value-is-simple*: $v \in \text{Values} \implies \text{unbox } v \in \text{SimpleValues}$

apply (*cases rule: Values.cases*)
apply *auto*
apply (*cases rule: SimpleValues.cases*)
apply *auto*
done

lemma *eval-value[rule-format]*: $e \hookrightarrow_n r \implies \forall e'. r = \text{Val } e' \longrightarrow e' \in \text{Values}$

apply (*induct rule: eval.induct*)
apply *force+*
apply *clarify* **apply** (*erule-tac x=v in allE*) **apply** *simp* **apply** (*rule unbox-value-is-simple*)
apply *simp*
apply *clarify* **apply** (*erule-tac x=v in allE*) **apply** *simp* **apply** (*rule unbox-value-is-simple*)
apply *simp*
apply *force+*
apply (*case-tac c₁*) **apply** *force+*
apply (*case-tac c₂*) **apply** *force+*
apply (*case-tac c₂*) **apply** *force+*
apply (*case-tac c₂*) **apply** *force+*
done

lemma *bvalue-eval[rule-format]*: $\llbracket v \in \text{SimpleValues}; \text{empty} \vdash v : \tau \rrbracket \implies (\exists n. v \hookrightarrow_n \text{Val } v)$

apply (*cases rule: SimpleValues.cases*) **apply** *simp*
apply *simp* **apply** (*erule welltyped-var-inv*) **apply** *simp*
apply *simp* **apply** (*rule-tac x=1 in exI*) **apply** (*rule EConst*) **apply** *arith*
apply *simp* **apply** (*rule-tac x=1 in exI*) **apply** (*rule ELam*) **apply** *arith*
done

lemma *value-eval*[*rule-format*]: $e \in \text{Values} \implies (\forall \tau. \text{empty} \vdash e : \tau \longrightarrow (\exists n. e \hookrightarrow_n \text{Val } e))$

apply (*cases rule: Values.cases*)
apply *simp*
apply *clarify* **apply** (*rule bvalue-eval*) **apply** *simp* **apply** *simp*
apply *clarify*
apply (*erule welltyped-cast-inv*)
apply *simp*
proof (*case-tac* $\sigma = ?$)
fix $v \tau \sigma$
assume $CV: \langle ? \rangle v \in \text{Values}$ **and** $VSV: v \in \text{SimpleValues}$ **and** $VS: \text{empty} \vdash v : \sigma$
and $\sigma \sim ?$
and $T: ? = \tau$ **and** $S: \sigma = ?$
from $VSV VS S$ **have** *False*
apply *simp* **apply** (*erule welltyped-dyn-inv*)
apply *auto* **apply** (*case-tac c*) **apply** *auto* **done**
thus $\exists n. \langle ? \rangle v \hookrightarrow_n \text{Val } (\langle ? \rangle v)$ **by** *simp*
next
fix $v \tau \sigma$
assume $\langle ? \rangle v \in \text{Values}$ **and** $VSV: v \in \text{SimpleValues}$ **and** $VS: \text{empty} \vdash v : \sigma$ **and**
 $\sigma \sim ?$ **and** $? = \tau$ **and** $S: \sigma \neq ?$
from $VSV VS$ **obtain** n **where** $VV: v \hookrightarrow_n \text{Val } v$ **using** *bvalue-eval* **by** *blast*
from $VV VS S$ **have** $\langle ? \rangle v \hookrightarrow_{\text{Suc } n} \text{Val } (\langle ? \rangle (\text{unbox } v))$ **using** *ECastU* **by** *blast*
with VSV **have** $\langle ? \rangle v \hookrightarrow_{\text{Suc } n} \text{Val } (\langle ? \rangle v)$ **apply** (*cases v*) **by** *auto*
thus $\exists n. \langle ? \rangle v \hookrightarrow_n \text{Val } (\langle ? \rangle v)$ **by** *blast*
qed

lemma *delta-sound*:

assumes $V2V: v2 \in \text{Values}$
and $E1S: e1 \hookrightarrow_n \text{Val } (\text{Const } c)$
and $V1S: \text{empty} \vdash (\text{Const } c) : (\tau' \rightarrow \tau)$
and $E2V2: e2 \hookrightarrow_n \text{Val } v2$
and $V2S: \text{empty} \vdash v2 : \tau'$
shows $\exists r. \text{App } e1 e2 \hookrightarrow_{(\text{Suc } n)} r$
 $\wedge ((\exists v \sigma. r = \text{Val } v \wedge v \in \text{Values} \wedge \text{empty} \vdash v : \tau)$
 $\vee (r = \text{Error CastE}) \vee (r = \text{Error KillE}))$

proof –
from $V1S$ **have** $S1: \tau' = \text{BaseT IntT}$
apply (*rule welltyped-const-inv*)
apply (*case-tac c*) **apply** *auto* **done**
from $V2S S1$ **have** $V2I: \text{empty} \vdash v2 : \text{BaseT IntT}$ **by** *simp*
from $V2V V2I$ **have** $X: (\exists i. v2 = \text{Const } (\text{IntC } i))$
using *canonical-form-int* **by** *auto*
from X **obtain** i **where** $V2: v2 = \text{Const } (\text{IntC } i)$ **by** *blast*
from $V2 E2V2$ **have**
 $E2I: e2 \hookrightarrow_n \text{Val } (\text{Const } (\text{IntC } i))$ **by** *simp*
from $E1S E2I$ **have**
 $EI: \text{App } e1 e2 \hookrightarrow_{(\text{Suc } n)} \delta c (\text{IntC } i)$

```

using EDelta by auto
from V1S EI show ?thesis
apply auto
apply (erule welltyped-const-inv)
apply (case-tac c)
apply simp+
apply (rule-tac x=δ c (IntC i) in exI)
using WTCConst apply force
apply (rule-tac x=δ c (IntC i) in exI)
using WTCConst apply force
apply (rule-tac x=δ c (IntC i) in exI)
using WTCConst apply force
done
qed

lemma eval-sound[rule-format]:
  ∀ e τ. empty ⊢ e : τ
  → (∃ r. e ↦n r
  ∧ ((∃ v σ. r = Val v ∧ v ∈ Values ∧ empty ⊢ v : τ)
  ∨ (r = Error CastE) ∨ (r = Error KillE))) (is ?P n)
proof (induct rule: nat-less-induct)
  fix n assume IH: ∀ m < n. ?P m
  show ?P n
  proof clarify
    fix e τ assume ET: empty ⊢ e : τ
    show (∃ r. e ↦n r
    ∧ ((∃ v σ. r = Val v ∧ v ∈ Values ∧ empty ⊢ v : τ)
    ∨ (r = Error CastE) ∨ (r = Error KillE)))
    proof (cases n = 0)
      assume n = 0
      thus ?thesis using EKill by auto
    next
      assume n ≠ 0 hence ngz: n > 0 by simp
      from ET show ?thesis
      proof (cases rule: welltyped.cases)
        fix Γ τ' x assume A: (empty, e, τ) = (Γ, Var x, τ')
          and X: Γ x = [τ']
          from A X have False by auto
          thus ?thesis by simp
        next
          fix Γ τ' c assume A: (empty, e, τ) = (Γ, Const c, τ')
            and typeof- c τ'
            with ngz show ?thesis using EConst by auto
        next
          fix Γ σ τ' e' x
          assume A: (empty, e, τ) = (Γ, λ x:σ. e', σ → τ')
            and ET: welltyped (Γ(x ↦ σ)) e' τ'
            from A ET have LT: Γ ⊢ (λ x:σ. e') : (σ → τ') by auto
            from A ET LT ngz show ?thesis using ELam by blast
        next

```

— Application

fix $\Gamma \tau'' \tau' e1 e2$

assume $A: (empty, e, \tau) = (\Gamma, App\ e1\ e2, \tau')$

and $E1: welltyped\ \Gamma\ e1\ (\tau'' \rightarrow \tau')$ **and** $E2: welltyped\ \Gamma\ e2\ \tau''$

from $E1\ A$ **have** $E1a: empty \vdash e1 : (\tau'' \rightarrow \tau')$ **by** *simp*

from *ngz* **have** $NN1: n - 1 < n$ **by** *arith*

from $IH\ NN1\ E1a$ **obtain** $r1$

where $E1R1: e1 \hookrightarrow_{(n-1)} r1$

and $R1: (\exists v\ \sigma. r1 = Val\ v \wedge v \in Values \wedge empty \vdash v : (\tau'' \rightarrow \tau'))$
 $\vee (r1 = Error\ CastE) \vee (r1 = Error\ KillE)$ **by** *blast*

from $E2\ A$ **have** $E2a: empty \vdash e2 : \tau''$ **by** *simp*

from $IH\ NN1\ E2a$ **obtain** $r2$

where $E2R2: e2 \hookrightarrow_{(n-1)} r2$

and $R2: (\exists v. r2 = Val\ v \wedge v \in Values \wedge empty \vdash v : \tau')$
 $\vee (r2 = Error\ CastE) \vee (r2 = Error\ KillE)$ **by** *blast*

from $R1\ R2$ **have**

$(\exists \varepsilon. r1 = Error\ \varepsilon \wedge \varepsilon \in \{CastE, KillE\})$
 $\vee ((\exists v. r1 = Val\ v \wedge v \in Values \wedge empty \vdash v : (\tau'' \rightarrow \tau'))$
 $\wedge ((\exists \varepsilon. r2 = Error\ \varepsilon \wedge \varepsilon \in \{CastE, KillE\})$
 $\vee (\exists v. r2 = Val\ v \wedge v \in Values \wedge empty \vdash v : \tau')))$

by *blast*

moreover { **assume** $X: (\exists \varepsilon. r1 = Error\ \varepsilon \wedge \varepsilon \in \{CastE, KillE\})$

from X **obtain** ε **where** $R1: r1 = Error\ \varepsilon$

and $EP: \varepsilon \in \{CastE, KillE\}$ **by** *blast*

from $A\ E1R1\ R1$ *ngz* **have** $e \hookrightarrow_n r1$

using $EAppP1[of\ e1\ n - 1\ \varepsilon\ e2]$ **by** *auto*

with $R1\ EP$ **have** *?thesis* **by** *auto*

} **moreover** { **assume** $X:$

$((\exists v. r1 = Val\ v \wedge v \in Values \wedge empty \vdash v : (\tau'' \rightarrow \tau'))$
 $\wedge ((\exists \varepsilon. r2 = Error\ \varepsilon \wedge \varepsilon \in \{CastE, KillE\})$
 $\vee (\exists v. r2 = Val\ v \wedge v \in Values \wedge empty \vdash v : \tau')))$

from X **obtain** $v1$ **where**

$R1: r1 = Val\ v1$ **and** $V1V: v1 \in Values$

and $V1S: empty \vdash v1 : (\tau'' \rightarrow \tau')$ **by** *blast*

from $E1R1\ R1$ **have** $E1V1: e1 \hookrightarrow_{(n-1)} Val\ v1$ **by** *simp*

from $V1V\ V1S$ *canonical-form-fun*[$of\ v1\ \tau''\ \tau'$]

have $V1F: v1 \in SimpleFunVal$ **by** *simp*

note X

moreover { **assume** $XE: (\exists \varepsilon. r2 = Error\ \varepsilon \wedge \varepsilon \in \{CastE, KillE\})$

from XE **obtain** ε **where** $R2: r2 = Error\ \varepsilon$

and $EP: \varepsilon \in \{CastE, KillE\}$ **by** *blast*

from $E2R2\ R2$ **have** $E2E: e2 \hookrightarrow_{(n-1)} Error\ \varepsilon$ **by** *simp*

from $E1V1\ V1F\ E2E$ *ngz* A **have** $e \hookrightarrow_n Error\ \varepsilon$

using $EAppP2[of\ e1\ n - 1\ v1\ e2]$ **by** *auto*

with EP **have** *?thesis* **by** *blast*

} **moreover** {

assume $XV: (\exists v. r2 = Val\ v \wedge v \in Values \wedge empty \vdash v : \tau')$

from XV **obtain** $v2$ **where**

$R2: r2 = Val\ v2$ **and** $V2V: v2 \in Values$

and $V2S: \text{empty} \vdash v2 : \tau''$ **by** *blast*
from $V1F$ **have** $(\exists x \tau e. v1 = (\lambda x:\tau. e))$
 $\vee (\exists c. v1 = \text{Const } c \wedge c \in \text{ConstFun})$
apply (*cases rule: SimpleFunVal.cases*)
by *auto*
moreover { **assume** $X: (\exists x \tau e. v1 = (\lambda x:\tau. e))$
from X **obtain** $x \ t1 \ e'$ **where** $v1 = (\lambda x:t1. e')$ **by** *blast*
with $V1S$ **have** $V1: v1 = (\lambda x:\tau''. e')$
using *welltyped-lam-inv* **by** *blast*
from $E1R1 \ R1 \ V1$ **have** $E1L: e1 \hookrightarrow_{(n-1)} \text{Val } (\lambda x:\tau''. e')$ **by** *simp*
from $E2R2 \ R2$ **have** $E2V2: e2 \hookrightarrow_{(n-1)} \text{Val } v2$ **by** *simp*
from $V1S \ V1$ **have** $LSS: \text{empty} \vdash (\lambda x:\tau''. e') : (\tau'' \rightarrow \tau')$
by *simp*
from LSS **have** $EP: \text{empty}(x \mapsto \tau'') \vdash e' : \tau'$
apply (*cases rule: welltyped.cases*) **by** *auto*
from $EP \ V2S$ **have** $EPT: \text{empty} \vdash [x:=v2]e' : \tau'$
by (*rule substitution*)
from $IH \ NN1 \ EPT$ **obtain** $r3$
where $E3R3: [x:=v2]e' \hookrightarrow_{(n-1)} r3$
and $R3: (\exists v. r3 = \text{Val } v \wedge v \in \text{Values} \wedge \text{empty} \vdash v : \tau')$
 $\vee (r3 = \text{Error CastE} \vee r3 = \text{Error KillE})$ **by** *blast*
note $R3$
moreover { **assume**
 $X: (\exists v. r3 = \text{Val } v \wedge v \in \text{Values} \wedge \text{empty} \vdash v : \tau')$
from X **obtain** $v3 \ s3$ **where** $R3: r3 = \text{Val } v3$
and $V3V: v3 \in \text{Values}$ **and** $V3S: \text{empty} \vdash v3 : \tau'$ **by** *blast*
from $E3R3 \ R3$ **have** $E3V3: [x:=v2]e' \hookrightarrow_{(n-1)} \text{Val } v3$ **by** *simp*
from $E1L \ E2V2 \ E3V3 \ A \ ngz$ **have** $EV3: e \hookrightarrow_n \text{Val } v3$
using $EApp[\text{of } e1 \ n - 1 \ x \ \tau'' \ e' \ e2 \ v2 \ v3]$ **by** *auto*
from $EV3 \ V3V \ V3S \ A$ **have** *?thesis* **by** *blast*
} **moreover** { **assume** $R3: r3 = \text{Error CastE} \vee r3 = \text{Error KillE}$
from $R3$ **obtain** ε **where** $R3: r3 = \text{Error } \varepsilon$
and $EPS: \varepsilon \in \{\text{CastE}, \text{KillE}\}$ **by** *auto*
from $E3R3 \ R3$ **have** $E3E: [x:=v2]e' \hookrightarrow_{(n-1)} \text{Error } \varepsilon$ **by** *simp*
from $E1L \ E2V2 \ E3E \ A \ ngz$ **have** $EE: e \hookrightarrow_n \text{Error } \varepsilon$
using $EAppP3[\text{of } e1 \ n - 1 \ x \ \tau'' \ e' \ e2 \ v2]$ **by** *auto*
with EPS **have** *?thesis* **by** *blast*
} **ultimately** **have** *?thesis* **by** *blast*
} **moreover** { **assume**
 $X: (\exists c. v1 = \text{Const } c \wedge c \in \text{ConstFun})$
from X **obtain** c **where** $V1: v1 = \text{Const } c$
and $C: c \in \text{ConstFun}$ **by** *blast*
from $E1R1 \ R1 \ V1$ **have** $E1C: e1 \hookrightarrow_{(n-1)} \text{Val } (\text{Const } c)$ **by** *simp*
from $E2R2 \ R2$ **have** $E2V2: e2 \hookrightarrow_{(n-1)} \text{Val } v2$ **by** *simp*
from $V1S \ V1$ **have** $V1T: \text{empty} \vdash \text{Const } c : (\tau'' \rightarrow \tau')$ **by** *simp*
from $V2V \ E1C \ E2V2 \ V1T \ V2S \ ngz \ A$
have *?thesis*
using $\text{delta-sound}[\text{of } v2 \ e1 \ n - 1 \ c \ \tau'' \ \tau' \ e2]$ **by** *auto*
} **ultimately** **have** *?thesis* **by** *auto*

```

    } ultimately have ?thesis by blast
  } ultimately show ?thesis by blast
next — Cast
fix  $\Gamma \sigma \tau' e'$ 
assume  $A: (empty, e, \tau) = (\Gamma, \langle \tau \rangle e', \tau')$ 
  and  $ES: welltyped \Gamma e' \sigma$ 
from  $A ES$  have  $ESb: empty \vdash e' : \sigma$  by simp
from  $ngz$  have  $NN1: n - 1 < n$  by arith
from  $IH NN1 ESb$  obtain  $r$ 
  where  $ER: e' \hookrightarrow_{(n-1)} r$ 
  and  $R: (\exists v \sigma. r = Val v \wedge v \in Values \wedge empty \vdash v : \sigma) \vee (r = Error CastE \vee r = Error KillE)$  by blast
note  $R$ 
moreover {
  assume  $X: (\exists v \sigma. r = Val v \wedge v \in Values \wedge empty \vdash v : \sigma)$ 
  from  $X$  obtain  $v \sigma'$  where  $R: r = Val v$ 
  and  $VV: v \in Values$  and  $VS: empty \vdash v : \sigma'$  by blast
  from  $ER R$  have  $EV: e' \hookrightarrow_{(n-1)} Val v$  by simp
  from  $VV VS$  obtain  $s2$  where  $UVS: empty \vdash unbox v : s2$ 
  apply (cases v) apply (cases v) apply auto
  apply (erule welltyped-cast-inv) apply auto done
  have ?thesis
  proof (cases  $\tau$ )
    fix  $t1 t2$  assume  $T: \tau = t1 \rightarrow t2$ 
    show ?thesis
    proof (cases  $s2 \sim \tau$ )
      assume  $s2t: s2 \sim \tau$ 
      from  $s2t T$  show ?thesis
      apply simp apply (erule compat-fun-r) apply simp
    proof —
      assume  $sp: (t1 \rightarrow t2) = s2$ 
      from  $UVS sp$  have  $UVS: empty \vdash unbox v : (t1 \rightarrow t2)$  by simp
      have  $SSTT: (t1 \rightarrow t2) \sim (t1 \rightarrow t2)$  by auto
      let  $?Z = maxv v + 1$ 
      let  $?L = (\lambda ?Z:t1. (Cast t2 (App (unbox v) (Cast t1 (Var ?Z)))))$ 
      from  $EV UVS SSTT$  have
         $CEL: (Cast (t1 \rightarrow t2) e') \hookrightarrow_{(Suc (n-1))} Val ?L$  apply (rule ECastF)
    by simp
    have  $CS: empty(?Z \mapsto t1) \vdash Cast t1 (Var ?Z) : t1$  by auto

    have  $maxv v < ?Z$  by arith
    hence  $?Z \notin Vars v$  by (rule maxv-vars)
    hence  $?Z \notin Vars (unbox v)$  apply (cases v) apply auto done
    with  $UVS$  have  $UVS2: empty(?Z \mapsto t1) \vdash unbox v : (t1 \rightarrow t2)$ 
      by (rule expand-env)
    from  $UVS2 CS$  have  $empty(?Z \mapsto t1) \vdash App (unbox v) (Cast t1 (Var$ 
?Z)) : t2
      by (rule WApp)
    hence  $empty(?Z \mapsto t1) \vdash Cast t2 (App (unbox v) (Cast t1 (Var ?Z))) :$ 

```

$t2$
apply (rule *WTCast*) **by** (rule *compatible-reflexive*)
hence $LT: \text{empty} \vdash ?L : t1 \rightarrow t2$ **by** (rule *WTLam*)
from $CEL\ LT\ A\ T\ ngz\ sp$ **show** $\exists r. e \hookrightarrow_n r \wedge$
 $((\exists v. r = \text{Val } v \wedge v \in \text{Values} \wedge \text{empty} \vdash v : s2)$
 $\vee r = \text{Error CastE} \vee r = \text{Error KillE})$
by *auto*
next
fix $\sigma_1\ \sigma_2$
assume $s1t1: \sigma_1 \sim t1$ **and** $s2t2: \sigma_2 \sim t2$ **and** $sp: s2 = \sigma_1 \rightarrow \sigma_2$
from $UVS\ sp$ **have** $UVS: \text{empty} \vdash \text{unbox } v : (\sigma_1 \rightarrow \sigma_2)$ **by** *simp*
from $sp\ T\ s2t$ **have** $SSTT: (\sigma_1 \rightarrow \sigma_2) \sim (t1 \rightarrow t2)$ **by** *simp*
let $?Z = \text{maxv } v + 1$
let $?L = (\lambda\ ?Z:t1. (\text{Cast } t2\ (\text{App } (\text{unbox } v)\ (\text{Cast } \sigma_1\ (\text{Var } ?Z)))))$
from $EV\ UVS\ SSTT$ **have**
 $CEL: (\text{Cast } (t1 \rightarrow t2)\ e^\wedge) \hookrightarrow_{(\text{Suc } (n - 1))} \text{Val } ?L$ **apply** (rule *ECastF*)
by *simp*
from $s1t1$ **have** $CS: \text{empty}(?Z \mapsto t1) \vdash \text{Cast } \sigma_1\ (\text{Var } ?Z) : \sigma_1$
apply *auto* **using** *compatible-symmetric* **by** *simp*

have $\text{maxv } v < ?Z$ **by** *arith*
hence $?Z \notin \text{Vars } v$ **by** (rule *maxv-vars*)
hence $?Z \notin \text{Vars } (\text{unbox } v)$ **apply** (cases v) **apply** *auto* **done**
with UVS **have** $UVS2: \text{empty}(?Z \mapsto t1) \vdash \text{unbox } v : (\sigma_1 \rightarrow \sigma_2)$
by (rule *expand-env*)
from $UVS2\ CS$ **have** $\text{empty}(?Z \mapsto t1) \vdash \text{App } (\text{unbox } v)\ (\text{Cast } \sigma_1\ (\text{Var } ?Z)) : \sigma_2$

by (rule *WTApp*)
hence $\text{empty}(?Z \mapsto t1) \vdash \text{Cast } t2\ (\text{App } (\text{unbox } v)\ (\text{Cast } \sigma_1\ (\text{Var } ?Z))) :$
 $t2$

by (rule *WTCast*)
hence $LT: \text{empty} \vdash ?L : t1 \rightarrow t2$ **by** (rule *WTLam*)
from $CEL\ LT\ A\ T\ ngz\ show\ \exists r. e \hookrightarrow_n r \wedge$
 $((\exists v. r = \text{Val } v \wedge v \in \text{Values} \wedge \text{empty} \vdash v : t1 \rightarrow t2)$
 $\vee r = \text{Error CastE} \vee r = \text{Error KillE})$
by *auto*
next
assume $S2: s2 = ?$
with $VV\ UVS$ **have** *False* **apply** *simp*
apply (rule *welltyped-dyn-inv*)
apply (cases v) **apply** *auto*
apply (case-tac c) **apply** *auto*
apply (cases v) **apply** *auto*
apply (cases v) **apply** *auto*
done
thus $\exists r. e \hookrightarrow_n r \wedge$
 $((\exists v. r = \text{Val } v \wedge v \in \text{Values} \wedge \text{empty} \vdash v : t1 \rightarrow t2)$
 $\vee r = \text{Error CastE} \vee r = \text{Error KillE})$ **by** *simp*
qed

```

    next
      assume notst:  $\neg(s2 \sim \tau)$ 
      from EV UVS notst have Cast  $\tau e' \hookrightarrow_{Suc (n - 1)}$  Error CastE by (rule
ECastE)
        with A ngz show ?thesis by auto
      qed
    next
      fix b assume T:  $\tau = BaseT b$ 
      show ?thesis
      proof (cases s2  $\sim \tau$ )
        assume s2t: s2  $\sim \tau$ 
        with UVS T VV have S2: s2 = BaseT b
          apply simp apply (erule compat-base-r) apply auto
          apply (erule welltyped-dyn-inv) apply auto
          apply (case-tac c) apply auto
          apply (cases v) apply auto
          apply (cases v) apply auto
        done
        from UVS S2 have UVB: empty  $\vdash$  unbox v : BaseT b by simp
        from EV T UVB have CEV: (Cast  $\tau e' \hookrightarrow_{(Suc (n - 1))}$  Val (unbox v))
          apply simp apply (rule ECastG) apply auto done
        from VV have UVV: unbox v  $\in$  Values
          apply (cases v) apply auto apply (cases v) apply auto done
        from CEV A T UVV UVS ngz S2 show ?thesis by auto
      next
        assume s2t:  $\neg (s2 \sim \tau)$ 
        from EV UVS s2t have CEE: (Cast  $\tau e' \hookrightarrow_{(Suc (n - 1))}$  Error CastE)
by (rule ECastE)
          with A ngz show ?thesis by auto
        qed
      next
        assume T:  $\tau = ?$ 
        from EV have CEC: (Cast ? e'  $\hookrightarrow_{(Suc (n - 1))}$  Val (Cast ? (unbox v)))
by (rule ECastU)
          from UVS have CVT: empty  $\vdash$  Cast ? (unbox v) : ? apply (rule WTCast)
apply auto done
          from VV have CVV: Cast ? (unbox v)  $\in$  Values
            apply (cases v) apply auto apply (cases v) apply auto done
          from T A CEC CVT CVV ngz show ?thesis by auto
        qed
      } moreover { assume X: (r = Error CastE  $\vee$  r = Error KillE)
        from ER X ngz
          have Cast  $\tau e' \hookrightarrow_n$  r using ECastP[of e' n - 1] by auto
          with A X have ?thesis by auto
        } ultimately show ?thesis by blast
    qed
  qed
  qed
  qed

```

theorem *type-safety*:

assumes *EET*: $\text{empty} \vdash e \Rightarrow e' : \tau$

shows $(\exists r. e' \hookrightarrow_n r$

$\wedge ((\exists v \sigma. r = \text{Val } v \wedge v \in \text{Values} \wedge \text{empty} \vdash v : \tau)$

$\vee (r = \text{Error CastE}) \vee (r = \text{Error KillE}))$)

proof –

from *EET* **have** $\text{empty} \vdash e' : \tau$ **by** (*rule compilation-sound*)

thus *?thesis* **by** (*rule eval-sound*)

qed

end