Gradual Typing for Objects: Isabelle Formalization

Jeremy Siek
Department of Computer Science
University of Colorado
430 UCB
Boulder, CO 80309-0430 USA

Walid Taha
Department of Computer Science
Rice University
Mail Stop 132
6100 S. Main Street
Houston, TX 77005-1892

CU Technical Report CU-CS-1021-06
December 2006

Contents

1 Introduction

This technical report is a formalization of a new calculus named $\mathbf{FOb}^{?}_{<:}$ that extends the $\mathbf{FOb}_{<:}$ calculus of Abadi and Cardelli [1] with support for gradual typing. We introduced the notion of gradual typing in the context of functional languages, developing the $\lambda^{?}_{\rightarrow}$ calculus [6]. The work in this technical report, and in the companion paper, *Gradual Typing for Objects*, shows how to integrate gradual typing into object-oriented languages. The companion papers gives the motivation for this work, an introduction to gradual typing, and a detailed discussion of related work. This technical report contains the formalization of the type system and semantics using the Isabelle/HOL proof assistant [4], and is meant to be a reference for readers of the companion paper.

2 Choosing Fresh Variables

In various places within the formal development we need to choose a "fresh" variable. More specifically, we need to choose a variable that is not in some set, such as the domain of the type environment. Variables are represented here as natural numbers, and we constructively choose a fresh variable by taking the successor of the maximum number in the set. Of course, we must assume that the set in question is finite.

constdefs *max* :: *nat* ⇒ *nat* ⇒ *nat*
  max x y ≡ (*if x* < *y then y else x*)
declare *max-def* [*simp*]

To define the maximum number in a set, we take advantage of Isabelle's ability to fold over a finite set. To use fold with the above max function, we must first prove a few properties of max, but the proofs go through automatically.

interpretation *AC-max*: *ACe* [*max 0*::*nat*]
  by (*auto intro*: *ACf.intro ACe-axioms.intro*)

constdefs *setmax* :: *nat set* ⇒ *nat*
  setmax S ≡ *fold max* (λ *x*. *x*) *0 S*

We want to show that the successor of the maximum element of a set is not in the set. Towards proving that we prove the following lemma.

lemma *max-ge*: *finite L* ⟹ ∀ *x* ∈ *L*. *x* ≤ *setmax L*
  apply (*induct rule*: *finite-induct*)
  apply *simp*
  apply *clarify*
  apply (*case-tac xa* = *x*)
proof −
  fix *x* **and** *F*::*nat set* **and** *xa*
  assume *fF*: *finite F* **and** *xF*: *x* ∉ *F* **and** *xax*: *xa* = *x*
  from *fF xF* **have** *mc*: *setmax* (*insert x F*) = *max x* (*setmax F*)

apply (*simp only*: *setmax-def*)
apply (*rule AC-max.fold-insert*)
apply *auto* **done**
with *xax* **show** $xa \leq$ *setmax* (*insert x F*)
apply *clarify* **by** *simp*
next
fix *x* **and** *F*::*nat set* **and** *xa*
assume *fF*: *finite F* **and** *xF*: $x \notin F$
and *axF*: $\forall x \in F.\ x \leq$ *setmax F*
and *xsxF*: $xa \in$ *insert x F*
and *xax*: $xa \neq x$
from *xax xsxF* **have** *xaF*: $xa \in F$ **by** *auto*
with *axF* **have** *xasF*: $xa \leq$ *setmax F* **by** *blast*
from *fF xF* **have** *mc*: *setmax* (*insert x F*) = *max x* (*setmax F*)
apply (*simp only*: *setmax-def*)
apply (*rule AC-max.fold-insert*)
apply *auto* **done**
with *xasF* **show** $xa \leq$ *setmax* (*insert x F*) **by** *auto*
qed

lemma *max-is-fresh*[*simp*]:
assumes *F*: *finite L* **shows** *Suc* (*setmax L*) $\notin L$
proof
assume *ssl*: *Suc* (*setmax L*) $\in L$
with *F max-ge* **have** *Suc* (*setmax L*) $\leq$ *setmax L* **by** *blast*
thus *False* **by** *simp*
qed

lemma *greaterthan-max-is-fresh*[*simp*]:
assumes *F*: *finite L* **and** *I*: *setmax L* $< i$
shows $i \notin L$
proof
assume *ssl*: $i \in L$
with *F max-ge* **have** $i \leq$ *setmax L* **by** *blast*
with *I* **show** *False* **by** *simp*
qed

3 Abstract Syntax

The signatures in an object type are represented in a list, and are assumed to appear in the order of the method name. The methods in and object are also represented by a sorted list.

datatype *ty* = *IntT* | *FloatT* | *BoolT* | *ArrowT ty ty* (**infixr** → *95*)
| *ObjT sig list* | *UnknownT* (*?*)
and
sig = *Sig nat ty*

constdefs *ground* :: *ty set*
ground ≡ { *IntT*, *BoolT*, *FloatT* }

datatype *const* = *IntC int* | *FloatC int* | *BoolC bool* | *Succ* | *IsZero*

The expr datatype is used for both the source language, $\mathbf{FOb}^{?}_{<:}$ and the intermediate language $\mathbf{FOb}^{\langle\cdot\rangle}_{<:}$. We use the *locally nameless* approach for representing variables [2, 3, 5]. In the locally nameless approach, bound variables are represented with de Bruijn indices whereas free variables are represented with symbols. This approach enjoys the benefits of the de Bruijn indices (α-equivalent terms are syntactically identical) while avoiding much of the complication (normally caused by representing free variables with de Bruijn indices). Separate functions are used to substitution for free and bound variables.

datatype *expr* = *BVar nat* | *FVar nat* | *Const const*
| *Lam ty expr* (λ:-. - [53,53] 52) | *App expr expr*
| *Cast expr ty ty* (-⟨-⇒-⟩ [53,53,53] 52)
| *Obj method list ty* | *Invoke expr nat* | *Update expr method*
and
method = *Method nat expr*

syntax *just* :: *'a* ⇒ *'a option*
translations *just* τ == *Some* τ

3.1 Substitution for Free Variables

consts
fsubst :: *nat* ⇒ *expr* ⇒ *expr* ⇒ *expr* ([-→-]- [54,54,54] 53)
fsubstm :: *nat* ⇒ *expr* ⇒ *method* ⇒ *method* ([-:→-]- [54,54,54] 53)
fsubsts :: *nat* ⇒ *expr* ⇒ *method list* ⇒ *method list* ([-::→-]- [54,54,54] 53)
primrec
fbvar: [*z*→*e*](*BVar i*) = *BVar i*
ffvar:[*z*→*e*](*FVar x*) = (*if z* = *x then e else* (*FVar x*))
[*z*→*e*](*Const c*) = *Const c*
flam: [*z*→*e*](λ:σ. *e'*) = (λ:σ. [*z*→*e*]*e'*)
[*z*→*e*](*App e1 e2*) = *App* ([*z*→*e*]*e1*) ([*z*→*e*]*e2*)
[*z*→*e*](*Cast e' s t*) = *Cast* ([*z*→*e*]*e'*) *s t*
[*z*→*e*](*Obj ms* τ) = *Obj* (*fsubsts z e ms*) τ
[*z*→*e*](*Invoke e' l*) = *Invoke* ([*z*→*e*]*e'*) *l*
[*z*→*e*](*Update e' m*) = *Update* ([*z*→*e*]*e'*) ([*z*:→*e*]*m*)

[*z*:→*e*](*Method l e'*) = (*Method l* ([*z*→*e*]*e'*))

[*z*::→*e*][] = []
[*z*::→*e*](*m*#*ms*) = ([*z*:→*e*]*m*)#([*z*::→*e*]*ms*)

3.2 Substitution for Bound Variables

consts
bsubst :: *nat* ⇒ *expr* ⇒ *expr* ⇒ *expr* ({-→-}- [54,54,54] 53)

bsubstm :: *nat* ⇒ *expr* ⇒ *method* ⇒ *method* ({-:→-}- [54,54,54] 53)
bsubsts :: *nat* ⇒ *expr* ⇒ *method list* ⇒ *method list* ({-::→-}- [54,54,54] 53)
primrec
bbvar: {k→e}(*BVar i*) = (*if k* = *i then e else* (*BVar i*))
bfvar: {k→e}(*FVar x*) = *FVar x*
{k→e}(*Const c*) = *Const c*
blam: {k→e}(λ:σ. e′) = (λ:σ. {*Suc k*→e}e′)
{k→e}(*App e1 e2*) = *App* ({k→e}*e1*) ({k→e}*e2*)
{z→e}(*Cast e′ s t*) = *Cast* ({z→e}e′) *s t*
{k→e}(*Obj ms* τ) = *Obj* (*bsubsts k e ms*) τ
{k→e}(*Invoke e′ l*) = *Invoke* ({k→e}e′) *l*
{k→e}(*Update e′ m*) = *Update* ({k→e}e′) ({k:→e}*m*)

{k:→e}(*Method l e′*) = (*Method l* ({k→e}e′))

{k::→e} [] = []
{k::→e} (*m*#*ms*) = ({k:→e}*m*)#({k::→e}*ms*)

4 Consistency and Subtyping

constdefs
mname :: *method* ⇒ *nat* (*name*)
mname m ≡ (*case m of* (*Method l e*) ⇒ *l*)

constdefs
ms-name :: *sig* ⇒ *nat* (*name*)
ms-name m ≡ (*case m of* (*Sig l* τ) ⇒ *l*)
ms-ty :: *sig* ⇒ *ty*
ms-ty s ≡ (*case s of* (*Sig l* τ) ⇒ τ)

consts *lookup-sig* :: *sig list* ⇒ *nat* ⇒ *sig option*
primrec
lookup-sig [] *l* = *None*
lookup-sig (*m*#*ms*) *l* = (*if ms-name m* = *l then Some m*
else lookup-sig ms l)

consts *Dom* :: *method list* ⇒ *nat set*
primrec
Dom [] = {}
Dom (*m*#*ms*) = *insert* (*mname m*) (*Dom ms*)

consts *DomT* :: *sig list* ⇒ *nat set*
primrec
DomT [] = {}
DomT (*m*#*ms*) = *insert* (*ms-name m*) (*DomT ms*)

consts
consistent :: (*ty* × *ty*) *set*
consistent-sig :: (*sig* × *sig*) *set*

consistent-sigs :: (*sig list* × *sig list*) *set*
syntax
consistent :: *ty* ⇒ *ty* ⇒ *bool* (**infix** ∼ *51*)
consistent-sig :: *sig* ⇒ *sig* ⇒ *bool* (**infix** ≅ *51*)
consistent-sigs :: *sig list* ⇒ *sig list* ⇒ *bool* (**infix** ≈ *51*)
translations
$\tau 1 \sim \tau 2$ == $(\tau 1, \tau 2) \in$ *consistent*
$\tau 1 \cong \tau 2$ == $(\tau 1, \tau 2) \in$ *consistent-sig*
$\tau 1 \approx \tau 2$ == $(\tau 1, \tau 2) \in$ *consistent-sigs*
inductive *consistent consistent-sig consistent-sigs* **intros**
CRefl[*intro*!]: $\tau \sim \tau$
CFun[*intro*!]: $[\![\sigma_1 \sim \tau_1; \sigma_2 \sim \tau_2]\!] \Longrightarrow (\sigma_1 \rightarrow \sigma_2) \sim (\tau_1 \rightarrow \tau_2)$
CUnR[*intro*!]: $\tau \sim$ *?*
CUnL[*intro*!]: *?* $\sim \tau$
CObjT[*intro*!]: *ss* ≈ *tt* ⟹ *ObjT ss* ∼ *ObjT tt*

CSig[*intro*!]: $[\![l = l'; \sigma \sim \tau]\!] \Longrightarrow$ *Sig* $l\ \sigma \cong$ *Sig* $l'\ \tau$

CNilT[*intro*!]: [] ≈ []
CConsT[*intro*!]: ⟦ *s* ≅ *t*; *ss* ≈ *tt* ⟧ ⟹ (*s*#*ss*) ≈ (*t*#*tt*)

inductive-cases *con-fun-inv*[*elim*!]: *s1* → *s2* ∼ *t1* → *t2*
inductive-cases *con-sig-inv*[*elim*!]: *s* ≅ *t*
inductive-cases *con-sigs-inv*: *ss* ≈ *tt*

lemma *consistent-reflexive*:
$(\sigma \sim \sigma) \wedge (s \cong s) \wedge (ss \approx ss)$
apply (*induct rule*: *ty-sig.induct*)
apply *auto*
done

lemma *consistent-sigs-reflexive*:
ss ≈ *ss*
using *consistent-reflexive* **by** *simp*

lemma *consistent-symmetric*:
$(\sigma \sim \tau \longrightarrow \tau \sim \sigma) \wedge (s \cong t \longrightarrow t \cong s) \wedge (ss \approx tt \longrightarrow tt \approx ss)$
apply (*induct rule*: *consistent-consistent-sig-consistent-sigs.induct*)
by *auto*

inductive-cases *cons-int-bool*[*elim*!]: *IntT* ∼ *BoolT*

lemma *consistent-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*: *IntT* ∼ *?* **by** *auto*
have *B*: *?* ∼ *BoolT* **by** *auto*
have *C*: ¬ (*IntT* ∼ *BoolT*) **by** *auto*
from *A B C* **show** *?thesis* **by** *auto*
qed

lemma *cons-sig-name*: $s \cong t \Longrightarrow$ *ms-name s* = *ms-name t*
using *ms-name-def* **by** *auto*

4.1 Type Restriction

It is difficult to express the type restriction operator as a function in Isabelle because it requires general recursion and mutual recursion, which is not supported by the recdef facility. We instead define the restrict operator via axioms. Given more time, it would be preferable to define it as a relation and then prove that the relation is a function.

consts
restrict-ty :: *ty* ⇒ *ty* ⇒ *ty* (-|- [*99*,*99*] *98*)
restrict-sig :: *sig* ⇒ *sig* ⇒ *sig* (-↓- [*99*,*99*] *98*)
restrict-sigs :: *sig list* × *sig list* ⇒ *sig list*

syntax
restrict-sigs- :: *sig list* ⇒ *sig list* ⇒ *sig list* (-‖- [*202*,*202*] *201*)
translations
ss‖*tt* == *restrict-sigs*(*ss*, *tt*)

lemma *rbool*[*simp*]: *BoolT*|τ = (*if* τ = *?* *then* *?* *else* *BoolT*) **sorry**
lemma *rint*[*simp*]: *IntT*|τ = (*if* τ = *?* *then* *?* *else* *IntT*) **sorry**
lemma *rfloat*[*simp*]: *FloatT*|τ = (*if* τ = *?* *then* *?* *else* *FloatT*) **sorry**
lemma *rfun*[*simp*]:
$(\sigma_1 \rightarrow \sigma_2)|\tau$ =
(*case* τ *of* *IntT* ⇒ $(\sigma_1 \rightarrow \sigma_2)$ | *FloatT* ⇒ $(\sigma_1 \rightarrow \sigma_2)$ | *BoolT* ⇒ $(\sigma_1 \rightarrow \sigma_2)$
| $\tau_1 \rightarrow \tau_2$ ⇒ $\sigma_1|\tau_1 \rightarrow \sigma_2|\tau_2$
| *ObjT tt* ⇒ $(\sigma_1 \rightarrow \sigma_2)$
| *?* ⇒ *?*) **sorry**
lemma *robj*[*simp*]:
(*ObjT ss*)|τ =
(*case* τ *of* *IntT* ⇒ (*ObjT ss*) | *FloatT* ⇒ (*ObjT ss*) | *BoolT* ⇒ (*ObjT ss*)
| τ*1* → τ*2* ⇒ (*ObjT ss*)
| *ObjT tt* ⇒ *ObjT* (*ss*‖*tt*)
| *?* ⇒ *?*) **sorry**
lemma *runk*[*simp*]:
?|τ = *?* **sorry**

defs *restrict-sig-def*:
s↓*t* ≡ (*case s of* (*Sig l* σ) ⇒ (*case t of* (*Sig l'* τ) ⇒
if l = *l'* *then Sig l* ($\sigma|\tau$) *else Sig l* σ))
declare *restrict-sig-def*[*simp*]

recdef *restrict-sigs measure* (λ *p*. *size* (*fst p*) + *size* (*snd p*))
restrict-sigs(*ss*, *tt*) = (*case ss of* [] ⇒ []
| (*s*#*ss*) ⇒
(*case tt of* [] ⇒ (*s*#*ss*)
| (*t*#*tt*) ⇒

```
        if ms-name s < ms-name t then s#(restrict-sigs(ss, t#tt))
        else (if ms-name s > ms-name t then restrict-sigs(s#ss,tt)
              else (s↓t)#(restrict-sigs(ss,tt)))))

lemma restrict-sig-name: ms-name (s↓t) = ms-name s
  apply (case-tac s) apply (case-tac t) using ms-name-def by auto

lemma restrict-self-id: (τ|τ = (τ::ty)) ∧ (t↓t = t) ∧ (tt‖tt = tt)
  apply (induct rule: ty-sig.induct)
  apply force apply force apply force apply force apply force apply force
  apply force apply force apply force
  done

lemma consistent-restrict-impl:
  (∀ (ϱ::ty) τ. n = size τ + size ϱ ⟶ τ ∼ τ|ϱ )
   ∧ (∀ r t. n = size t + size r ⟶ t ≅ t↓r)
  ∧ (∀ rr tt. n = sig-list-size tt + sig-list-size rr ⟶ tt ≈ tt‖rr) (is ?P n)
proof (induct rule: nat-less-induct)
  fix n
  assume IH: ∀m<n. ?P m
  show ?P n
    apply (rule conjI) apply (rule allI)+ apply (rule impI) defer
    apply (rule conjI) apply (rule allI)+ apply (rule impI) defer
    apply (rule allI)+ apply (rule impI) defer
  proof −
    fix ϱ::ty and τ::ty assume n: n = size τ + size ϱ
    show τ ∼ τ|ϱ
      apply (cases τ)
      apply (cases ϱ) apply force apply force apply force apply force apply force
        apply force
      apply (cases ϱ) apply force apply force apply force apply force apply force
        apply force
      apply (cases ϱ) apply force apply force apply force apply force apply force
        apply force
      apply (cases ϱ) apply force apply force apply force defer apply force apply
force
      apply (cases ϱ) apply force apply force apply force apply force defer apply
force
      apply force defer
    proof −
      fix tt rr assume t: τ = ObjT tt and r: ϱ = ObjT rr
      let ?m = sig-list-size tt + sig-list-size rr
      from t r n have mn: ?m < n by auto
      from mn IH have ttrr: tt ≈ tt‖rr by auto
      with t r show ?thesis by auto
    next
      fix t1 t2 r1 r2 assume t: τ = t1 → t2 and r: ϱ = r1 → r2
      let ?m1 = size t1 + size r1
      from n t r have mn1: ?m1 < n by auto
      from mn1 IH have t1r1: t1 ∼ t1|r1 by simp
```

```
    let ?m2 = size t2 + size r2
    from n t r have mn2: ?m2 < n by auto
    from mn2 IH have t2r2: t2 ∼ t2|r2 by simp

    from t1r1 t2r2 t r show ?thesis by auto
  qed
next
  fix r::sig and t::sig
  assume n: n = size t + size r
  obtain l τ where t: t = Sig l τ apply (cases t) by auto
  obtain l′ ϱ where r: r = Sig l′ ϱ apply (cases r) by auto
  let ?m = size τ + size ϱ
  from t r n have mn: ?m < n by auto
  from mn IH have tr: τ ∼ τ|ϱ by auto
  with t r show t ≅ t↓r by auto
next
  fix rr tt assume n: n = sig-list-size tt + sig-list-size rr
  show tt ≈ tt‖rr
    apply (cases tt)
    apply force
    apply (cases rr)
      apply simp using consistent-reflexive apply blast
  proof −
    fix t ts r rs assume t: tt = t#ts and r: rr = r#rs
    show ?thesis
    proof (cases ms-name t < ms-name r)
      assume tr-name: ms-name t < ms-name r

      from t r tr-name have ttrr: tt‖rr = t#(ts‖(r#rs)) by simp
      let ?m = sig-list-size ts + sig-list-size (r#rs)
      from n t r have mn: ?m < n by auto
      from mn IH have tsrs: ts ≈ ts‖(r#rs) by blast
      have tt: t ≅ t using consistent-reflexive by blast
      from tt tsrs have t#(ts) ≈ t#(ts‖(r#rs)) by (rule CConsT)
      with t ttrr show ?thesis by simp
    next
      assume tr-name: ¬(ms-name t < ms-name r)
      show ?thesis
      proof (cases ms-name t = ms-name r)
        assume ter: ms-name t = ms-name r
        from t r ter have ttrr: tt‖rr = (t↓r)#(ts‖rs) by simp

        let ?m = sig-list-size ts + sig-list-size rs
        from n t r have mn: ?m < n by auto
        from mn IH have tsrs: ts ≈ ts‖rs by blast

        let ?m2 = size t + size r
        from n t r have mn2: ?m2 < n by auto
        from mn2 IH have ttr: t ≅ t↓r by blast
```

```
      from ttr tsrs have t#ts ≈ (t↓r)#(ts‖rs) by (rule CConsT)
      with t r ttrr show ?thesis by simp
    next
      assume tr-name2: ms-name t ≠ ms-name r
      from tr-name tr-name2 have tgr: ms-name t > ms-name r by simp
      from t r tgr have ttrr: tt‖rr = ((t#ts)‖rs) by simp
      let ?m = sig-list-size (t#ts) + sig-list-size rs
      from n t r have mn: ?m < n by auto
      from mn IH have tsrs: (t#ts) ≈ (t#ts)‖rs by blast
      with t ttrr show tt ≈ tt‖rr by simp
    qed
   qed
  qed
 qed
qed

lemma consistent-restrict: τ ∼ τ|(ϱ::ty)
 using consistent-restrict-impl by blast

lemma consistent-implies-intersect-eq:
 (σ ∼ τ ⟶ σ|τ = τ|σ)∧ (s ≅ t ⟶ s↓t = t↓s) ∧ (ss ≈ tt ⟶ ss‖tt = tt‖ss)
 apply (induct rule: consistent-consistent-sig-consistent-sigs.induct)
 apply force
 apply force
 apply simp apply (case-tac τ) apply simp+
 apply (case-tac τ) apply simp apply simp apply simp apply simp apply simp
apply simp
 apply simp
 apply simp
 apply simp
  apply (erule con-sig-inv) apply clarify
  apply (erule con-sigs-inv)
   apply (simp add: ms-name-def)
   apply (simp add: ms-name-def)
 done

lemma intersect-eq-implies-consistent:
 (∀ σ τ. n = size σ + size τ ∧ σ|τ = τ|σ ⟶ σ ∼ τ)
 ∧ (∀ s t. n = size s + size t ∧ s↓t = t↓s ⟶ s ≅ t)
 ∧ (∀ ss tt. n = sig-list-size ss + sig-list-size tt ∧ ss‖tt = tt‖ss ⟶ ss ≈ tt)
 (is ?P n)
proof (induct rule: nat-less-induct)
 fix n
 assume IH: ∀ m<n. ?P m
 show ?P n
   apply (rule conjI) apply (rule allI)+ apply (rule impI) apply (erule conjE)
defer
   apply (rule conjI) apply (rule allI)+ apply (rule impI) apply (erule conjE)
defer
  apply (rule allI)+ apply (rule impI) apply (erule conjE) defer
```

proof −
fix σ::*ty* **and** τ::*ty* **assume** *n*: $n = size\ \sigma + size\ \tau$ **and** *stts*: $\sigma|\tau = \tau|\sigma$
from *stts* **show** $\sigma \sim \tau$
apply (*cases* σ)
apply (*cases* τ) **apply** *force* **apply** *force* **apply** *force* **apply** *force* **apply** *force*
apply *force*
apply (*cases* τ) **apply** *force* **apply** *force* **apply** *force* **apply** *force* **apply** *force*
apply *force*
apply (*cases* τ) **apply** *force* **apply** *force* **apply** *force* **apply** *force* **apply** *force*
apply *force*
apply (*cases* τ) **apply** *force* **apply** *force* **apply** *force* **defer** **apply** *force* **apply** *force*
apply (*cases* τ) **apply** *force* **apply** *force* **apply** *force* **apply** *force* **defer** **apply** *force*
apply *force*
proof −
fix *s1 s2 t1 t2* **assume** *s*: $\sigma = s1 \to s2$ **and** *t*: $\tau = t1 \to t2$
let $?m1 = size\ s1 + size\ t1$
from *n s t* **have** *mn1*: $?m1 < n$ **by** *auto*
from *stts s t* **have** *stts1*: $s1|t1 = t1|s1$ **by** *auto*
from *stts s t* **have** *stts2*: $s2|t2 = t2|s2$ **by** *auto*
from *stts1 mn1 IH* **have** *s1t1*: $s1 \sim t1$ **by** *blast*
let $?m2 = size\ s2 + size\ t2$
from *n s t* **have** *mn2*: $?m2 < n$ **by** *auto*
from *stts2 mn2 IH* **have** *s2t2*: $s2 \sim t2$ **by** *blast*
from *s1t1 s2t2 s t* **show** *?thesis* **by** *auto*
next
fix *ss tt* **assume** *s*: $\sigma = ObjT\ ss$ **and** *t*: $\tau = ObjT\ tt$
let $?m = sig\text{-}list\text{-}size\ ss + sig\text{-}list\text{-}size\ tt$
from *s t n* **have** *mn*: $?m < n$ **by** *auto*
from *stts s t* **have** *ssttss*: $ss\|tt = tt\|ss$ **by** *simp*
from *ssttss mn IH* **have** *ttrr*: $ss \approx tt$ **by** *blast*
with *s t* **show** *?thesis* **by** *auto*
qed
next
fix *s*::*sig* **and** *t*::*sig*
assume *n*: $n = size\ s + size\ t$ **and** *stts*: $s{\downarrow}t = t{\downarrow}s$
obtain $l\ \sigma$ **where** *s*: $s = Sig\ l\ \sigma$ **apply** (*cases s*) **by** *auto*
obtain $l'\ \tau$ **where** *t*: $t = Sig\ l'\ \tau$ **apply** (*cases t*) **by** *auto*
let $?m = size\ \sigma + size\ \tau$
from *s t n* **have** *mn*: $?m < n$ **by** *auto*
from *stts s t* **have** *ll*: $l = l'$
apply (*case-tac* $l = l'$) **apply** *auto* **done**
from *ll stts s t* **have** *stts2*: $\sigma|\tau = \tau|\sigma$ **by** *simp*
from *stts2 mn IH* **have** *st*: $\sigma \sim \tau$ **by** *auto*
with *ll s t* **show** $s \cong t$ **by** *auto*
next
fix *ss tt* **assume** *n*: $n = sig\text{-}list\text{-}size\ ss + sig\text{-}list\text{-}size\ tt$
and *ssttss*: $ss\|tt = tt\|ss$
from *ssttss* **show** $ss \approx tt$

```
  apply (cases ss)
    apply (cases tt) apply force apply force
    apply (cases tt) apply force
proof -
  fix s ls t ts assume s: ss = s#ls and t: tt = t#ts
  show ?thesis
  proof (cases ms-name s < ms-name t)
    assume stn: ms-name s < ms-name t
    have ss ≈ ss‖tt using consistent-restrict-impl by blast
    with s t have s ≅ hd(ss‖tt)
      apply clarify apply (erule con-sigs-inv) by auto
    with cons-sig-name have shn: ms-name s = ms-name(hd(ss‖tt)) by  simp
    have tt ≈ tt‖ss using consistent-restrict-impl by blast
    with s t have t ≅ hd(tt‖ss)
      apply clarify apply (erule con-sigs-inv) by auto
    with ssttss have t ≅ hd(ss‖tt) by simp
    with cons-sig-name have thn: ms-name t = ms-name(hd(ss‖tt)) by simp
    from shn thn stn have False by simp
    thus ?thesis by simp
  next
    assume sget: ¬ (ms-name s < ms-name t)
    show ?thesis
    proof (cases ms-name s = ms-name t)
      assume ste: ms-name s = ms-name t
      from s t ste have sstt: ss‖tt = (s↓t)#(ls‖ts) by simp
      from s t ste have ttss: tt‖ss = (t↓s)#(ts‖ls) by simp
      from ssttss sstt ttss have sts: s↓t = t↓s by simp
      from ssttss sstt ttss have lstsls: ls‖ts = ts‖ls by simp

      let ?m = sig-list-size ls + sig-list-size ts
      from n s t have mn: ?m < n by auto
      from lstsls mn IH have lsts: ls ≈ ts by blast

      let ?m2 = size s + size t
      from n s t have mn2: ?m2 < n by auto
      from sts mn2 IH have st: s ≅ t by blast
      from st lsts have s#ls ≈ t#ts by (rule CConsT)
      with s t show ?thesis by simp
    next
      assume stne: ms-name s ≠ ms-name t
      have ss ≈ ss‖tt using consistent-restrict-impl by blast
      with s t have s ≅ hd(ss‖tt)
        apply clarify apply (erule con-sigs-inv) by auto
      with cons-sig-name have shn: ms-name s = ms-name(hd(ss‖tt)) by  simp
      have tt ≈ tt‖ss using consistent-restrict-impl by blast
      with s t have t ≅ hd(tt‖ss)
        apply clarify apply (erule con-sigs-inv) by auto
      with ssttss have t ≅ hd(ss‖tt) by simp
      with cons-sig-name have thn: ms-name t = ms-name(hd(ss‖tt)) by simp
      from shn thn stne have False by simp
```

thus *?thesis* **by** *simp*
qed
qed
qed
qed
qed

lemma *intersect-eq-implies-consistent-ty*:
$\sigma|\tau = \tau|\sigma \Longrightarrow \sigma \sim \tau$
using *intersect-eq-implies-consistent* **by** *blast*

lemma *consistent-iff-intersect-eq*:
$(\sigma \sim \tau) = (\sigma|\tau = \tau|(\sigma::ty))$
using *consistent-implies-intersect-eq*
intersect-eq-implies-consistent-ty
by *blast*

4.2 Type Merging

Like the type restriction operator, it is difficult to express type merging as a function in Isabelle, and we instead just define it using axioms.

consts
merge :: *ty* $\Rightarrow$ *ty* $\Rightarrow$ *ty* (**infixl** $\leftharpoonup$ *52*)
merge-sig :: *sig* $\Rightarrow$ *sig* $\Rightarrow$ *sig* (**infixl** $\leftharpoonup$: *52*)
merge-sigs :: *sig list* $\Rightarrow$ *sig list* $\Rightarrow$ *sig list* (**infixl** $\leftharpoonup$:: *52*)

lemma *mbool*[*simp*]: $BoolT \leftharpoonup \tau = (if\ \tau = ?\ then\ ?\ else\ BoolT)$ **sorry**
lemma *mint*[*simp*]: $IntT \leftharpoonup \tau = (if\ \tau = ?\ then\ ?\ else\ IntT)$ **sorry**
lemma *mfloat*[*simp*]: $FloatT \leftharpoonup \tau = (if\ \tau = ?\ then\ ?\ else\ FloatT)$ **sorry**
lemma *mfun*[*simp*]:
$(\sigma_1 \to \sigma_2) \leftharpoonup \tau =$
$(case\ \tau\ of\ IntT \Rightarrow (\sigma_1 \to \sigma_2) \mid FloatT \Rightarrow (\sigma_1 \to \sigma_2) \mid BoolT \Rightarrow (\sigma_1 \to \sigma_2)$
$\mid t_1 \to t_2 \Rightarrow (\sigma_1 \leftharpoonup t_1) \to (\sigma_2 \leftharpoonup t_2)$
$\mid ObjT\ tt \Rightarrow (\sigma_1 \to \sigma_2)$
$\mid\ ? \Rightarrow\ ?)$ **sorry**
lemma *mobj*[*simp*]:
$(ObjT\ ss) \leftharpoonup \tau =$
$(case\ \tau\ of\ IntT \Rightarrow (ObjT\ ss) \mid FloatT \Rightarrow (ObjT\ ss) \mid BoolT \Rightarrow (ObjT\ ss)$
$\mid \tau_1 \to \tau_2 \Rightarrow (ObjT\ ss)$
$\mid ObjT\ tt \Rightarrow ObjT\ (ss \leftharpoonup :: tt)$
$\mid\ ? \Rightarrow\ ?)$ **sorry**
lemma *munk*[*simp*]: $? \leftharpoonup \tau = \tau$ **sorry**

lemma *msig*[*simp*]:
$(Sig\ l\ \sigma) \leftharpoonup: (Sig\ l'\ \tau) = (if\ l = l'\ then\ Sig\ l\ (\sigma \leftharpoonup \tau)\ else\ Sig\ l\ \sigma)$ **sorry**

lemma *mnil1*[*simp*]: $[] \leftharpoonup :: tt = []$ **sorry**
lemma *mnil2*[*simp*]: $ss \leftharpoonup :: [] = ss$ **sorry**
lemma *mcons1*[*simp*]: $ms\text{-}name\ s < ms\text{-}name\ t \Longrightarrow (s\#ss) \leftharpoonup :: (t\#tt) = s\#(ss \leftharpoonup ::$

$(t\#tt)$) **sorry**
lemma *mcons2*[*simp*]: *ms-name s* = *ms-name t* ⟹ (*s*#*ss*)↼::(*t*#*tt*) = (*s*↼:*t*)#(*ss* ↼:: *tt*) **sorry**
lemma *mcons3*[*simp*]: *ms-name s* > *ms-name t* ⟹ (*s*#*ss*)↼::(*t*#*tt*) = ((*s*#*ss*) ↼:: *tt*) **sorry**

lemma *consistent-merge-impl*:
 (∀ ϱ τ. *n* = *size* ϱ + *size* τ ⟶ ϱ ∼ (ϱ ↼ τ))
 ∧ (∀ *r t*. *n* = *size r* + *size t* ⟶ *r* ≅ (*r* ↼: *t*))
 ∧ (∀ *rr tt*. *n* = *sig-list-size rr* + *sig-list-size tt* ⟶ *rr* ≈ (*rr* ↼:: *tt*)) (**is** *?P n*)
 apply (*induct rule*: *nat-less-induct*)
 apply (*rule conjI*)
  apply *clarify*
  apply (*case-tac* ϱ)
  apply *force*
  apply *force*
  apply *force*
  apply (*case-tac* τ) **apply** *force* **apply** *force* **apply** *force* **defer** **apply** *force* **apply** *force*
  apply (*case-tac* τ) **apply** *force* **apply** *force* **apply** *force* **apply** *force* **defer** **apply** *force*
  apply *force* **defer**
  apply (*rule-tac x=size ty1* + *size ty1a* **in** *allE*, *assumption*)
   apply *simp* **apply** (*erule conjE*) **apply** (*erule-tac x=ty1* **in** *allE*)
   apply (*erule-tac x=ty1a* **in** *allE*)
   apply (*erule-tac x=size ty2* + *size ty2a* **in** *allE*)
   apply *simp* **apply** (*erule conjE*) **apply** (*erule conjE*)
   apply (*erule-tac x=ty2* **in** *allE*)
   apply (*erule-tac x=ty2a* **in** *allE*)
   apply *force*
  apply (*erule-tac x=sig-list-size list* + *sig-list-size lista* **in** *allE*)
   apply *simp* **apply** (*erule conjE*)+
   apply (*erule-tac x=list* **in** *allE*)
   apply (*erule-tac x=lista* **in** *allE*)
   apply *force*
 apply (*rule conjI*)
  apply *clarify*
  apply (*case-tac r*) **apply** (*case-tac t*)
  apply (*erule-tac x=size ty* + *size tya* **in** *allE*)
   apply *simp* **apply** (*erule conjE*)+
   apply (*erule-tac x=ty* **in** *allE*)
   apply (*erule-tac x=tya* **in** *allE*)
   apply *force*
 apply *clarify*
  apply (*case-tac rr*) **apply** *force*
  apply (*case-tac tt*) **apply** *simp* **using** *consistent-reflexive* **apply** *simp*
  apply (*case-tac ms-name a* < *ms-name aa*)
   apply (*erule-tac x=sig-list-size list* + *sig-list-size* (*aa*#*lista*) **in** *allE*)
   apply *simp* **apply** (*erule conjE*)+

apply (*erule-tac x=list* **in** *allE*)
apply (*erule-tac x=aa#lista* **in** *allE*)
apply *simp* **apply** (*rule CConsT*) **using** *consistent-reflexive* **apply** *simp*
apply *assumption*
apply (*case-tac ms-name a = ms-name aa*)
apply (*rule-tac x=sig-list-size list + sig-list-size lista* **in** *allE*,*assumption*)
apply *simp* **apply** (*erule conjE*)+
apply (*erule-tac x=list* **in** *allE*)
apply (*erule-tac x=lista* **in** *allE*)
apply (*erule-tac x=size a + size aa* **in** *allE*)
apply *simp* **apply** (*erule conjE*)+
apply (*erule-tac x=a* **in** *allE*)
apply (*erule-tac x=a* **in** *allE*)
apply (*erule-tac x=aa* **in** *allE*)
apply (*erule-tac x=aa* **in** *allE*)
apply *simp* **apply** (*rule CConsT*) **apply** *assumption* **apply** *assumption*
apply (*erule-tac x=sig-list-size* (*a#list*) + *sig-list-size lista* **in** *allE*)
apply *auto*
done

lemma *consistent-merge*:
$\varrho \sim (\varrho \leftharpoondown \tau)$
using *consistent-merge-impl* **by** *simp*

4.3 Subtyping

consts
subtype :: (*ty* × *ty*) *set*
subtype-sig :: (*sig* × *sig*) *set*
subtype-sigs :: (*sig list* × *sig list*) *set*
syntax
subtype :: *ty* ⇒ *ty* ⇒ *bool* (**infixl** <: *51*)
subtype-sig :: *sig* ⇒ *sig* ⇒ *bool* (**infixl** ⪯ *51*)
subtype-sigs :: *sig list* ⇒ *sig list* ⇒ *bool* (**infixl** <:: *51*)
translations
$\sigma <: \tau == (\sigma, \tau) \in$ *subtype*
$\sigma \preceq \tau == (\sigma, \tau) \in$ *subtype-sig*
$\sigma <:: \tau == (\sigma, \tau) \in$ *subtype-sigs*
inductive *subtype subtype-sig subtype-sigs* **intros**
SIntInt[*intro*!]: *IntT* <: *IntT*
SBoolBool[*intro*!]: *BoolT* <: *BoolT*
SFF[*intro*!]: *FloatT* <: *FloatT*
SIntFloat[*intro*!]: *IntT* <: *FloatT*
SFun[*intro*!]: $[\![\tau <: \sigma; \sigma' <: \tau']\!] \Longrightarrow (\sigma \to \sigma') <: (\tau \to \tau')$
SUU[*intro*!]: *?* <: *?*
SObj[*intro*!]: *ss* <:: *tt* ⟹ *ObjT ss* <: *ObjT tt*

SSig[*intro*!]: $[\![l = l'; \tau = \tau']\!] \Longrightarrow$ *Sig* $l\ \tau \preceq$ *Sig* $l'\ \tau'$

SNil[*intro*!]: *ss* <:: []

SCons1[*intro*!]: $\llbracket$ *s* $\preceq$ *t*; *ss* <:: *tt* $\rrbracket \Longrightarrow$ (*s*#*ss*) <:: (*t*#*tt*)
SCons2[*intro*!]: $\llbracket$ *ms-name s* < *ms-name t*; *ss* <:: *t*#*tt* $\rrbracket \Longrightarrow$ (*s*#*ss*) <:: (*t*#*tt*)

inductive-cases *sub-fun-inv*[*elim*!]: $s \to s' <: t \to t'$
inductive-cases *sub-obj-inv*[*elim*!]: *ObjT ss* <: *ObjT tt*
inductive-cases *sub-sig-inv*[*elim*!]: *Sig l s* $\preceq$ *Sig l′ t*
inductive-cases *sub-sig-right-inv*[*elim*!]: *s* $\preceq$ *Sig l t*
inductive-cases *sub-sig-left-inv*[*elim*!]: *Sig l s* $\preceq$ *t*
inductive-cases *sub-sigs-inv*: *ss* <:: *tt*

theorem *subtype-reflexive*[*simp*]: $\sigma <: \sigma \wedge s \preceq s \wedge ss <:: ss$
apply (*induct rule*: *ty-sig.induct*)
apply *force* **apply** *force* **apply** *force* **apply** *force* **apply** *force* **apply** *force*
apply *force* **apply** *force* **apply** (*rule SCons1*) **apply** *auto* **done**

lemma *sub-sigs-reflexive*:
ms <:: *ms*
using *subtype-reflexive* **by** *simp*

lemma *subtype-trans*[*trans*]:
$\llbracket \varrho <: \sigma; \sigma <: \tau \rrbracket \Longrightarrow \varrho <: \tau$ **sorry**

lemma *subtype-sig-trans*[*trans*]:
$\llbracket \varrho \preceq \sigma; \sigma \preceq \tau \rrbracket \Longrightarrow \varrho \preceq \tau$
apply (*case-tac* ϱ) **apply** (*case-tac* σ) **apply** (*case-tac* τ) **by** *auto*

lemma *sub-sigs-trans*[*trans*]:
assumes *m12*: *ms1* <:: *ms2*
and *m23*: *ms2* <:: *ms3*
shows *ms1* <:: *ms3*
sorry

lemma *sub-obj-right-inv*:
σ <: *ObjT tt* $\Longrightarrow \exists$ *ss*. σ = *ObjT ss* $\wedge$ *ss* <:: *tt*
apply (*cases rule*: *subtype.cases*) **by** *auto*

lemma *sub-fun-right-inv*:
$\sigma <: \sigma' \to \tau' \Longrightarrow \exists\ s1\ s2.\ \sigma = s1 \to s2 \wedge \sigma' <: s1 \wedge s2 <: \tau'$
apply (*cases rule*: *subtype.cases*) **by** *auto*

lemma *merge-sub-sig*: (r ↼: t) $\preceq$ t $\Longrightarrow$ (r ↼: t) = t
apply (*case-tac t*) **apply** *simp*
apply (*erule sub-sig-right-inv*) **by** *auto*

lemma *sub-merge-sig*: t $\preceq$ (r ↼: t) $\Longrightarrow$ t = (r ↼: t)
apply (*case-tac t*) **apply** *simp*
apply (*erule sub-sig-left-inv*) **by** *auto*

lemma *restrict-sub-merge-impl*:
($\forall\ \varrho\ \tau.\ n = size\ \varrho + size\ \tau \longrightarrow (\varrho|\tau <: \tau|\varrho \longrightarrow (\varrho \leftharpoondown \tau) <: \tau) \wedge (\tau|\varrho <: \varrho|\tau \longrightarrow \tau <: (\varrho \leftharpoondown \tau))$)
$\wedge$ ($\forall\ r\ t.\ n = size\ r + size\ t \longrightarrow (r{\downarrow}t \preceq t{\downarrow}r \longrightarrow (r \leftharpoondown: t) \preceq t) \wedge (t{\downarrow}r \preceq r{\downarrow}t \longrightarrow t \preceq (r \leftharpoondown: t))$)
$\wedge$ ($\forall\ rr\ tt.\ n = sig\text{-}list\text{-}size\ rr + sig\text{-}list\text{-}size\ tt \longrightarrow$
$(rr\|tt <:: tt\|rr \longrightarrow (rr \leftharpoondown:: tt) <:: tt) \wedge (tt\|rr <:: rr\|tt \longrightarrow tt <:: (rr \leftharpoondown:: tt))$)
(**is** *?P n*)
proof (*induct rule*: *nat-less-induct*)
fix *n*
assume *IH*: $\forall m{<}n.$ *?P m*
show *?P n*
apply (*rule conjI*) **apply** (*rule allI*)+ **apply** (*rule impI*) **apply** (*rule conjI*) **defer defer**
apply (*rule conjI*) **apply** (*rule allI*)+ **apply** (*rule impI*) **apply** (*rule conjI*) **defer defer**
apply (*rule allI*)+ **apply** (*rule impI*) **apply** (*rule conjI*) **defer defer**
proof −
fix ϱ::*ty* **and** τ::*ty* **assume** *n*: $n = size\ \varrho + size\ \tau$
show $\varrho|\tau <: \tau|\varrho \longrightarrow (\varrho \leftharpoondown \tau) <: \tau$
apply *clarify*
apply (*case-tac* ϱ::*ty*)
apply (*case-tac* τ::*ty*) **apply** *simp* **apply** *simp* **apply** *simp* **apply** *simp* **apply** *simp* **apply** *simp*
apply (*case-tac* τ::*ty*) **apply** *simp* **apply** *simp* **apply** *simp* **apply** *simp* **apply** *simp* **apply** *simp*
apply (*case-tac* τ::*ty*) **apply** *simp* **apply** *simp* **apply** *simp* **apply** *simp* **apply** *simp* **apply** *simp*
apply (*case-tac* τ::*ty*) **apply** *simp* **apply** *simp* **apply** *simp* **defer** **apply** *simp* **apply** *simp*
apply (*case-tac* τ::*ty*) **apply** *simp* **apply** *simp* **apply** *simp* **apply** *simp* **defer** **apply** *simp*
apply *simp*
proof −
fix *r1 r2 t1 t2* **assume** *rttr*: $\varrho|\tau <: \tau|\varrho$ **and** *r*: $\varrho = r1 \rightarrow r2$ **and** *t*: $\tau = t1 \rightarrow t2$
from *rttr r t* **have** *rttr-sub*: $(r1|t1) \rightarrow (r2|t2) <: (t1|r1) \rightarrow (t2|r2)$ **by** *simp*
from *rttr-sub* **have** *trrt1*: $(t1|r1) <: (r1|t1)$ **by** *auto*
from *rttr-sub* **have** *rtrt2*: $(r2|t2) <: (t2|r2)$ **by** *auto*
let *?m1* = *size r1* + *size t1*
from *n t r* **have** *mn1*: *?m1* < *n* **by** *auto*
from *trrt1 mn1 IH* **have** *trt1*: $t1 <: (r1 \leftharpoondown t1)$ **by** *blast*
let *?m2* = *size r2* + *size t2*
from *n r t* **have** *mn2*: *?m2* < *n* **by** *auto*
from *rtrt2 mn2 IH* **have** *rtt2*: $(r2 \leftharpoondown t2) <: t2$ **by** *blast*
from *trt1 rtt2 r t* **show** $(\varrho \leftharpoondown \tau) <: \tau$ **by** *auto*
next
fix *rr tt* **assume** *rttr*: $\varrho|\tau <: \tau|\varrho$ **and** *r*: $\varrho = ObjT\ rr$ **and** *t*: $\tau = ObjT\ tt$
from *rttr r t* **have** *rttr-sub*: $(rr\|tt) <:: (tt\|rr)$ **by** *auto*
let *?m* = *sig-list-size rr* + *sig-list-size tt*

from *n t r* **have** *mn*: *?m* < *n* **by** *auto*
from *rttr-sub mn IH* **have** *rtt*: *rr* ↼:: *tt* <:: *tt* **by** *blast*
from *rtt r t* **show** ϱ ↼ τ <: τ **by** *auto*
qed
next
fix ϱ::*ty* **and** τ::*ty* **assume** *n*: *n* = *size* ϱ + *size* τ
show τ|ϱ <: ϱ|τ ⟶ τ <: ϱ ↼ τ
apply *clarify*
apply (*case-tac* ϱ::*ty*)
apply (*case-tac* τ::*ty*) **apply** *simp* **apply** *simp* **apply** *simp* **apply** *simp* **apply** *simp* **apply** *simp*
apply (*case-tac* τ::*ty*) **apply** *simp* **apply** *simp* **apply** *simp* **apply** *simp* **apply** *simp* **apply** *simp*
apply (*case-tac* τ::*ty*) **apply** *simp* **apply** *simp* **apply** *simp* **apply** *simp* **apply** *simp* **apply** *simp*
apply (*case-tac* τ::*ty*) **apply** *simp* **apply** *simp* **apply** *simp* **defer** **apply** *simp* **apply** *simp*
apply (*case-tac* τ::*ty*) **apply** *simp* **apply** *simp* **apply** *simp* **apply** *simp* **defer** **apply** *simp*
apply *simp*
proof −
fix *r1 r2 t1 t2* **assume** *rttr*: τ|ϱ <: ϱ|τ **and** *r*: ϱ = *r1* → *r2* **and** *t*: τ = *t1* → *t2*
from *rttr r t* **have** *rttr-sub*: (*t1*|*r1*) → (*t2*|*r2*) <: (*r1*|*t1*) → (*r2*|*t2*) **by** *simp*
from *rttr-sub* **have** *trrt1*: (*r1*|*t1*) <: (*t1*|*r1*) **by** *auto*
from *rttr-sub* **have** *rtrt2*: (*t2*|*r2*) <: (*r2*|*t2*) **by** *auto*
let *?m1* = *size r1* + *size t1*
from *n t r* **have** *mn1*: *?m1* < *n* **by** *auto*
from *trrt1 mn1 IH* **have** *trt1*: *r1* ↼ *t1* <: *t1* **by** *blast*
let *?m2* = *size r2* + *size t2*
from *n r t* **have** *mn2*: *?m2* < *n* **by** *auto*
from *rtrt2 mn2 IH* **have** *rtt2*: *t2* <: *r2* ↼ *t2* **by** *blast*
from *trt1 rtt2 r t* **show** τ <: ϱ ↼ τ **by** *auto*
next
fix *rr tt* **assume** *rttr*: τ|ϱ <: ϱ|τ **and** *r*: ϱ = *ObjT rr* **and** *t*: τ = *ObjT tt*
from *rttr r t* **have** *rttr-sub*: (*tt*‖*rr*) <:: (*rr*‖*tt*) **by** *auto*
let *?m* = *sig-list-size rr* + *sig-list-size tt*
from *n t r* **have** *mn*: *?m* < *n* **by** *auto*
from *rttr-sub mn IH* **have** *rtt*: *tt* <:: *rr* ↼:: *tt* **by** *blast*
from *rtt r t* **show** τ <: ϱ ↼ τ **by** *auto*
qed
next
fix *r*::*sig* **and** *t*::*sig* **assume** *n* = *size r* + *size t* **show** *r*↓*t* ⪯ *t*↓*r* ⟶ *r* ↼: *t* ⪯ *t* **sorry**
next
fix *r*::*sig* **and** *t*::*sig* **assume** *n* = *size r* + *size t* **show** *t*↓*r* ⪯ *r*↓*t* ⟶ *t* ⪯ *r* ↼: *t* **sorry**
next
fix *rr tt* **assume** *n* = *sig-list-size rr* + *sig-list-size tt*
show *rr*‖*tt* <:: *tt*‖*rr* ⟶ *rr* ↼:: *tt* <:: *tt* **sorry**
next

fix *rr tt* **assume** $n = \mathit{sig\text{-}list\text{-}size}\ rr + \mathit{sig\text{-}list\text{-}size}\ tt$
show $tt \| rr <:: rr \| tt \longrightarrow tt <:: rr \leftharpoonup:: tt$ **sorry**
qed
qed

constdefs $\mathit{subcons} :: ty \Rightarrow ty \Rightarrow bool$ (**infixl** $\lesssim$ *51*)
$\sigma \lesssim \tau \equiv \sigma|\tau <: \tau|\sigma$

lemma *restrict-sub-merge*:
$\varrho \lesssim \tau \Longrightarrow \varrho \leftharpoonup \tau <: \tau$
using *restrict-sub-merge-impl subcons-def* **by** *simp*

5 Operational Semantics

consts $\mathit{SimpleValues} :: expr \Rightarrow bool$
primrec
SimpleValues (*BVar i*) = *True*
SimpleValues (*FVar x*) = *True*
SimpleValues (*Const c*) = *True*
SimpleValues ($\lambda{:}\sigma.\ e$) = *True*
SimpleValues (*App e1 e2*) = *False*
SimpleValues (*Cast e s t*) = *False*
SimpleValues (*Obj ms* τ) = *True*
SimpleValues (*Invoke e l*) = *False*
SimpleValues (*Update e m*) = *False*

consts $\mathit{Values} :: expr \Rightarrow bool$
primrec
Values (*BVar i*) = *True*
Values (*FVar x*) = *True*
Values (*Const c*) = *True*
Values ($\lambda{:}\sigma.\ e$) = *True*
Values (*App e1 e2*) = *False*
Values (*Cast e s t*) = *SimpleValues e*
Values (*Obj ms* τ) = *True*
Values (*Invoke e l*) = *False*
Values (*Update e m*) = *False*

consts $\mathit{to\text{-}int} :: expr \Rightarrow int\ option$
primrec
to-int (*BVar x*) = *None*
to-int (*FVar x*) = *None*
to-int (*Const c*) =
(*case c of*
IntC n $\Rightarrow$ *Some n*
| *FloatC n* $\Rightarrow$ *None*
| *BoolC b* $\Rightarrow$ *None*
| *Succ* $\Rightarrow$ *None*

| *IsZero* ⇒ *None*)
to-int (*Lam* τ *e*) = *None*
to-int (*App e1 e2*) = *None*
to-int (*Cast e s t*) = *None*
to-int (*Obj ms* τ) = *None*
to-int (*Invoke e l*) = *None*
to-int (*Update e m*) = *None*

consts *delta* :: *const* ⇒ *expr* ⇒ *expr option* (δ)
primrec
delta (*IntC n*) *e* = *None*
delta (*FloatC n*) *e* = *None*
delta (*BoolC b*) *e* = *None*
delta Succ e =
(*case to-int e of*
None ⇒ *None*
| *Some n* ⇒ *Some* (*Const* (*IntC* (*n* + *1*))))
delta IsZero e =
(*case to-int e of*
None ⇒ *None*
| *Some n* ⇒ *Some* (*Const* (*BoolC* (*n* = *0*))))

consts *lookup* :: *method list* ⇒ *nat* ⇒ *method option*
primrec
lookup [] *l* = *None*
lookup (*m#ms*) *l* = (*if l* = *mname m then Some m else lookup ms l*)

consts *replace* :: *method list* ⇒ *method* ⇒ *method list*
primrec
replace-nil: *replace* [] *m'* = []
replace-cons: *replace* (*m#ms*) *m'* = (*if mname m* = *mname m' then m'#ms else m#*(*replace ms m'*))

constdefs *mcast* :: *expr* ⇒ *ty* ⇒ *ty* ⇒ *expr*
mcast e σ τ ≡ *if* $\sigma = \tau$ *then e else Cast e* σ τ

consts *reduces* :: (*expr* × *expr*) *set*
syntax *reduces* :: *expr* ⇒ *expr* ⇒ *bool* (**infixl** −→ *51*)
translations *e* −→ *e'* == (*e*,*e'*) ∈ *reduces*
inductive *reduces* **intros**
Beta: *Values v* ⟹ *App* (λ:τ. *e*) *v* −→ {*0*→*v*}*e*
Delta: ⟦ *Values v*; δ *c v* = *Some v'* ⟧ ⟹ *App* (*Const c*) *v* −→ *v'*
Sel: ⟦ *lookup ms l* = *Some* (*Method l e*) ⟧
⟹ *Invoke* (*Obj ms* τ) *l* −→ *App e* (*Obj ms* τ)
Upd: *Update* (*Obj ms* τ) *m* −→ *Obj* (*replace ms m*) τ
ApCst: ⟦ *SimpleValues* v_1; *Values* v_2 ⟧ ⟹
App ($v_1\langle(\sigma\to\tau)\Rightarrow(\varrho\to\nu)\rangle$) v_2 −→ *mcast* (*App* v_1 (*mcast* v_2 ϱ σ)) τ ν
SelCst: ⟦ *Values v*; *lookup-sig ss l* = *Some* (*Sig l* σ);
lookup-sig tt l = *Some* (*Sig l* τ) ⟧

$\Longrightarrow$ *Invoke* ($v\langle ObjT\ ss \Rightarrow ObjT\ tt\rangle$) $l \longrightarrow$ *mcast* (*Invoke v l*) σ τ
UpdCst: $[\![$ *Values v*; *lookup-sig ss l* = *Some* (*Sig l* σ);
lookup-sig tt l = *Some* (*Sig l* τ);
m' = *Method l* ($b\langle(ObjT\ tt \rightarrow\tau) \Rightarrow(ObjT\ ss \rightarrow \sigma)\rangle$) $]\!]$
$\Longrightarrow$ *Update* ($v\langle ObjT\ ss \Rightarrow ObjT\ tt\rangle$) (*Method l b*) $\longrightarrow$ (*Update v* m')$\langle ObjT\ ss$
$\Rightarrow ObjT\ tt\rangle$
Merge: $[\![$ *SimpleValues v*; $\varrho \lesssim \tau$; $\varrho \neq \tau$ $]\!]$
$\Longrightarrow$ ($v\langle\varrho\Rightarrow\sigma\rangle$)$\langle\sigma'\Rightarrow\tau\rangle \longrightarrow$ *mcast v* ϱ ($\varrho\leftharpoondown\tau$)
Remove: $[\![$ *SimpleValues v*; $\varrho = \tau$ $]\!]$
$\Longrightarrow$ ($v\langle\varrho\Rightarrow\sigma\rangle$)$\langle\sigma'\Rightarrow\tau\rangle \longrightarrow v$

constdefs *redex* :: *expr* $\Rightarrow$ *bool*
redex r $\equiv$ ($\exists$ r'. $r \longrightarrow r'$)

datatype *ctx* = *Hole* | *AppL ctx expr* | *AppR expr ctx* | *InvokeC ctx nat* | *UpdateC ctx method*
| *CastC ctx ty ty* (-⟨-⇒-⟩ [*53*,*53*,*53*] *52*)

consts *wf-ctx* :: *ctx set*
inductive *wf-ctx* **intros**
WFHole: *Hole* $\in$ *wf-ctx*
WFAppL: $E \in$ *wf-ctx* $\Longrightarrow$ *AppL E e* $\in$ *wf-ctx*
WFAppR: $[\![$ *Values v*; $E \in$ *wf-ctx* $]\!] \Longrightarrow$ *AppR v E* $\in$ *wf-ctx*
WFInvoke: $E \in$ *wf-ctx* $\Longrightarrow$ *InvokeC E l* $\in$ *wf-ctx*
WFUpdate: $E \in$ *wf-ctx* $\Longrightarrow$ *UpdateC E m* $\in$ *wf-ctx*
WFCastC: $E \in$ *wf-ctx* $\Longrightarrow$ *CastC E* σ τ $\in$ *wf-ctx*

consts *fill* :: *ctx* $\Rightarrow$ *expr* $\Rightarrow$ *expr* (-[-] [*82*,*82*] *81*)
primrec
Hole[*e*] = *e*
(*AppL E e2*)[*e*] = *App* (*E*[*e*]) *e2*
(*AppR e1 E*)[*e*] = *App e1* (*E*[*e*])
(*InvokeC E l*)[*e*] = *Invoke* (*E*[*e*]) *l*
(*UpdateC E m*)[*e*] = *Update* (*E*[*e*]) *m*
(*CastC E s t*)[*e*] = *Cast* (*E*[*e*]) *s t*

consts *eval-step* :: (*expr* $\times$ *expr*) *set*
syntax *eval-step* :: *expr* $\Rightarrow$ *expr* $\Rightarrow$ *bool* (**infixl** $\longmapsto$ *51*)
translations $e \longmapsto e'$ == (e,e') $\in$ *eval-step*
inductive *eval-step* **intros**
Step: $[\![$ $E \in$ *wf-ctx*; $r \longrightarrow r'$ $]\!] \Longrightarrow E[r] \longmapsto E[r']$

6 The Gradual Type System

lemma *lookup-implies-in-dom*:
lookup-sig ms l = *Some s* $\Longrightarrow$ $l \in$ *DomT ms*
apply (*induct ms*) **apply** *force* **apply** *force* **apply** *force* **apply** *force* **apply** *force*
apply *force* **apply** *force* **apply** *force* **apply** (*case-tac sig*) **apply** *simp*
apply (*case-tac l* = *nat*) **apply** (*simp add*: *ms-name-def*) **apply** *force*

apply (*simp add*: *ms-name-def*) **done**

consts
FV :: *expr* ⇒ *nat set*
FVm :: *method* ⇒ *nat set*
FVs :: *method list* ⇒ *nat set*
primrec
FV (*BVar i*) = {}
FV (*FVar x*) = {*x*}
FV (*Const c*) = {}
FV (λ:σ. *e*) = *FV e*
FV (*App e1 e2*) = *FV e1* ∪ *FV e2*
FV (*Obj ms* τ) = *FVs ms*
FV (*Invoke e l*) = *FV e*
FV (*Update e m*) = *FV e* ∪ *FVm m*
FV (*Cast e s t*) = *FV e*

FVm (*Method l e*) = *FV e*

FVs [] = {}
FVs (*m*#*ms*) = *FVm m* ∪ *FVs ms*

lemma *finite-FV-impl*: *finite* (*FV e*) ∧ *finite* (*FVm m*) ∧ *finite* (*FVs ms*)
apply (*induct rule*: *expr-method.induct*) **by** *auto*

lemma *finite-FV*: *finite* (*FV e*)
using *finite-FV-impl* **by** *simp*

types *env* = *nat* ⇒ *ty option*

constdefs *remove-bind* :: *env* ⇒ *nat* ⇒ *env* ⇒ *bool* (- − - ⊂ - [*50*,*50*,*50*] *49*)
$\Gamma - z \subset \Gamma' \equiv \forall\ x\ \tau.\ x \neq z \wedge \Gamma\ x = Some\ \tau \longrightarrow \Gamma'\ x = Some\ \tau$

constdefs *finite-env* :: *env* ⇒ *bool*
finite-env Γ ≡ *finite* (*dom* Γ)

consts *TypeOf* :: *const* ⇒ *ty*
primrec
TypeOf (*IntC n*) = *IntT*
TypeOf (*FloatC n*) = *FloatT*
TypeOf (*BoolC b*) = *BoolT*
TypeOf Succ = *IntT* → *IntT*
TypeOf IsZero = *IntT* → *BoolT*

consts
gt :: (*env* × *expr* × *ty*) *set*
gtm :: (*env* × *method* × *sig* × *ty*) *set*
gtms :: (*env* × *method list* × *sig list* × *ty*) *set*
syntax
gt :: *env* ⇒ *expr* ⇒ *ty* ⇒ *bool* (- $\vdash_G$ - : - [*52*,*52*,*52*] *51*)

gtm :: *env* ⇒ *method* ⇒ *sig* ⇒ *ty* ⇒ *bool* (- $\vdash_G$ - : - *in* - [*52*,*52*,*52*,*52*] *51*)
gtms :: *env* ⇒ *method list* ⇒ *sig* ⇒ *ty* ⇒ *bool* (- $\vdash_G$ - :: - *in* - [*52*,*52*,*52*,*52*] *51*)
translations
$\Gamma \vdash_G e : \tau$ == $(\Gamma, e, \tau) \in gt$
$\Gamma \vdash_G m : \tau\ in\ ot$ == $(\Gamma, m, \tau, ot) \in gtm$
$\Gamma \vdash_G ms :: \tau\ in\ ot$ == $(\Gamma, ms, \tau, ot) \in gtms$
inductive *gt gtm gtms* **intros**
GVar[*intro*!]: $\Gamma\ x = just\ \tau \Longrightarrow \Gamma \vdash_G (FVar\ x) : \tau$
GConst[*intro*!]: $\Gamma \vdash_G Const\ c : TypeOf\ c$
GLam[*intro*!]:
$[\![$ *finite L*; $\forall\ x.\ x \notin L \longrightarrow \Gamma(x \mapsto \sigma) \vdash_G \{0 \to FVar\ x\}e : \tau \wedge x \notin dom\ \Gamma$ $]\!]$
$\Longrightarrow \Gamma \vdash_G (\lambda{:}\sigma.\ e) : \sigma \to \tau$
GApp1[*intro*!]: $[\![\ \Gamma \vdash_G e_1 : ?;\ \Gamma \vdash_G e_2 : \tau_2\ ]\!]$
$\Longrightarrow \Gamma \vdash_G (App\ e_1\ e_2) : ?$
GApp2[*intro*!]: $[\![\ \Gamma \vdash_G e_1 : (\tau \to \tau');\ \Gamma \vdash_G e_2 : \tau_2;\ \tau_2 \lesssim \tau\ ]\!]$
$\Longrightarrow \Gamma \vdash_G (App\ e_1\ e_2) : \tau'$
GCast[*intro*!]: $[\![\ \Gamma \vdash_G e : \sigma;\ \sigma \lesssim \tau\ ]\!]$
$\Longrightarrow \Gamma \vdash_G Cast\ e\ \sigma\ \tau : \tau$
GSel1: $[\![\ \Gamma \vdash_G e : ?\ ]\!] \Longrightarrow \Gamma \vdash_G Invoke\ e\ l : ?$
GSel2: $[\![\ \Gamma \vdash_G e : ObjT\ ss;\ lookup\text{-}sig\ ss\ l = just\ (Sig\ l\ \tau)\ ]\!]$
$\Longrightarrow \Gamma \vdash_G Invoke\ e\ l : \tau$

GUpd1: $[\![\Gamma \vdash_G e : ?;\ \Gamma \vdash_G m : s\ in\ \tau\ ]\!]$
$\Longrightarrow \Gamma \vdash_G Update\ e\ m : ObjT\ [s]$
GUpd2: $[\![\Gamma \vdash_G e : ObjT\ ss;$
$\Gamma \vdash_G m : (Sig\ l\ \sigma)\ in\ ObjT\ ss;$
$lookup\text{-}sig\ ss\ l = just\ (Sig\ l\ \tau);\ \sigma \lesssim \tau\ ]\!]$
$\Longrightarrow \Gamma \vdash_G Update\ e\ m : ObjT\ ss$

GObj: $\Gamma \vdash_G ms :: ss\ in\ ObjT\ ss$
$\Longrightarrow \Gamma \vdash_G Obj\ ms\ (ObjT\ ss) : ObjT\ ss$

GMtd: $[\![\ \Gamma \vdash_G e : \sigma \to \tau;\ ObjT\ ss \lesssim \sigma;$
$lookup\text{-}sig\ ss\ l = just\ (Sig\ l\ \tau)\ ]\!]$
$\Longrightarrow \Gamma \vdash_G Method\ l\ e : (Sig\ l\ \tau)\ in\ ObjT\ ss$

GNil: $\Gamma \vdash_G [] :: []\ in\ \tau$
GCons: $[\![\ \Gamma \vdash_G m : s\ in\ \tau;\ \Gamma \vdash_G ms :: ss\ in\ \tau\ ]\!]$
$\Longrightarrow \Gamma \vdash_G (m\#ms) :: (s\#ss)\ in\ \tau$

7 Translation to Intermediate Language

consts
compile :: (*env* × *expr* × *expr* × *ty*) *set*
cm :: (*env* × *method* × *method* × *sig* × *ty*) *set*
cms :: (*env* × *method list* × *method list* × *sig list* × *ty*) *set*
syntax
compile :: *env* ⇒ *expr* ⇒ *expr* ⇒ *ty* ⇒ *bool* (- ⊢ - ⇒ - : - [*52*,*52*,*52*,*52*] *51*)
cm :: *env* ⇒ *method* ⇒ *method* ⇒ *sig* ⇒ *ty* ⇒ *bool* (- ⊢ - ⇒ - : - *in* - [*52*,*52*,*52*,*52*,*52*]

51)

$cms :: env \Rightarrow method\ list \Rightarrow method\ list \Rightarrow sig \Rightarrow ty \Rightarrow bool$ (- $\vdash$ - $\Rightarrow$ - :: - *in* - [52,52,52,52,52] 51)

translations

$\Gamma \vdash e \Rightarrow e' : \tau == (\Gamma,\ e,\ e',\ \tau) \in compile$

$\Gamma \vdash m \Rightarrow m' : s\ in\ \tau == (\Gamma,\ m,\ m',\ s,\ \tau) \in cm$

$\Gamma \vdash ms \Rightarrow ms' :: ss\ in\ \tau == (\Gamma,\ ms,\ ms',\ ss,\ \tau) \in cms$

inductive *compile cm cms* **intros**

CVar[*intro*!]: $\Gamma\ x = just\ \tau \Longrightarrow \Gamma \vdash FVar\ x \Rightarrow FVar\ x : \tau$

CConst[*intro*!]: $\Gamma \vdash Const\ c \Rightarrow Const\ c : TypeOf\ c$

CLam[*intro*!]: $[\![\ finite\ L;\ \forall\ x.\ x \notin L \longrightarrow \Gamma(x{\mapsto}\sigma) \vdash \{0{\to}FVar\ x\}e \Rightarrow \{0{\to}FVar\ x\}e' : \tau \wedge x \notin dom\ \Gamma\ ]\!]$

$\Longrightarrow \Gamma \vdash (\lambda{:}\sigma.\ e) \Rightarrow (\lambda{:}\sigma.\ e') : (\sigma \to \tau)$

CApp1[*intro*!]: $[\![\ \Gamma \vdash e_1 \Rightarrow e'_1 : ?;\ \Gamma \vdash e_2 \Rightarrow e'_2 : \tau_2\ ]\!] \Longrightarrow$

$\Gamma \vdash (App\ e_1\ e_2) \Rightarrow (App\ (mcast\ e'_1\ ?\ (\tau_2 \to ?))\ e'_2) : ?$

CApp2[*intro*!]: $[\![\ \Gamma \vdash e_1 \Rightarrow e'_1 : (\tau \to \tau');$

$\Gamma \vdash e_2 \Rightarrow e'_2 : \tau_2;\ \tau_2 \lesssim \tau\ ]\!] \Longrightarrow$

$\Gamma \vdash (App\ e_1\ e_2) \Rightarrow (App\ e'_1\ (mcast\ e'_2\ \tau_2\ (\tau_2 \leftharpoondown \tau))) : \tau'$

CCast[*intro*!]: $[\![\ \Gamma \vdash e \Rightarrow e' : \sigma;\ \sigma \lesssim \tau\ ]\!] \Longrightarrow$

$\Gamma \vdash Cast\ e\ \sigma\ \tau \Rightarrow mcast\ e'\ \sigma\ (\sigma \leftharpoondown \tau) : \tau$

CSel1: $[\![\ \Gamma \vdash e \Rightarrow e' : ?\ ]\!]$

$\Longrightarrow \Gamma \vdash Invoke\ e\ l \Rightarrow Invoke\ (mcast\ e'\ ?\ (ObjT\ [Sig\ l\ ?]))\ l : ?$

CSel2: $[\![\ \Gamma \vdash e \Rightarrow e' : ObjT\ ss;\ lookup\text{-}sig\ ss\ l = just\ (Sig\ l\ \tau)\ ]\!]$

$\Longrightarrow \Gamma \vdash Invoke\ e\ l \Rightarrow Invoke\ e'\ l : \tau$

CUpd1: $[\![\Gamma \vdash e \Rightarrow e' : ?;\ \Gamma \vdash m \Rightarrow m' : s\ in\ ObjT\ [s]\ ]\!]$

$\Longrightarrow \Gamma \vdash Update\ e\ m \Rightarrow Update\ (mcast\ e'\ ?\ (ObjT\ [s]))\ m' : ObjT\ [s]$

CUpd2: $[\![\Gamma \vdash e \Rightarrow e' : ObjT\ ss;$

$\Gamma \vdash m \Rightarrow (Method\ l\ b) : (Sig\ l\ \sigma)\ in\ ObjT\ ss;$

$lookup\text{-}sig\ ss\ l = just\ (Sig\ l\ \tau);\ \sigma \lesssim \tau\ ]\!]$

$\Longrightarrow \Gamma \vdash Update\ e\ m \Rightarrow Update\ e'\ (Method\ l\ (mcast\ b\ (ObjT\ ss{\to}\sigma)\ (ObjT\ ss{\to}(\sigma{\leftharpoondown}\tau)))) : ObjT\ ss$

CObj: $\Gamma \vdash ms \Rightarrow ms' :: ss\ in\ ObjT\ ss$

$\Longrightarrow \Gamma \vdash Obj\ ms\ (ObjT\ ss) \Rightarrow Obj\ ms'\ (ObjT\ ss) : ObjT\ ss$

CMtd: $[\![\ \Gamma \vdash e \Rightarrow e' : \sigma \to \tau;\ ObjT\ ss \lesssim \sigma;$

$lookup\text{-}sig\ ss\ l = just\ (Sig\ l\ \tau)\ ]\!]$

$\Longrightarrow \Gamma \vdash Method\ l\ e \Rightarrow Method\ l\ (mcast\ e'\ (\sigma \to \tau)\ ((\sigma{\leftharpoondown}ObjT\ ss) \to \tau)) : (Sig\ l\ \tau)\ in\ ObjT\ ss$

CNil: $\Gamma \vdash [] \Rightarrow [] :: []\ in\ \tau$

CCons: $[\![\ \Gamma \vdash m \Rightarrow m' : s\ in\ \tau;\ \Gamma \vdash ms \Rightarrow ms' :: ss\ in\ \tau\ ]\!]$

$\Longrightarrow \Gamma \vdash (m\#ms) \Rightarrow (m'\#ms') :: (s\#ss)\ in\ \tau$

7.1 Type System for Intermediate Language

consts

$wte :: (env \times expr \times ty)\ set$

$wtm :: (env \times method \times sig \times ty)\ set$

$wtms :: (env \times method\ list \times sig\ list \times ty)\ set$

syntax

$wte :: env \Rightarrow [expr,ty] \Rightarrow bool$ (- $\vdash$ - : - [52,52,52] 51)

wtm :: *env* ⇒ [*method*,*sig*,*ty*] ⇒ *bool* (- ⊢ - : - *in* - [*52*,*52*,*52*,*52*] *51*)
wtms :: *env* ⇒ [*method list*,*sig list*,*ty*] ⇒ *bool* (- ⊢ - :: - *in* - [*52*,*52*,*52*,*52*] *51*)

translations

$\Gamma \vdash e : \tau \rightleftharpoons (\Gamma,\ e,\ \tau) \in wte$
$\Gamma \vdash m : \sigma\ in\ \tau \rightleftharpoons (\Gamma,\ m,\ \sigma,\ \tau) \in wtm$
$\Gamma \vdash ms :: ss\ in\ \tau \rightleftharpoons (\Gamma,\ ms,\ ss,\ \tau\ ) \in wtms$

inductive *wte wtm wtms* **intros**

wte-var: $\Gamma\ x = just\ \tau \Longrightarrow \Gamma \vdash FVar\ x : \tau$
wte-const: $\Gamma \vdash Const\ c : TypeOf\ c$
wte-abs:
$[\![\ finite\ L;\ \forall\ x.\ x \notin L \longrightarrow \Gamma(x{\mapsto}\sigma) \vdash \{0{\to}FVar\ x\}e : \tau \wedge x \notin dom\ \Gamma\ ]\!]$
$\Longrightarrow \Gamma \vdash (\lambda{:}\sigma.\ e) : \sigma \to \tau$
wte-app: $[\![\ \Gamma \vdash e_1 : \sigma \to \tau;\ \Gamma \vdash e_2 : \sigma\ ]\!]$
$\Longrightarrow \Gamma \vdash App\ e_1\ e_2 : \tau$
wte-sub: $[\![\ \Gamma \vdash e : \sigma;\ \sigma <: \tau\ ]\!] \Longrightarrow \Gamma \vdash e : \tau$
wte-cast: $[\![\ \Gamma \vdash e : \sigma;\ \sigma \sim \tau;\ \sigma \neq \tau\ ]\!] \Longrightarrow \Gamma \vdash e\langle\sigma{\Rightarrow}\tau\rangle : \tau$

wte-sel: $[\![\ \Gamma \vdash e : ObjT\ ss;\ lookup\text{-}sig\ ss\ l = just\ (Sig\ l\ \tau)\ ]\!]$
$\Longrightarrow \Gamma \vdash Invoke\ e\ l : \tau$

wte-upd: $[\![\Gamma \vdash e : ObjT\ ss;$
$\Gamma \vdash m : s\ in\ ObjT\ ss;$
$lookup\text{-}sig\ ss\ l = just\ s\ ]\!]$
$\Longrightarrow \Gamma \vdash Update\ e\ m : ObjT\ ss$

wte-obj: $\Gamma \vdash ms :: ss\ in\ ObjT\ ss \Longrightarrow \Gamma \vdash Obj\ ms\ (ObjT\ ss) : ObjT\ ss$

wt-mtd: $[\![\ \Gamma \vdash e : (ObjT\ ss) \to \tau;$
$lookup\text{-}sig\ ss\ l = just\ (Sig\ l\ \tau)\ ]\!]$
$\Longrightarrow \Gamma \vdash Method\ l\ e : (Sig\ l\ \tau)\ in\ ObjT\ ss$

wt-nil: $\Gamma \vdash [] :: []\ in\ \tau$
wt-cons: $[\![\ \Gamma \vdash m : s\ in\ \tau;\ \Gamma \vdash ms :: ss\ in\ \tau\ ]\!]$
$\Longrightarrow \Gamma \vdash (m\#ms) :: (s\#ss)\ in\ \tau$

inductive-cases *wt-mtd-inv*[*elim*!]:
$\Gamma \vdash (Method\ l\ b) : (Sig\ l\ \sigma)\ in\ \varrho$

lemma *restrict-sub-merge2*:
$\tau \lesssim \varrho \Longrightarrow \tau <: \varrho{\leftharpoonup}\tau$
using *restrict-sub-merge-impl subcons-def* **by** *simp*

7.2 The Translation is Sound

lemma *compilation-sound-impl*:
$(\Gamma \vdash e \Rightarrow e' : \tau \longrightarrow \Gamma \vdash e' : \tau)$
$\wedge (\Gamma \vdash m \Rightarrow m' : s\ in\ \tau \longrightarrow \Gamma \vdash m' : s\ in\ \tau)$
$\wedge (\Gamma \vdash ms \Rightarrow ms' :: ss\ in\ \tau \longrightarrow \Gamma \vdash ms' :: ss\ in\ \tau)$
apply (*induct rule*: *compile-cm-cms.induct*)
using *wte-var* **apply** *simp*

using *wte-const* **apply** *simp*
apply (*rule wte-abs*) **apply** *simp* **apply** *clarify*
apply (*erule-tac x=x* **in** *allE*)
apply (*erule impE*) **apply** *simp* **apply** (*erule conjE*)+
apply (*rule conjI*) **apply** *assumption* **apply** *assumption*
apply (*simp add*: *mcast-def*) **using** *wte-app wte-cast* **apply** *blast*
apply (*simp add*: *mcast-def*)
apply (*rule conjI*) **apply** *clarify* **apply** (*rule wte-app*) **apply** *simp*
apply (*rule wte-sub*) **apply** *simp* **using** *restrict-sub-merge* **apply** *force*
apply *clarify* **apply** (*rule wte-app*) **apply** *simp* **apply** (*rule wte-sub*)
apply (*rule wte-cast*) **apply** *simp* **apply** (*rule consistent-merge*) **apply** *simp*
apply (*rule restrict-sub-merge*) **apply** *simp*
apply (*simp add*: *mcast-def*) **apply** (*rule conjI*) **apply** *clarify*
apply (*rule wte-sub*) **apply** *simp* **using** *restrict-sub-merge* **apply** *force*
apply *clarify* **apply** (*rule wte-sub*) **apply** (*rule wte-cast*) **apply** *simp*
apply (*rule consistent-merge*) **apply** *simp*
apply (*rule restrict-sub-merge*) **apply** *simp*
apply (*simp add*: *mcast-def*) **apply** (*rule wte-sel*) **apply** (*rule wte-cast*) **apply** *simp* **apply** *force*
apply *simp* **apply** (*simp add*: *ms-name-def*)
using *wte-sel* **apply** *simp*
apply (*simp add*: *mcast-def*) **apply** (*rule wte-upd*) **apply** (*rule wte-cast*) **apply** *simp* **apply** *force* **apply** *simp*
apply *simp* **apply** *simp*
defer
using *wte-obj* **apply** *simp*
defer
apply (*rule wt-nil*)
apply (*rule wt-cons*) **apply** *simp* **apply** *simp*
proof −
fix $\Gamma\ \sigma\ \tau\ b\ e\ e'\ l\ m\ ss$
assume *ce*: $\Gamma \vdash e \Rightarrow e' : ObjT\ ss$ **and** *ep*: $\Gamma \vdash e' : ObjT\ ss$ **and** *wtm*: $\Gamma \vdash m \Rightarrow (Method\ l\ b) : (Sig\ l\ \sigma)\ in\ ObjT\ ss$
and *wts*: $\Gamma \vdash (Method\ l\ b) : (Sig\ l\ \sigma)\ in\ ObjT\ ss$ **and** *ssl*: $lookup\text{-}sig\ ss\ l = just\ (Sig\ l\ \tau)$
and *st*: $\sigma \lesssim \tau$

from *st* **have** *stt*: $(\sigma \leftharpoondown \tau) <: \tau$ **by** (*rule restrict-sub-merge*)
let $?ct = ObjT\ ss \to (\sigma \leftharpoondown \tau)$
from *stt* **have** *sub*: $?ct <: (ObjT\ ss \to \tau)$ **by** *auto*
have *sim*: $(ObjT\ ss \to \sigma) \sim\ ?ct$ **using** *consistent-merge* **by** *force*
from *wts* **have** *wtb*: $\Gamma \vdash b : ObjT\ ss \to \sigma$ **by** *auto*
from *wtb sim* **have** *wtc*: $\Gamma \vdash mcast\ b\ (ObjT\ ss \to \sigma)\ ?ct : ?ct$ **using** *mcast-def wte-cast* **by** *auto*
from *wtc sub* **have** *wtc2*: $\Gamma \vdash mcast\ b\ (ObjT\ ss \to \sigma)\ ?ct : (ObjT\ ss \to \tau)$ **by** (*rule wte-sub*)
let $?m = Method\ l\ (mcast\ b\ (ObjT\ ss \to \sigma)\ ?ct)$
from *wtc2 ssl* **have** *wtm*: $\Gamma \vdash\ ?m : (Sig\ l\ \tau)\ in\ ObjT\ ss$ **by** (*rule wt-mtd*)
from *ep wtm ssl* **show** $\Gamma \vdash Update\ e'\ ?m : ObjT\ ss$ **by** (*rule wte-upd*)
next

fix $\Gamma\ \sigma\ \tau\ e\ e'\ l\ ss$
assume $\Gamma \vdash e \Rightarrow e' : \sigma \rightarrow \tau$ **and** *ep*: $\Gamma \vdash e' : \sigma \rightarrow \tau$ **and** *ss*: $ObjT\ ss \lesssim \sigma$
and *ssl*: *lookup-sig ss l* = *just* (*Sig l* τ)
let *?ct* = $((\sigma \leftharpoondown ObjT\ ss) \rightarrow \tau)$
from *ss* **have** *sub1*: $ObjT\ ss <: \sigma \leftharpoondown ObjT\ ss$ **by** (*rule restrict-sub-merge2*)
hence *sub*: $?ct <: (ObjT\ ss \rightarrow \tau)$ **using** *subtype-reflexive* **by** *auto*
have *sim*: $(\sigma \rightarrow \tau) \sim ?ct$ **using** *consistent-merge* **by** *blast*
from *ep sim* **have** *wtc*: $\Gamma \vdash (mcast\ e'\ (\sigma{\rightarrow}\tau)\ ?ct) : ?ct$ **using** *wte-cast mcast-def* **by** *auto*
from *wtc sub* **have** *wtc2*: $\Gamma \vdash (mcast\ e'\ (\sigma{\rightarrow}\tau)\ ?ct) : (ObjT\ ss \rightarrow \tau)$ **by** (*rule wte-sub*)
from *wtc2 ssl* **show** $\Gamma \vdash Method\ l\ (mcast\ e'\ (\sigma{\rightarrow}\tau)\ ?ct) : (Sig\ l\ \tau)\ in\ ObjT\ ss$ **by** (*rule wt-mtd*)
qed

theorem *compilation-sound*:
$\Gamma \vdash e \Rightarrow e' : \tau \Longrightarrow \Gamma \vdash e' : \tau$
using *compilation-sound-impl* **by** *blast*

7.3 Sound and Complete with Respect to $FOb_{<:}$

consts
fob-type :: *ty set*
fob-sig :: *sig set*
fob-sigs :: (*sig list*) *set*
inductive *fob-type fob-sig fob-sigs* **intros**
FObInt[*intro*!]: $IntT \in fob\text{-}type$
FObFloat[*intro*!]: $FloatT \in fob\text{-}type$
FObBool[*intro*!]: $BoolT \in fob\text{-}type$
FObArrow[*intro*!]: $[\![\ \tau_1 \in fob\text{-}type;\ \tau_2 \in fob\text{-}type\ ]\!] \Longrightarrow$
$(\tau_1 \rightarrow \tau_2) \in fob\text{-}type$
FObObjT[*intro*!]: $ss \in fob\text{-}sigs \Longrightarrow ObjT\ ss \in fob\text{-}type$

FObSig[*intro*!]: $\tau \in fob\text{-}type \Longrightarrow Sig\ l\ \tau \in fob\text{-}sig$

FObNilT[*intro*!]: $[] \in fob\text{-}sigs$
FObConsT[*intro*!]: $[\![\ s \in fob\text{-}sig;\ ss \in fob\text{-}sigs\ ]\!] \Longrightarrow s\#ss \in fob\text{-}sigs$

inductive-cases *fob-unk-inv*[*elim*!]: $? \in fob\text{-}type$
inductive-cases *fob-fun-inv*[*elim*!]: $\sigma \rightarrow \tau \in fob\text{-}type$
inductive-cases *fob-objt-inv*[*elim*!]: $ObjT\ ss \in fob\text{-}type$
inductive-cases *fob-sig-inv*[*elim*!]: $Sig\ l\ \tau \in fob\text{-}sig$
inductive-cases *fob-sigs-inv*[*elim*!]: $ss \in fob\text{-}sigs$
inductive-cases *fob-cons-inv*[*elim*!]: $s\#ss \in fob\text{-}sigs$

consts
fob-term :: *expr set*
fob-method :: *method set*
fob-methods :: *method list set*
inductive *fob-term fob-method fob-methods* **intros**

$FObFVar[intro!]$: $(FVar\ x) \in fob\text{-}term$
$FObBVar[intro!]$: $(BVar\ x) \in fob\text{-}term$
$FObConst[intro!]$: $(Const\ c) \in fob\text{-}term$
$FObLam[intro!]$: $[\![\ \tau \in fob\text{-}type;\ e \in fob\text{-}term\ ]\!] \Longrightarrow$
$(Lam\ \tau\ e) \in fob\text{-}term$
$FObApp[intro!]$: $[\![\ e_1 \in fob\text{-}term;\ e_2 \in fob\text{-}term\ ]\!] \Longrightarrow$
$(App\ e_1\ e_2) \in fob\text{-}term$
$FObObj[intro!]$: $[\![\ ms \in fob\text{-}methods;\ \tau \in fob\text{-}type\ ]\!] \Longrightarrow Obj\ ms\ \tau \in fob\text{-}term$
$FObSel[intro!]$: $e \in fob\text{-}term \Longrightarrow Invoke\ e\ l \in fob\text{-}term$
$FObUpd[intro!]$: $[\![\ e \in fob\text{-}term;\ m \in fob\text{-}method\ ]\!] \Longrightarrow Update\ e\ m \in fob\text{-}term$

$FObMtd[intro!]$: $e \in fob\text{-}term \Longrightarrow Method\ l\ e \in fob\text{-}method$
$FObNil[intro!]$: $[] \in fob\text{-}methods$
$FObCons[intro!]$: $[\![\ m \in fob\text{-}method;\ ms \in fob\text{-}methods\ ]\!]$
$\Longrightarrow m\#ms \in fob\text{-}methods$

inductive-cases *fob-lam-inv*[*elim!*]: $\lambda{:}\tau.\ e \in fob\text{-}term$
inductive-cases *fob-app-inv*[*elim!*]: $App\ e\ e' \in fob\text{-}term$
inductive-cases *fob-obj-inv*[*elim!*]: $Obj\ ms\ \tau \in fob\text{-}term$
inductive-cases *fob-cast-inv*[*elim!*]: $Cast\ e\ s\ t \in fob\text{-}term$
inductive-cases *fob-sel-inv*[*elim!*]: $Invoke\ e\ l \in fob\text{-}term$
inductive-cases *fob-upd-inv*[*elim!*]: $Update\ e\ m \in fob\text{-}term$
inductive-cases *fob-obj-inv*[*elim!*]: $Obj\ ms\ \tau \in fob\text{-}term$
inductive-cases *fob-mtd-inv*[*elim!*]: $Method\ l\ e \in fob\text{-}method$
inductive-cases *fob-nil-inv*[*elim!*]: $[] \in fob\text{-}methods$
inductive-cases *fob-cons-inv*[*elim!*]: $m\#ms \in fob\text{-}methods$

lemma *fob-subst*: $e \in fob\text{-}term \Longrightarrow \{i{\rightarrow}FVar\ x\}e \in fob\text{-}term$
sorry

lemma *consistent-fob-eq-impl*:
$(\sigma \sim \tau \longrightarrow \sigma \in fob\text{-}type \wedge \tau \in fob\text{-}type \longrightarrow \sigma = \tau)$
$\wedge\ (s \cong t \longrightarrow s \in fob\text{-}sig \wedge t \in fob\text{-}sig \longrightarrow s = t)$
$\wedge\ (ss \approx tt \longrightarrow ss \in fob\text{-}sigs \wedge tt \in fob\text{-}sigs \longrightarrow ss = tt)$
apply (*induct rule*: *consistent-consistent-sig-consistent-sigs.induct*)
apply *force* **apply** *force* **apply** *force* **apply** *force* **apply** *force* **apply** *force*
apply *force* **apply** *clarify* **apply** *blast* **done**

lemma *consistent-fob-eq*:
$\tau \sim \tau' \Longrightarrow \tau \in fob\text{-}type \wedge \tau' \in fob\text{-}type \longrightarrow \tau = \tau'$
using *consistent-fob-eq-impl* **by** *blast*

lemma *consistent-fob-noteq*:
$\sigma \sim \tau \Longrightarrow \sigma \in fob\text{-}type \wedge \sigma \neq \tau \longrightarrow \tau \notin fob\text{-}type$
using *consistent-fob-eq* **by** *blast*

lemma *restrict-fob-impl*:
$(\forall\ \tau.\ \sigma \in fob\text{-}type \wedge \tau \in fob\text{-}type \longrightarrow \sigma|\tau = \sigma \wedge \tau|\sigma = \tau)$
$\wedge\ (\forall\ t.\ s \in fob\text{-}sig \wedge t \in fob\text{-}sig \longrightarrow s{\downarrow}t = s \wedge t{\downarrow}s = t)$
$\wedge\ (\forall\ tt.\ ss \in fob\text{-}sigs \wedge tt \in fob\text{-}sigs \longrightarrow ss\|tt = ss \wedge tt\|ss = tt)$

apply (*induct rule*: *ty-sig.induct*)
apply *clarify* **apply** (*case-tac* τ) **apply** *force* **apply** *force* **apply** *force* **apply** *force* **apply** *force* **apply** *force*
apply *clarify* **apply** (*case-tac* τ) **apply** *force* **apply** *force* **apply** *force* **apply** *force* **apply** *force* **apply** *force*
apply *clarify* **apply** (*case-tac* τ) **apply** *force* **apply** *force* **apply** *force* **apply** *force* **apply** *force* **apply** *force*
apply *clarify* **apply** (*case-tac* τ) **apply** *force* **apply** *force* **apply** *force* **apply** *force* **apply** *force* **apply** *force*
apply *clarify* **apply** (*case-tac* τ) **apply** *force* **apply** *force* **apply** *force* **apply** *force* **apply** *force* **apply** *force*
apply *force*
apply *clarify* **apply** (*case-tac t*) **apply** *simp* **apply** *clarify* **apply** *simp*
apply *clarify* **apply** (*case-tac tt*) **apply** *simp* **apply** *simp*
apply *clarify* **apply** (*case-tac tt*) **apply** *simp*
apply *clarify* **apply** (*erule-tac x=a* **in** *allE*) **apply** (*erule-tac x=lista* **in** *allE*)
apply (*erule impE*) **apply** *simp* **apply** (*erule impE*) **apply** *simp*
apply *clarify* **sorry**

lemma *restrict-fob*:
$[\![\ \sigma \in \textit{fob-type};\ \tau \in \textit{fob-type}\ ]\!] \Longrightarrow \sigma|\tau = \sigma$
using *restrict-fob-impl* **by** *blast*

lemma *subcon-fob-sub*:
$[\![\ \tau \lesssim \tau';\ \tau \in \textit{fob-type};\ \tau' \in \textit{fob-type}\ ]\!] \Longrightarrow \tau <: \tau'$
using *restrict-fob subcons-def* **by** *force*

lemma *lookup-fob*: $[\![\ ss \in \textit{fob-sigs};\ \textit{lookup-sig}\ ss\ l = \textit{just}\ (\textit{Sig}\ l\ \tau)\ ]\!] \Longrightarrow \tau \in \textit{fob-type}$
sorry

lemma *merge-fob*:
$[\![\ \sigma \in \textit{fob-type};\ \tau \in \textit{fob-type}\ ]\!] \Longrightarrow \sigma \leftharpoondown \tau = \sigma$ **sorry**

lemma *merge-fob-neq*:
$[\![\ \sigma \in \textit{fob-type};\ \tau \in \textit{fob-type};\ \sigma \neq \sigma \leftharpoondown \tau\ ]\!] \Longrightarrow \textit{False}$
using *merge-fob* **by** *auto*

lemma *gradual-soundness-fob-impl*:
$(\Gamma \vdash_G e : \tau \longrightarrow$
$e \in \textit{fob-term} \land (\forall\ x\ \tau.\ \Gamma\ x = \textit{just}\ \tau \longrightarrow \tau \in \textit{fob-type}) \longrightarrow$
$\Gamma \vdash e : \tau \land \tau \in \textit{fob-type})$
$\land (\Gamma \vdash_G m : s\ \textit{in}\ \tau \longrightarrow$
$m \in \textit{fob-method} \land \tau \in \textit{fob-type} \land (\forall\ x\ \tau.\ \Gamma\ x = \textit{just}\ \tau \longrightarrow \tau \in \textit{fob-type}) \longrightarrow$
$\Gamma \vdash m : s\ \textit{in}\ \tau \land s \in \textit{fob-sig})$
$\land (\Gamma \vdash_G ms :: ss\ \textit{in}\ \tau \longrightarrow$
$ms \in \textit{fob-methods} \land \tau \in \textit{fob-type} \land (\forall\ x\ \tau.\ \Gamma\ x = \textit{just}\ \tau \longrightarrow \tau \in \textit{fob-type}) \longrightarrow$
$\Gamma \vdash ms :: ss\ \textit{in}\ \tau \land ss \in \textit{fob-sigs})$
(**is** $(\Gamma \vdash_G e : \tau \longrightarrow ?P\ \Gamma\ e\ \tau) \land (\Gamma \vdash_G m : s\ \textit{in}\ \tau \longrightarrow ?PM\ \Gamma\ m\ s\ \tau) \land (\Gamma \vdash_G ms :: ss\ \textit{in}\ \tau \longrightarrow ?PS\ \Gamma\ ms\ ss\ \tau)$)
apply (*induct rule*: *gt-gtm-gtms.induct*)

using *wte-var* **apply** *blast*
apply *clarify* **apply** (*rule conjI*) **apply** (*rule wte-const*)
apply (*case-tac c*) **apply** *force* **apply** *force* **apply** *force* **apply** *force*
apply *force*
apply *clarify*
apply (*rule conjI*)
apply (*rule wte-abs*) **apply** *simp*
apply *clarify* **apply** (*erule-tac x=x* **in** *allE*) **apply** (*erule impE*)
apply *simp* **apply** (*erule conjE*)+
apply (*erule impE*) **apply** (*rule conjI*)
apply (*rule fob-subst*) **apply** *simp*
apply (*rule allI*)+ **apply** (*rule impI*)
apply (*erule-tac x=xa* **in** *allE*) **apply** (*erule-tac x=τ'* **in** *allE*)
apply (*case-tac x = xa*) **apply** *simp* **apply** *simp*
apply (*rule conjI*) **apply** (*erule conjE*) **apply** *assumption* **apply** *assumption*
apply (*erule-tac x=Suc* (*setmax L*) **in** *allE*)
apply (*erule impE*) **apply** (*rule max-is-fresh*) **apply** *simp*
apply (*erule conjE*)+ **apply** (*erule impE*) **apply** (*rule conjI*) **apply** (*rule fob-subst*) **apply** *simp*
apply (*rule allI*)+ **apply** (*rule impI*) **apply** (*case-tac x = Suc* (*setmax L*)) **apply** *simp* **apply** *simp* **apply** *blast*
apply *force*
apply *clarify* **apply** (*rule conjI*) **apply** (*rule wte-app*) **apply** *force* **apply** (*erule impE*) **apply** *blast*
apply (*erule impE*) **apply** *blast* **apply** (*erule conjE*)+ **apply** (*frule subcon-fob-sub*) **apply** *simp*
apply *force* **apply** (*rule wte-sub*) **apply** *simp* **apply** *simp* **apply** *force*
apply *force*
apply *force*
apply *clarify* **apply** (*erule impE*) **apply** *force* **apply** (*rule conjI*) **apply** (*rule wte-sel*) **apply** *blast*
apply *simp* **apply** *clarify* **apply** (*rule lookup-fob*) **apply** *simp* **apply** *simp*
apply *force*
defer
apply *clarify* **apply** (*erule impE*) **apply** *blast* **apply** (*rule conjI*) **apply** (*rule wte-obj*) **apply** *simp* **apply** *force*
defer
apply *clarify* **apply** (*rule conjI*) **apply** (*rule wt-nil*) **apply** *force*
apply *clarify* **apply** (*rule conjI*) **apply** (*rule wt-cons*) **apply** *simp* **apply** *simp* **apply** *simp* **apply** *blast*
apply *clarify* **apply** (*rule conjI*)
apply (*erule impE*) **apply** *blast* **apply** *clarify*
apply (*erule impE*) **apply** *blast*
apply (*case-tac m*) **apply** (*cases rule*: *wtm.cases*) **apply** *force*
apply *simp* **apply** (*rule wte-upd*) **apply** *simp* **apply** (*rule wt-mtd*)
apply *simp* **apply** *simp* **apply** *simp* **apply** *simp*
apply *clarify* **apply** (*rule conjI*)
apply (*erule impE*) **apply** *blast*
apply (*rule wt-mtd*) **apply** (*frule subcon-fob-sub*) **apply** *force* **apply** *force* **apply** (*rule wte-sub*)

apply *force* **apply** *force* **apply** *simp* **apply** *force*
done

lemma *gradual-soundness-fob*:
$[\![\Gamma \vdash_G e : \tau;\ e \in \textit{fob-term};\ \forall\ x\ \tau.\ \Gamma\ x = \textit{just}\ \tau \longrightarrow \tau \in \textit{fob-type}\ ]\!] \Longrightarrow$
$\Gamma \vdash e : \tau \wedge \tau \in \textit{fob-type}$
using *gradual-soundness-fob-impl* **by** *simp*

lemma *subst-eq*: $\{i \rightarrow \textit{FVar}\ x\}e = \{i \rightarrow \textit{FVar}\ x\}e' \Longrightarrow e = e'$ **sorry**

inductive-cases *wt-mtd-inv*: $\Gamma \vdash (\textit{Method}\ l\ e) : (\textit{Sig}\ l\ \sigma)\ \textit{in}\ \textit{ss}$

lemma *compile-soundness-fob-impl*:
$(\Gamma \vdash e \Rightarrow e' : \tau \longrightarrow$
$e \in \textit{fob-term} \wedge (\forall\ x\ \tau.\ \Gamma\ x = \textit{just}\ \tau \longrightarrow \tau \in \textit{fob-type}) \longrightarrow$
$\Gamma \vdash e : \tau \wedge \tau \in \textit{fob-type} \wedge e = e')$
$\wedge (\Gamma \vdash m \Rightarrow m' : s\ \textit{in}\ \tau \longrightarrow$
$m \in \textit{fob-method} \wedge \tau \in \textit{fob-type} \wedge (\forall\ x\ \tau.\ \Gamma\ x = \textit{just}\ \tau \longrightarrow \tau \in \textit{fob-type}) \longrightarrow$
$\Gamma \vdash m : s\ \textit{in}\ \tau \wedge s \in \textit{fob-sig} \wedge m = m')$
$\wedge (\Gamma \vdash ms \Rightarrow ms' :: ss\ \textit{in}\ \tau \longrightarrow$
$ms \in \textit{fob-methods} \wedge \tau \in \textit{fob-type} \wedge (\forall\ x\ \tau.\ \Gamma\ x = \textit{just}\ \tau \longrightarrow \tau \in \textit{fob-type}) \longrightarrow$
$\Gamma \vdash ms :: ss\ \textit{in}\ \tau \wedge ss \in \textit{fob-sigs} \wedge ms = ms')$
(**is** $(\Gamma \vdash e \Rightarrow e' : \tau \longrightarrow ?P\ \Gamma\ e\ e'\ \tau) \wedge (\Gamma \vdash m \Rightarrow m' : s\ \textit{in}\ \tau \longrightarrow ?PM\ \Gamma\ m\ m'\ s\ \tau) \wedge (\Gamma \vdash ms \Rightarrow ms' :: ss\ \textit{in}\ \tau \longrightarrow ?PS\ \Gamma\ ms\ ms'\ ss\ \tau)$)
apply (*induct rule*: *compile-cm-cms.induct*)
using *wte-var* **apply** *blast*
apply *clarify* **apply** (*rule conjI*) **apply** (*rule wte-const*)
apply (*case-tac c*) **apply** *force* **apply** *force* **apply** *force* **apply** *force*
apply *force*
apply *clarify*
apply (*rule conjI*)
apply (*rule wte-abs*) **apply** *simp*
apply *clarify* **apply** (*erule-tac x=x* **in** *allE*) **apply** (*erule impE*)
apply *simp* **apply** (*erule conjE*)+
apply (*erule impE*) **apply** (*rule conjI*)
apply (*rule fob-subst*) **apply** *simp*
apply (*rule allI*)+ **apply** (*rule impI*)
apply (*erule-tac x=xa* **in** *allE*) **apply** (*erule-tac* $x{=}\tau'$ **in** *allE*)
apply (*case-tac x = xa*) **apply** *simp* **apply** *simp*
apply (*rule conjI*) **apply** (*erule conjE*) **apply** *assumption* **apply** *assumption*
apply (*erule-tac x=Suc* (*setmax L*) **in** *allE*)
apply (*erule impE*) **apply** (*rule max-is-fresh*) **apply** *simp*
apply (*erule conjE*)+ **apply** (*erule impE*) **apply** (*rule conjI*) **apply** (*rule fob-subst*) **apply** *simp*
apply (*rule allI*)+ **apply** (*rule impI*) **apply** (*case-tac x = Suc* (*setmax L*)) **apply** *simp* **apply** *simp*
apply (*rule conjI*) **apply** *blast*
apply (*erule conjE*)+ **apply** (*frule subst-eq*) **apply** *simp*
apply *force*

apply *clarify* **apply** (*rule conjI*) **apply** (*rule wte-app*) **apply** *force* **apply** (*erule impE*) **apply** *blast*
apply (*erule impE*) **apply** *blast* **apply** (*erule conjE*)+ **apply** (*frule subcon-fob-sub*) **apply** *simp*
apply *force* **apply** (*rule wte-sub*) **apply** *simp* **apply** *simp* **apply** (*rule conjI*) **apply** *force*
apply (*erule impE*) **apply** *blast* **apply** (*erule impE*) **apply** *blast* **apply** (*erule conjE*)+
apply *simp*
apply (*simp add*: *mcast-def*) **apply** *clarify*
using *merge-fob* **apply** *force*
apply *force*
apply *force*
apply *clarify* **apply** (*erule impE*) **apply** *force* **apply** (*rule conjI*) **apply** (*rule wte-sel*) **apply** *blast*
apply *simp* **apply** *clarify* **apply** (*rule conjI*) **apply** (*rule lookup-fob*) **apply** *simp* **apply** *simp* **apply** *simp*
apply *force*
defer
apply *clarify* **apply** (*erule impE*) **apply** *blast* **apply** (*rule conjI*) **apply** (*rule wte-obj*) **apply** *simp* **apply** *force*
apply (*rule conjI*) **apply** *force* **apply** *simp*
defer
apply *clarify* **apply** (*rule conjI*) **apply** (*rule wt-nil*) **apply** *force*
apply *clarify* **apply** (*rule conjI*) **apply** (*rule wt-cons*) **apply** *blast* **apply** *blast* **apply** (*rule conjI*) **apply** *blast*
apply *simp*
apply *clarify* **apply** (*erule impE*) **apply** *blast* **apply** *clarify*
apply (*erule impE*) **apply** *blast* **apply** (*rule conjI*)
apply (*rule wte-upd*) **apply** *simp* **apply** *simp* **apply** (*rule wt-mtd*)
apply *force* **apply** *simp* **apply** *simp* **apply** (*rule conjI*) **apply** *force*
apply (*simp add*: *mcast-def*) **apply** *clarify* **using** *merge-fob* **apply** *force*
apply *clarify* **apply** (*erule impE*) **apply** *blast*
apply (*rule conjI*) **apply** (*rule wt-mtd*) **apply** (*frule subcon-fob-sub*) **apply** *force* **apply** *force*
apply (*rule wte-sub*) **apply** *force* **apply** *force* **apply** *simp* **apply** (*rule conjI*) **apply** *force*
apply (*simp add*: *mcast-def*) **apply** *clarify*
proof −
fix Γ σ τ *e* e' *l* *ss*
assume *ss*: $ss \in$ *fob-sigs* **and** *s*: $\sigma \in$ *fob-type* **and** *sss*: $\sigma \neq \sigma \leftharpoonup$ *ObjT ss*
from *ss* **have** *ObjT ss* $\in$ *fob-type* **by** *auto*
with *sss s merge-fob* **have** *False* **by** *auto*
thus $e' =$ *Cast* e' $(\sigma \rightarrow \tau)$ $((\sigma \leftharpoonup \textit{ObjT ss}) \rightarrow \tau)$ **by** *simp*
qed

lemma *compile-soundness-fob*:
$[\![\, \Gamma \vdash e \Rightarrow e' : \tau;\ e \in \textit{fob-term};\ \forall\ x\ \tau.\ \Gamma\ x = \textit{just}\ \tau \longrightarrow \tau \in \textit{fob-type}\,]\!] \Longrightarrow$
$\Gamma \vdash e : \tau \wedge \tau \in \textit{fob-type} \wedge e = e'$
using *compile-soundness-fob-impl* **by** *simp*

8 The Substitution Lemma

8.1 Lemmas About Substitution

lemma *bsubst-cross-all*:
$(\forall\ i\ j\ u\ v.\ i \neq j \wedge\ \{i{\rightarrow}u\}(\{j{\rightarrow}v\}e) = \{j{\rightarrow}v\}e \longrightarrow \{i{\rightarrow}u\}e = e)$
$\wedge\ (\forall\ i\ j\ u\ v.\ i \neq j \wedge\ \{i{:}{\rightarrow}u\}(\{j{:}{\rightarrow}v\}m) = \{j{:}{\rightarrow}v\}m \longrightarrow$
$\{i{:}{\rightarrow}u\}\ m = m)$
$\wedge\ (\forall\ i\ j\ u\ v.\ i \neq j \wedge\ \{i{::}{\rightarrow}u\}(\{j{::}{\rightarrow}v\}ms) = \{j{::}{\rightarrow}v\}ms \longrightarrow$
$\{i{::}{\rightarrow}u\}ms = ms)$
apply (*induct rule*: *expr-method.induct*)
apply *force*
apply *force*
apply *force*
apply *clarify*
apply (*erule-tac x=Suc i* **in** *allE*)
apply (*erule-tac x=Suc j* **in** *allE*)
apply (*erule-tac x=u* **in** *allE*)
apply (*erule-tac x=v* **in** *allE*)
apply *simp*
apply *clarify*
apply (*erule-tac x=i* **in** *allE*)
apply (*erule-tac x=i* **in** *allE*)
apply (*erule-tac x=j* **in** *allE*)
apply (*erule-tac x=j* **in** *allE*)
apply *simp* **apply** *blast*
apply *clarify*
apply (*erule-tac x=i* **in** *allE*)
apply (*erule-tac x=j* **in** *allE*)
apply *simp* **apply** *blast*
apply *clarify*
apply (*erule-tac x=i* **in** *allE*)
apply (*erule-tac x=j* **in** *allE*)
apply (*erule-tac x=u* **in** *allE*)
apply (*erule-tac x=v* **in** *allE*)
apply *simp*
apply *clarify*
apply (*erule-tac x=i* **in** *allE*)
apply (*erule-tac x=j* **in** *allE*)
apply (*erule-tac x=u* **in** *allE*)
apply (*erule-tac x=v* **in** *allE*)
apply *simp*
apply *clarify*
apply (*erule-tac x=i* **in** *allE*)
apply (*erule-tac x=i* **in** *allE*)
apply (*erule-tac x=j* **in** *allE*)
apply (*erule-tac x=j* **in** *allE*)
apply (*erule-tac x=u* **in** *allE*)
apply (*erule-tac x=u* **in** *allE*)
apply (*erule-tac x=v* **in** *allE*)

apply (*erule-tac x=v* **in** *allE*)
apply *simp*
apply *clarify*
apply (*erule-tac x=i* **in** *allE*)
apply (*erule-tac x=j* **in** *allE*)
apply (*erule-tac x=u* **in** *allE*)
apply (*erule-tac x=v* **in** *allE*)
apply *simp*
apply *clarify* **apply** *simp*
apply *clarify*
apply (*erule-tac x=i* **in** *allE*)
apply (*erule-tac x=i* **in** *allE*)
apply (*erule-tac x=j* **in** *allE*)
apply (*erule-tac x=j* **in** *allE*)
apply (*erule-tac x=u* **in** *allE*)
apply (*erule-tac x=u* **in** *allE*)
apply (*erule-tac x=v* **in** *allE*)
apply (*erule-tac x=v* **in** *allE*)
apply *simp*
done

lemma *bsubst-cross*[*rule-format*]:
$(\forall\ i\ j\ u\ v.\ i \neq j \wedge\ \{i{\rightarrow}u\}(\{j{\rightarrow}v\}e) = \{j{\rightarrow}v\}e \longrightarrow \{i{\rightarrow}u\}e = e)$
using *bsubst-cross-all* **apply** *blast* **done**

lemma *finite-env-update*: *finite-env* $\Gamma \Longrightarrow$ *finite-env* $(\Gamma(x{\mapsto}\tau))$
by (*simp add*: *finite-env-def*)

lemma *bsubst-wt-all*:
$(\Gamma \vdash e : \tau \longrightarrow$ *finite-env* $\Gamma \longrightarrow (\forall\ k\ e'.\ \{k{\rightarrow}e'\}e = e))$
$\wedge\ (\Gamma \vdash m : \sigma$ *in* $A \longrightarrow$ *finite-env* $\Gamma \longrightarrow (\forall\ k\ e'.$ *bsubstm* $k\ e'\ m = m))$
$\wedge\ (\Gamma \vdash ms :: msigs$ *in* $A' \longrightarrow$ *finite-env* $\Gamma \longrightarrow (\forall\ k\ e'.$ *bsubsts* $k\ e'\ ms = ms))$
apply (*induct rule*: *wte-wtm-wtms.induct*)
apply *force*
apply *force*
apply *clarify* **apply** (*simp del*: *fun-upd-apply*)
apply (*erule-tac x=Suc* (*setmax L*) **in** *allE*)
apply (*erule impE*)
apply (*rule max-is-fresh*) **apply** *simp*
apply (*erule conjE*)+
apply (*erule impE*) **apply** (*rule finite-env-update*) **apply** *assumption*
apply (*erule-tac x=Suc k* **in** *allE*)
apply (*erule-tac x=e′* **in** *allE*)
apply (*rule bsubst-cross*) **apply** *blast*
apply *force*
apply *force*
apply *force*
apply *force*
apply *clarify*
apply (*erule-tac x=k* **in** *allE*)

apply (*erule-tac x=k* **in** *allE*)
apply (*erule-tac x=e′* **in** *allE*)
apply (*erule-tac x=e′* **in** *allE*)
apply *simp*
apply *clarify*
apply (*erule-tac x=k* **in** *allE*)
apply (*erule-tac x=e′* **in** *allE*)
apply *simp*
apply *clarify*
apply (*erule-tac x=k* **in** *allE*)
apply (*erule-tac x=e′* **in** *allE*)
apply *simp*
apply *force*
apply *force*
done

lemma *bsubst-wt*:
$[\![\Gamma \vdash e : \tau;\ \textit{finite-env}\ \Gamma]\!] \Longrightarrow \{k \rightarrow e'\}e = e$
using *bsubst-wt-all* **by** *blast*

lemma *subst-permute-impl*[*rule-format*]:
$(\forall\ j\ x\ z\ \Gamma\ \tau\ e'.\ x \neq z \wedge \Gamma \vdash e' : \tau \wedge \textit{finite-env}\ \Gamma$
$\longrightarrow [z \rightarrow e'](\{j \rightarrow FVar\ x\}e) = \{j \rightarrow\ FVar\ x\}([z \rightarrow e']e))$
$\wedge (\forall\ j\ x\ z\ \Gamma\ \tau\ e'.\ x \neq z \wedge \Gamma \vdash e' : \tau \wedge \textit{finite-env}\ \Gamma$
$\longrightarrow [z{:}\rightarrow e'](\{j{:}\rightarrow FVar\ x\}m) = \{j{:}\rightarrow\ FVar\ x\}([z{:}\rightarrow e']m))$
$\wedge (\forall\ j\ x\ z\ \Gamma\ \tau\ e'.\ x \neq z \wedge \Gamma \vdash e' : \tau \wedge \textit{finite-env}\ \Gamma$
$\longrightarrow [z{::}\rightarrow e'](\{j{::}\rightarrow FVar\ x\}ms) = \{j{::}\rightarrow\ FVar\ x\}([z{::}\rightarrow e']ms))$
apply (*induct rule*: *expr-method.induct*)
apply *force*
apply *simp* **apply** *clarify*
using *bsubst-wt* **apply** *force*
apply *simp*
apply *simp* **apply** *clarify* **apply** *blast*
apply *simp* **apply** *clarify*
apply (*erule-tac x=j* **in** *allE*)
apply (*erule-tac x=j* **in** *allE*)
apply (*erule-tac x=x* **in** *allE*)
apply (*erule-tac x=x* **in** *allE*)
apply (*erule-tac x=z* **in** *allE*)
apply (*erule-tac x=z* **in** *allE*)
apply (*erule-tac x=Γ* **in** *allE*)
apply (*erule-tac x=Γ* **in** *allE*)
apply *blast*
apply *simp*
apply *simp*
apply *simp*
apply *simp* **apply** *clarify*
apply (*erule-tac x=j* **in** *allE*)
apply (*erule-tac x=j* **in** *allE*)
apply (*erule-tac x=x* **in** *allE*)

apply (*erule-tac* $x=x$ **in** *allE*)
apply (*erule-tac* $x=z$ **in** *allE*)
apply (*erule-tac* $x=z$ **in** *allE*)
apply (*erule-tac* $x=\Gamma$ **in** *allE*)
apply (*erule-tac* $x=\Gamma$ **in** *allE*)
apply *blast*
apply *simp*
apply *simp*
apply *simp* **apply** *clarify*
apply (*erule-tac* $x=j$ **in** *allE*)
apply (*erule-tac* $x=j$ **in** *allE*)
apply (*erule-tac* $x=x$ **in** *allE*)
apply (*erule-tac* $x=x$ **in** *allE*)
apply (*erule-tac* $x=z$ **in** *allE*)
apply (*erule-tac* $x=z$ **in** *allE*)
apply (*erule-tac* $x=\Gamma$ **in** *allE*)
apply (*erule-tac* $x=\Gamma$ **in** *allE*)
apply *blast*
done

lemma *subst-permute*:
$[\![x \neq z; \Gamma \vdash e' : \tau; \textit{finite-env}\ \Gamma]\!]$
$\Longrightarrow \{j \rightarrow FVar\ x\}([z \rightarrow e']e) = [z \rightarrow e']([j \rightarrow FVar\ x\}e)$
using *subst-permute-impl*[*of e Method l e* []] **apply** *simp*
apply (*erule-tac* $x=j$ **in** *allE*)
apply (*erule-tac* $x=x$ **in** *allE*)
apply (*erule-tac* $x=z$ **in** *allE*)
apply (*erule-tac* $x=\Gamma$ **in** *allE*)
apply (*erule-tac* $x=\tau$ **in** *allE*)
apply (*erule-tac* $x=e'$ **in** *allE*)
apply *force*
done

lemma *decompose-subst-impl*:
$(\forall\ u\ x\ i.\ x \notin FV\ e \longrightarrow \{i \rightarrow u\}e = [x \rightarrow u](\{i \rightarrow FVar\ x\}e))$
$\wedge (\forall\ u\ x\ i.\ x \notin FVm\ m \longrightarrow \{i{:}\rightarrow u\}m = [x{:}\rightarrow u](\{i{:}\rightarrow FVar\ x\}m))$
$\wedge (\forall\ u\ x\ i.\ x \notin FVs\ ms \longrightarrow \{i{::}\rightarrow u\}ms = [x{::}\rightarrow u](\{i{::}\rightarrow FVar\ x\}ms))$
apply (*induct rule*: *expr-method.induct*)
apply *force*
apply *force*
apply *force*
apply *clarify*
apply (*erule-tac* $x=u$ **in** *allE*)
apply (*erule-tac* $x=x$ **in** *allE*)
apply (*erule-tac* $x=Suc\ i$ **in** *allE*)
apply *simp*
apply *force*
apply *clarify*
apply (*erule-tac* $x=u$ **in** *allE*)
apply (*erule-tac* $x=x$ **in** *allE*)

apply (*erule-tac x=i* **in** *allE*)
apply *simp*
apply *clarify*
apply (*erule-tac x=u* **in** *allE*)
apply (*erule-tac x=x* **in** *allE*)
apply (*erule-tac x=i* **in** *allE*)
apply *simp*
apply *clarify*
apply (*erule-tac x=u* **in** *allE*)
apply (*erule-tac x=x* **in** *allE*)
apply (*erule-tac x=i* **in** *allE*)
apply *simp*
apply *simp*
apply *clarify*
apply (*erule-tac x=u* **in** *allE*)
apply (*erule-tac x=x* **in** *allE*)
apply (*erule-tac x=i* **in** *allE*)
apply *simp*
apply *simp*
apply *clarify*
apply (*erule-tac x=u* **in** *allE*)
apply (*erule-tac x=u* **in** *allE*)
apply (*erule-tac x=x* **in** *allE*)
apply (*erule-tac x=x* **in** *allE*)
apply (*erule-tac x=i* **in** *allE*)
apply (*erule-tac x=i* **in** *allE*)
apply *simp*
done

lemma *decompose-subst*[*rule-format*]:
$(\forall\ u\ x\ i.\ x \notin FV\ e \longrightarrow \{i{\rightarrow}u\}e = [x{\rightarrow}u](\{i{\rightarrow}FVar\ x\}e))$
using *decompose-subst-impl* **by** *blast*

8.2 Lammas About Environments

constdefs *subseteq* :: *env* ⇒ *env* ⇒ *bool* (**infixl** ⊆ *80*)
$\Gamma \subseteq \Gamma' \equiv \forall\ x\ \tau.\ \Gamma\ x = Some\ \tau \longrightarrow \Gamma'\ x = Some\ \tau$

lemma *env-weakening-impl*:
$(\Gamma \vdash e : \tau \longrightarrow (\forall\ \Gamma'.\ \Gamma \subseteq \Gamma' \wedge finite\text{-}env\ \Gamma' \longrightarrow \Gamma' \vdash e : \tau))$
$\wedge\ (\Gamma \vdash m : \sigma\ in\ A \longrightarrow (\forall\ \Gamma'.\ \Gamma \subseteq \Gamma' \wedge finite\text{-}env\ \Gamma' \longrightarrow \Gamma' \vdash m : \sigma\ in\ A))$
$\wedge\ (\Gamma \vdash ms :: \sigma s\ in\ A' \longrightarrow (\forall\ \Gamma'.\ \Gamma \subseteq \Gamma' \wedge finite\text{-}env\ \Gamma' \longrightarrow \Gamma' \vdash ms :: \sigma s\ in\ A'))$
apply (*induct rule*: *wte-wtm-wtms.induct*)
using *subseteq-def wte-var* **apply** *blast*
using *wte-const* **apply** *blast*
defer
using *wte-app* **apply** *blast*
using *wte-sub* **apply** *blast*
using *wte-cast* **apply** *blast*
using *wte-sel* **apply** *blast*

using *wte-upd* **apply** *blast*
using *wte-obj* **apply** *blast*
using *wt-mtd* **apply** *blast*
using *wt-nil* **apply** *blast*
using *wt-cons* **apply** *blast*
apply (*rule allI*) **apply** (*rule impI*)
proof −
fix $L\ \Gamma\ \sigma\ \tau\ e\ \Gamma'$
assume *fL*: *finite L*
and *IH*: $\forall x.\ x \notin L \longrightarrow$
$(\Gamma(x \mapsto \sigma) \vdash \{0 \rightarrow FVar\ x\}e : \tau \wedge$
$(\forall \Gamma'.\ \Gamma(x \mapsto \sigma) \subseteq \Gamma' \wedge finite\text{-}env\ \Gamma' \longrightarrow \Gamma' \vdash \{0 \rightarrow FVar\ x\}e : \tau)) \wedge x \notin dom\ \Gamma$
and *GGP*: $\Gamma \subseteq \Gamma' \wedge finite\text{-}env\ \Gamma'$
let $?L = L \cup dom\ \Gamma'$
from *GGP* **have** *finite* $(dom\ \Gamma')$ **using** *finite-env-def* **by** *auto*
with *fL* **have** *fL2*: *finite ?L* **by** *auto*
{ fix *x* **assume** *xL*: $x \notin ?L$
from *GGP* **have** *xGxGP*: $\Gamma(x \mapsto \sigma) \subseteq \Gamma'(x \mapsto \sigma)$ **using** *subseteq-def* **by** *auto*
from *GGP* **have** *fGP*: *finite-env* $(\Gamma'(x \mapsto \sigma))$ **using** *finite-env-def* **by** *auto*
from *xL fGP IH xGxGP* **have** $\Gamma'(x \mapsto \sigma) \vdash \{0 \rightarrow FVar\ x\}e : \tau \wedge x \notin dom\ \Gamma'$ **by**
blast
} hence *X*: $\forall\ x.\ x \notin ?L \longrightarrow \Gamma'(x \mapsto \sigma) \vdash \{0 \rightarrow FVar\ x\}e : \tau \wedge x \notin dom\ \Gamma'$ **by** *blast*
from *fL2 X* **show** $\Gamma' \vdash (\lambda{:}\sigma.\ e) : \sigma \rightarrow \tau$ **by** (*rule wte-abs*)
qed

lemma *env-weakening*:
$[\![\ \Gamma \vdash e : \tau;\ \Gamma \subseteq \Gamma';\ finite\text{-}env\ \Gamma'\ ]\!] \Longrightarrow \Gamma' \vdash e : \tau$
using *env-weakening-impl* **by** *blast*

8.3 Main Lemma

lemma *substitution-impl*:
$(\Gamma \vdash e1 : \tau \longrightarrow \Gamma\ x = Some\ \sigma \wedge finite\text{-}env\ \Gamma \longrightarrow$
$(\forall\ \Gamma'.\ finite\text{-}env\ \Gamma' \wedge \Gamma - x \subset \Gamma' \wedge \Gamma' \vdash e2 : \sigma \longrightarrow$
$\Gamma' \vdash [x \rightarrow e2]e1 : \tau))$
$\wedge (\Gamma \vdash m : sig\ in\ A \longrightarrow \Gamma\ x = Some\ \sigma \wedge finite\text{-}env\ \Gamma \longrightarrow$
$(\forall\ \Gamma'.\ finite\text{-}env\ \Gamma' \wedge \Gamma - x \subset \Gamma' \wedge \Gamma' \vdash e2 : \sigma \longrightarrow$
$\Gamma' \vdash [x{:}\rightarrow e2]m : sig\ in\ A))$
$\wedge (\Gamma \vdash ms :: sigs\ in\ A' \longrightarrow \Gamma\ x = Some\ \sigma \wedge finite\text{-}env\ \Gamma \longrightarrow$
$(\forall\ \Gamma'.\ finite\text{-}env\ \Gamma' \wedge \Gamma - x \subset \Gamma' \wedge \Gamma' \vdash e2 : \sigma \longrightarrow$
$\Gamma' \vdash [x{::}\rightarrow e2]ms :: sigs\ in\ A'))$
apply (*induct rule* : *wte-wtm-wtms.induct*)
apply (*case-tac x* = *xa*) **apply** *simp*
apply *clarify* **apply** (*simp only*: *remove-bind-def*)
apply (*erule-tac x=xa* **in** *allE*) **apply** *simp* **apply** (*rule wte-var*) **apply** *assumption*
using *wte-const* **apply** *force*
defer
apply *clarify* **apply** *simp* **apply** (*rule wte-app*) **apply** *blast* **apply** *blast*
apply *clarify* **apply** *simp* **apply** (*rule wte-sub*) **apply** *blast* **apply** *blast*
apply *clarify* **apply** *simp* **apply** (*rule wte-cast*) **apply** *blast* **apply** *blast* **apply**

blast
apply *clarify* **apply** *simp* **apply** (*rule wte-sel*) **apply** *blast* **apply** *blast*
apply *clarify* **apply** *simp* **apply** (*rule wte-upd*) **apply** *blast* **apply** *blast*
apply *blast*
apply *clarify* **apply** *simp* **apply** (*rule wte-obj*) **apply** *blast*
apply *clarify* **apply** *simp* **apply** (*rule wt-mtd*) **apply** *blast* **apply** *blast*
apply *clarify* **apply** *simp* **apply** (*rule wt-nil*)
apply *clarify* **apply** *simp* **apply** (*rule wt-cons*) **apply** *blast* **apply** *blast*
proof *clarify*
fix L::*nat set* **and** Γ::*env* **and** σ'::*ty* **and** τ e Γ'
assume *fL*: *finite* L
and *IH*: $\forall xa.\ xa \notin L \longrightarrow$
$(\Gamma(xa \mapsto \sigma') \vdash \{0 \to FVar\ xa\}e : \tau\ \wedge$
$((\Gamma(xa \mapsto \sigma'))\ x = Some\ \sigma \wedge finite\text{-}env\ (\Gamma(xa \mapsto \sigma')) \longrightarrow$
$(\forall \Gamma'.\ finite\text{-}env\ \Gamma' \wedge \Gamma(xa \mapsto \sigma') - x \subset \Gamma' \wedge \Gamma' \vdash e2 : \sigma \longrightarrow$
$\Gamma' \vdash [x \to e2](\{0 \to FVar\ xa\}e) : \tau))) \wedge$
$xa \notin dom\ \Gamma$
and *xG*: $\Gamma\ x = Some\ \sigma$ **and** *fG*: *finite-env* Γ
and *fGP*: *finite-env* Γ' **and** *GxG*: $\Gamma - x \subset \Gamma'$ **and** *wte2*: $\Gamma' \vdash e2 : \sigma$
let $?L = insert\ x\ (L \cup dom\ \Gamma \cup dom\ \Gamma')$
from *fL fG fGP* **have** *fL2*: *finite ?L* **using** *finite-env-def* **by** *auto*
show $\Gamma' \vdash [x \to e2](\lambda{:}\sigma'.\ e) : \sigma' \to \tau$
apply *simp* **apply** (*rule wte-abs*[*of ?L*])
using *fL2* **apply** *simp* **apply** (*rule allI*) **apply** (*rule impI*) **apply** (*rule conjI*)
defer apply *simp*
proof $-$
fix x' **assume** *xL*: $x' \notin ?L$
let $?G = \Gamma(x' \mapsto \sigma')$
let $?GP = \Gamma'(x' \mapsto \sigma')$
note *xL*
moreover from *xG xL* **have** $?G\ x = Some\ \sigma$ **by** *auto*
moreover from *fG* **have** *finite-env ?G* **using** *finite-env-def* **by** *auto*
moreover from *fGP* **have** *fGP2*: *finite-env ?GP* **using** *finite-env-def* **by** *auto*
moreover from *GxG* **have** $?G - x \subset ?GP$ **using** *remove-bind-def* **by** *auto*
moreover have *wte2*: $?GP \vdash e2 : \sigma$
proof $-$
from *xL* **have** *GPGP*: $\Gamma' \subseteq ?GP$ **using** *subseteq-def* **by** *auto*
from *wte2 GPGP fGP2* **show** *?thesis* **using** *env-weakening* **by** *blast*
qed
moreover note *IH*
ultimately have *wte*: $?GP \vdash [x \to e2](\{0 \to FVar\ x'\}e) : \tau$ **by** *blast*

from *xL* **have** *xpx*: $x' \neq x$ **by** *auto*
from *xpx wte2 fGP2* **have** $\{0 \to FVar\ x'\}([x \to e2]e) = [x \to e2](\{0 \to FVar\ x'\}e)$
by (*rule subst-permute*)

with *wte* **have** *wteb*: $?GP \vdash \{0 \to FVar\ x'\}([x \to e2]e) : \tau$ **by** *simp*
from *xL* **have** *xGP*: $x' \notin dom\ \Gamma'$ **by** *auto*
from *wteb xGP* **show** $?GP \vdash \{0 \to FVar\ x'\}([x \to e2]e) : \tau$
by *blast*

qed
qed

lemma *substitution*:
$[\![\ \Gamma \vdash e_1 : \tau;\ \Gamma\ x = Some\ \sigma;\ finite\text{-}env\ \Gamma;\ finite\text{-}env\ \Gamma';\ \Gamma - x \subset \Gamma';\ \Gamma' \vdash e_2 : \sigma\ ]\!]$
$\Longrightarrow\ \Gamma' \vdash [x \to e_2]e_1 : \tau$
using *substitution-impl* **apply** *blast* **done**

9 Type Safety

9.1 Canonical Forms

lemma *canonical-form-simple-dyn-impl*[*rule-format*]:
$(\Gamma \vdash v : \tau \longrightarrow \Gamma = empty \wedge \tau = ? \wedge SimpleValues\ v \longrightarrow False)$
$\wedge\ (\Gamma \vdash m : \sigma\ in\ A \longrightarrow True)$
$\wedge\ (\Gamma \vdash ms :: \sigma s\ in\ A' \longrightarrow True)$
apply (*induct rule*: *wte-wtm-wtms.induct*)
apply *force* **apply** (*case-tac c*) **apply** *force* **apply** *force* **apply** *force* **apply** *force* **apply** *force*
apply *force* **apply** *force*
apply (*cases rule*: *subtype.cases*) **apply** *force* **apply** *force* **apply** *force* **apply** *force*
apply *force* **apply** *force* **apply** *force* **apply** *force*
apply *force* **apply** *force* **apply** *force* **apply** *force* **apply** *force* **apply** *force* **apply** *force*
done

lemma *canonical-form-simple-dyn*:
$[\![\ empty \vdash v : ?;\ SimpleValues\ v\ ]\!] \Longrightarrow False$
using *canonical-form-simple-dyn-impl* **by** *blast*

lemma *canonical-form-int-impl*:
$(\Gamma \vdash e : \tau \longrightarrow \tau = IntT \wedge Values\ e \wedge \Gamma = empty \longrightarrow (\exists\ n.\ e = Const\ (IntC\ n)))$
$\wedge\ (\Gamma \vdash m : \sigma\ in\ A \longrightarrow True)$
$\wedge\ (\Gamma \vdash ms :: \sigma s\ in\ A' \longrightarrow True)$
apply (*induct rule*: *wte-wtm-wtms.induct*)
apply *force*
apply (*case-tac c*) **apply** *force* **apply** *force* **apply** *force* **apply** *force* **apply** *force*
apply *force* **apply** *force*
apply *clarify* **apply** (*cases rule*: *subtype.cases*) **apply** *force* **apply** *force* **apply** *force*
apply *force* **apply** *force* **apply** *force* **apply** *force* **apply** *force*
apply *simp* **apply** *clarify*
apply (*rule canonical-form-simple-dyn*) **apply** *simp* **apply** *auto*
apply (*cases rule*: *consistent.cases*) **apply** *auto*
done

lemma *canonical-form-int*:
$[\![\ empty \vdash e : IntT;\ Values\ e\ ]\!] \Longrightarrow \exists\ n.\ e = Const\ (IntC\ n)$
using *canonical-form-int-impl* **by** *simp*

lemma *simple-implies-value*[*simp*]: *SimpleValues v* $\Longrightarrow$ *Values v*
apply (*cases v*) **by** *auto*

lemma *canonical-form-simple-fun-impl*:
$(\Gamma \vdash v : st \longrightarrow \Gamma = empty \wedge SimpleValues\ v \longrightarrow$
$(\forall\ \sigma\ \tau.\ st = \sigma \to \tau \longrightarrow$
$(\exists\ \sigma'\ e.\ v = \lambda{:}\sigma'.\ e) \vee (\exists\ c.\ v = Const\ c)))$
$\wedge\ (\Gamma \vdash m : s\ in\ A \longrightarrow True)$
$\wedge\ (\Gamma \vdash ms :: ss\ in\ A' \longrightarrow True)$
apply (*induct rule*: *wte-wtm-wtms.induct*)
apply *force* **apply** *force* **apply** *force* **apply** *force*
apply *clarify* **apply** *simp*
apply (*frule sub-fun-right-inv*) **apply** (*erule exE*)+
apply *simp*
apply *simp* **apply** *simp* **apply** *simp* **apply** *simp* **apply** *simp* **apply** *simp* **apply** *simp*
done

lemma *canonical-form-simple-fun*:
$[\![\ empty \vdash v : \sigma \to \tau;\ SimpleValues\ v\ ]\!] \Longrightarrow$
$(\exists\ \sigma'\ e.\ v = \lambda{:}\sigma'.\ e) \vee (\exists\ c.\ v = Const\ c)$
using *canonical-form-simple-fun-impl* **by** *blast*

lemma *canonical-form-fun-impl*:
$(\Gamma \vdash v : st \longrightarrow (\forall\ \sigma\ \tau.\ st = (\sigma \to \tau) \wedge \Gamma = empty \wedge\ Values\ v \longrightarrow$
$(\exists\ \sigma'\ e.\ v = \lambda{:}\sigma'.\ e) \vee (\exists\ c.\ v = Const\ c)$
$\vee\ (\exists\ \sigma'\ \tau'\ v'\ \varrho\ \nu.\ v = v'\langle\sigma'{\to}\tau' \Rightarrow \varrho{\to}\nu\rangle)))$
$\wedge\ (\Gamma \vdash m : s\ in\ A \longrightarrow True)$
$\wedge\ (\Gamma \vdash ms :: ss\ in\ A' \longrightarrow True)$
apply (*induct rule*: *wte-wtm-wtms.induct*)
apply *force* **apply** *force* **apply** *force* **apply** *force*
apply *clarify* **apply** (*frule sub-fun-right-inv*) **apply** (*erule exE*)+
apply *simp*
apply (*rule allI*)+ **apply** (*rule impI*) **apply** (*erule conjE*)
apply (*cases rule*: *consistent.cases*)
apply *simp* **apply** *simp* **apply** *simp* **apply** *simp*
defer apply *simp*+
using *canonical-form-simple-dyn* **apply** *blast*
done

lemma *canonical-form-fun*:
assumes *wtf*: $empty \vdash v : \sigma \to \tau$
and *v*: *Values v*
shows $(\exists\ \sigma'\ e.\ v = \lambda{:}\sigma'.\ e) \vee (\exists\ c.\ v = Const\ c)$
$\vee\ (\exists\ \sigma'\ \tau'\ v'\ \varrho\ \nu.\ v = v'\langle\sigma'{\to}\tau' \Rightarrow \varrho{\to}\nu\rangle)$
using *wtf v canonical-form-fun-impl* **by** *simp*

lemma *canonical-form-obj-impl*:
$(\Gamma \vdash v : ot \longrightarrow (\forall\ ss.\ ot = ObjT\ ss \wedge \Gamma = empty \wedge\ Values\ v \longrightarrow$

$(\exists\ ms\ \tau.\ v = Obj\ ms\ \tau) \vee (\exists\ ms\ \tau\ rr\ tt.\ v = (Obj\ ms\ \tau)\langle ObjT\ rr \Rightarrow ObjT\ tt\rangle)))$
$\wedge\ (\Gamma \vdash m : s\ in\ A \longrightarrow True)$
$\wedge\ (\Gamma \vdash ms :: ss\ in\ A' \longrightarrow True)$
apply (*induct rule*: *wte-wtm-wtms.induct*)
apply *force*
apply (*case-tac c*) **apply** *simp* **apply** *simp* **apply** *simp* **apply** *simp* **apply** *simp*
apply *simp* **apply** *simp*
apply *clarify* **apply** (*frule sub-obj-right-inv*) **apply** (*erule exE*)+
apply *simp*
defer
apply *simp* **apply** *simp* **apply** *simp* **apply** *simp* **apply** *simp* **apply** *simp*
apply (*rule allI*) **apply** (*rule impI*) **apply** (*erule conjE*)
apply (*cases rule*: *consistent.cases*) **apply** *simp*+
using *canonical-form-simple-dyn* **apply** *blast*
apply *auto*
done

lemma *canonical-form-obj*:
$[\![\ empty \vdash v :\ ObjT\ ss;\ Values\ v\ ]\!]$
$\Longrightarrow (\exists\ ms\ \tau.\ v = Obj\ ms\ \tau) \vee (\exists\ ms\ \tau\ rr\ tt.\ v = (Obj\ ms\ \tau)\langle ObjT\ rr \Rightarrow ObjT\ tt\rangle)$
using *canonical-form-obj-impl* **by** *blast*

9.2 Delta Typability

lemma *delta-typability*:
assumes *tc*: $TypeOf\ c = \tau' \rightarrow \tau$
and *vt*: $empty \vdash v : \tau'$ **and** *vv*: *Values v*
shows $\exists\ v'.\ \delta\ c\ v = Some\ v' \wedge empty \vdash v' : \tau$
using *tc vt vv* **apply** (*cases c*) **apply** *simp* **apply** *simp* **apply** *simp*
proof −
assume *tc*: $TypeOf\ c = \tau' \rightarrow \tau$ **and** *vt*: $empty \vdash v : \tau'$
and *vv*: *Values v* **and** *c*: $c = Succ$
from *c tc* **have** *st*: $\tau' = IntT \wedge \tau = IntT$ **by** *simp*
from *st vt vv* **obtain** *n* **where** *v*: $v = Const\ (IntC\ n)$
apply *simp* **using** *canonical-form-int* **by** *blast*
let $?VP = Const\ (IntC\ (n + 1))$
have *wtvp*: $empty \vdash ?VP : IntT$
using *wte-const*[*of empty IntC* $(n + 1)$] **by** *auto*
from *c v* **have** *d*: $\delta\ c\ v = Some\ ?VP$ **by** *simp*
from *d wtvp st* **show** *?thesis* **by** *simp*
next
assume *tc*: $TypeOf\ c = \tau' \rightarrow \tau$ **and** *vt*: $empty \vdash v : \tau'$
and *vv*: *Values v* **and** *c*: $c = IsZero$
from *c tc* **have** *st*: $\tau' = IntT \wedge \tau = BoolT$ **by** *simp*
from *st vt vv* **obtain** *n* **where** *v*: $v = Const\ (IntC\ n)$
apply *simp* **using** *canonical-form-int* **by** *blast*
let $?VP = Const\ (BoolC\ (n = 0))$
have *wtvp*: $empty \vdash ?VP : BoolT$

using *wte-const*[*of empty BoolC* $(n = 0)$] **by** *auto*
from *c v* **have** *d*: $\delta\ c\ v = Some\ ?VP$ **by** *simp*
from *d wtvp st* **show** *?thesis* **by** *simp*
qed

9.3 Some Inversion Lemmas

lemma *wte-lambda-inv-impl*:
$(\Gamma \vdash e' : \tau' \longrightarrow (\forall\ \sigma\ \sigma'\ e\ \tau.\ e' = \lambda{:}\sigma'.\ e \wedge \tau' = \sigma \rightarrow \tau \longrightarrow$
$\sigma <: \sigma' \wedge (\exists\ L.\ finite\ L \wedge (\forall x.\ x \notin L \longrightarrow \Gamma(x \mapsto \sigma') \vdash \{0 \rightarrow FVar\ x\}e : \tau))))$
$\wedge (\Gamma \vdash m : \sigma\ in\ A \longrightarrow True)$
$\wedge (\Gamma \vdash ms :: \sigma s\ in\ A' \longrightarrow True)$
apply (*induct rule*: *wte-wtm-wtms.induct*)
apply *simp* **apply** *simp*
apply *clarify* **apply** (*rule conjI*) **using** *subtype-reflexive* **apply** *blast*
apply (*rule-tac x=L* **in** *exI*) **apply** *clarify*
apply (*erule-tac x=x* **in** *allE*) **apply** (*erule impE*) **apply** *simp*
apply (*erule conjE*)+ **apply** *assumption*
apply *simp* **apply** *simp*
apply *clarify* **apply** (*rule conjI*) **apply** (*cases rule*: *subtype.cases*) **apply** *force*
apply *force* **apply** *force* **apply** *force* **apply** *force*
apply (*rule subtype-trans*) **apply** *force* **apply** *force* **apply** *force* **apply** *force*
apply (*frule sub-fun-right-inv*)
apply (*erule exE*)+
apply (*erule-tac x=s1* **in** *allE*)
apply (*erule-tac x=σ'a* **in** *allE*)
apply (*erule-tac x=ea* **in** *allE*)
apply (*erule-tac x=s2* **in** *allE*)
apply *clarify* **apply** (*erule impE*) **apply** *simp*
apply (*erule conjE*) **apply** (*erule exE*) **apply** (*erule conjE*)
apply (*rule-tac x=L* **in** *exI*) **apply** (*rule conjI*) **apply** *simp*
apply *clarify* **apply** (*erule-tac x=x* **in** *allE*) **apply** *clarify*
apply (*rule wte-sub*) **apply** *simp*
apply *simp*
apply *simp* **apply** *simp* **apply** *simp* **apply** *simp* **apply** *simp* **apply** *simp* **apply** *simp*
done

lemma *wte-lambda-inv*:
$\Gamma \vdash e' : \tau' \Longrightarrow (\forall\ \sigma\ \sigma'\ e\ \tau.\ e' = \lambda{:}\sigma'.\ e \wedge \tau' = \sigma \rightarrow \tau \longrightarrow$
$\sigma <: \sigma' \wedge (\exists\ L.\ finite\ L \wedge (\forall x.\ x \notin L \longrightarrow \Gamma(x \mapsto \sigma') \vdash \{0 \rightarrow FVar\ x\}e : \tau)))$
using *wte-lambda-inv-impl* **by** *force*

lemma *wte-cast-inv-impl*[*rule-format*]:
$(\Gamma \vdash e' : \tau \longrightarrow (\forall\ e\ \sigma.$
$e' = e\langle\sigma \Rightarrow \tau'\rangle \longrightarrow \tau' <: \tau \wedge \sigma \sim \tau' \wedge \sigma \neq \tau' \wedge \Gamma \vdash e : \sigma))$
$\wedge (\Gamma \vdash m : \sigma\ in\ A \longrightarrow True)$
$\wedge (\Gamma \vdash ms :: \sigma s\ in\ A' \longrightarrow True)$
apply (*induct rule*: *wte-wtm-wtms.induct*)
apply *force* **apply** *force* **apply** *force* **apply** *force*

apply *clarify* **apply** (*erule-tac x=ea* **in** *allE*)
  apply (*erule-tac* $x=\sigma'$ **in** *allE*) **apply** *simp* **apply** (*rule subtype-trans*)
  apply *blast* **apply** *simp*
apply *force* **apply** *force* **apply** *force* **apply** *force* **apply** *force* **apply** *force* **apply** *force*
done

lemma *wte-cast-inv*:
$\Gamma \vdash e\langle\sigma \Rightarrow \tau'\rangle : \tau \Longrightarrow \tau' <: \tau \wedge \sigma \sim \tau' \wedge \sigma \neq \tau' \wedge \Gamma \vdash e : \sigma$
using *wte-cast-inv-impl* **by** *blast*

lemma *wte-const-inv-impl*:
$(\Gamma \vdash e' : T \longrightarrow (\forall\ \sigma\ c\ \tau.\ e' = Const\ c \wedge T = \sigma \to \tau \longrightarrow$
$(\exists\ \sigma'\ \tau'.\ TypeOf\ c = \sigma' \to \tau' \wedge \sigma <: \sigma' \wedge \tau' <: \tau)))$
$\wedge (\Gamma \vdash m : \sigma\ in\ A \longrightarrow True)$
$\wedge (\Gamma \vdash ms :: \sigma s\ in\ A' \longrightarrow True)$
apply (*induct rule*: *wte-wtm-wtms.induct*)
apply *simp+*
apply *clarify*
apply (*frule sub-fun-right-inv*) **apply** (*erule exE*)+
apply *clarify*
apply (*erule-tac x=s1* **in** *allE*)
apply (*erule-tac x=c* **in** *allE*)
apply (*erule-tac x=s2* **in** *allE*)
apply *simp*
apply (*erule exE*)+
apply (*rule-tac* $x=\sigma'a$ **in** *exI*)
apply (*rule-tac* $x=\tau'a$ **in** *exI*)
apply *clarify* **apply** (*rule conjI*)
apply (*rule subtype-trans*) **apply** *simp+*
apply (*rule subtype-trans*) **apply** *simp+*
done

lemma *wte-const-inv*:
$\Gamma \vdash e' : T \Longrightarrow (\forall\ \sigma\ c\ \tau.\ e' = Const\ c \wedge T = \sigma \to \tau \longrightarrow$
$(\exists\ \sigma'\ \tau'.\ TypeOf\ c = \sigma' \to \tau' \wedge \sigma <: \sigma' \wedge \tau' <: \tau))$
using *wte-const-inv-impl* **apply** *force* **done**

lemma *wte-obj-inv-impl*:
$(\Gamma \vdash e' : \tau' \longrightarrow (\forall\ ms\ ss.\ e' = Obj\ ms\ \tau \wedge \tau' = ObjT\ ss \longrightarrow$
$(\exists\ tt.\ \tau = ObjT\ tt \wedge tt <:: ss \wedge \Gamma \vdash ms :: tt\ in\ ObjT\ tt)))$
$\wedge (\Gamma \vdash m : s\ in\ A \longrightarrow True)$
$\wedge (\Gamma \vdash ms :: ss\ in\ A' \longrightarrow True)$
apply (*induct rule*: *wte-wtm-wtms.induct*)
apply *simp* **apply** *simp* **apply** *simp* **apply** *simp*
apply *clarify*
  apply (*frule sub-obj-right-inv*)
  apply (*erule exE*) **apply** (*erule conjE*)
  apply (*erule-tac x=ms* **in** *allE*)
  apply (*erule-tac x=ssa* **in** *allE*)

apply *simp* **apply** *clarify*
apply (*rule-tac x=tt* **in** *exI*) **apply** *simp*
apply (*rule sub-sigs-trans*) **apply** *force* **apply** *force*
apply *simp* **apply** *simp* **apply** *simp* **apply** *simp* **apply** *force* **apply** *simp* **apply** *simp* **apply** *simp*
done

lemma *wte-obj-inv*:
$\Gamma \vdash$ *Obj ms* τ : *ObjT ss* $\Longrightarrow$
($\exists$ *tt*. τ = *ObjT tt* $\wedge$ *tt* <:: *ss* $\wedge$ $\Gamma \vdash$ *ms* :: *tt in ObjT tt*)
using *wte-obj-inv-impl* **by** *blast*

9.4 Some Properties of Objects

lemma *lookup-wtm-impl*:
($\Gamma \vdash e'$: τ' $\longrightarrow$ *True*)
$\wedge$ ($\Gamma \vdash$ *m* : *s in A* $\longrightarrow$ *mname m* = *ms-name s*)
$\wedge$ ($\Gamma \vdash$ *ms* :: *ss in ObjT ss* $\longrightarrow$
lookup ms l = *Some* (*Method l b*) $\longrightarrow$
($\exists$ τ. $\Gamma \vdash$ (*Method l b*) : (*Sig l* τ) *in ObjT ss*))
apply (*induct rule*: *wte-wtm-wtms.induct*)
apply *simp* **apply** *simp* **apply** *simp* **apply** *simp* **apply** *simp* **apply** *simp*
apply *simp* **apply** *simp* **apply** *simp*
apply (*simp add*: *mname-def ms-name-def*) **apply** *simp*
apply (*rule impI*)
apply (*case-tac m*) **apply** *simp*
apply (*case-tac s*) **apply** *simp*
apply (*simp add*: *mname-def ms-name-def*)
apply (*case-tac l* = *mname m*)
apply (*simp add*: *mname-def*)
apply (*rule-tac x=ty* **in** *exI*)
apply *simp*
apply (*simp add*: *mname-def*)
done

lemma *lookup-wtm*:
$[\![$ $\Gamma \vdash$ *ms* :: *ss in ObjT ss*; *lookup ms l* = *Some* (*Method l b*) $]\!]$
$\Longrightarrow$ $\exists$ τ. $\Gamma \vdash$ (*Method l b*) : (*Sig l* τ) *in ObjT ss*
using *lookup-wtm-impl* **by** *blast*

inductive-cases *wtm-inv*:
$\Gamma \vdash$ (*Method l b*) : (*Sig l* τ) *in ObjT ss*

lemma *replace-wt-impl*:
($\Gamma \vdash e'$: τ' $\longrightarrow$ *True*)
$\wedge$ ($\Gamma \vdash$ *m* : *s in A* $\longrightarrow$ *mname m* = *ms-name s*)
$\wedge$ ($\Gamma \vdash$ *ms* :: *ss in* τ $\longrightarrow$ ($\forall$ *msigs m* σ.
$\Gamma \vdash$ *m* : σ *in* τ $\wedge$ *lookup-sig ss* (*mname m*) = *Some* σ
$\longrightarrow$ $\Gamma \vdash$ *replace ms m* :: *ss in* τ))
apply (*induct rule*: *wte-wtm-wtms.induct*)

apply *simp* **apply** *simp* **apply** *simp* **apply** *simp* **apply** *simp* **apply** *simp*
apply *simp* **apply** *simp* **apply** *simp*
apply (*simp add*: *mname-def ms-name-def*)
apply *simp*
apply *clarify*
  apply (*case-tac mname m* = *mname ma*)
    apply *simp*
    apply (*rule wt-cons*)
      apply (*simp add*: *mname-def ms-name-def*)
      apply (*erule-tac x*=*m* **in** *allE*)
      apply *blast*
  apply *simp*
  apply (*rule wt-cons*)
    apply *simp*
    apply (*erule-tac x*=*ma* **in** *allE*)
    apply *simp*
done

lemma *replace-wt*:
⟦ Γ ⊢ *ms* :: *ss in* τ; Γ ⊢ *m* : *s in* τ; *lookup-sig ss* (*mname m*) = *Some s* ⟧
⟹ Γ ⊢ *replace ms m* :: *ss in* τ
using *replace-wt-impl* **by** *simp*

lemma *method-sig-name*:
Γ ⊢ *m* : *s in A* ⟹ *mname m* = *ms-name s*
using *replace-wt-impl* **by** *simp*

lemma *lookup-name-result*[*rule-format*]:
∀ *l* σ. *lookup-sig msigs l* = *Some s* ⟶ *ms-name s* = *l*
apply (*induct msigs*)
apply *force* **apply** *force* **apply** *force* **apply** *force* **apply** *force*
apply *force* **apply** *force* **apply** *force* **apply** (*simp add*: *ms-name-def*)
done

lemma *sub-obj-left-inv*:
ObjT ss <: τ ⟹ ∃ *tt*. τ = *ObjT tt* ∧ *ss* <:: *tt*
apply (*cases rule*: *subtype.cases*) **by** *auto*

lemma *lookup-sub-impl*:
(σ <: τ ⟶ *True*)
∧ (*s* ⪯ *t* ⟶ *True*)
∧ (*ss* <:: *msigs* ⟶ (∀ *l s*. *lookup-sig msigs l* = *Some s* ⟶ *lookup-sig ss l* = *Some s*))
apply (*induct rule*: *subtype-subtype-sig-subtype-sigs.induct*)
apply *force* **apply** *force* **apply** *force* **apply** *force* **apply** *force* **apply** *force*
apply *force* **apply** *force* **apply** *force*
apply *clarify*
  apply (*cases rule*: *subtype-sig.cases*) **apply** *simp*
  apply *simp* **apply** *force*
apply *clarify*

apply (*erule-tac x=l* **in** *allE*) **apply** (*erule-tac x=sa* **in** *allE*)
apply *clarify*
sorry

lemma *lookup-sub*:
⟦ *ss* <:: *tt*; *lookup-sig tt l* = *Some s* ⟧ ⟹ *lookup-sig ss l* = *Some s*
using *lookup-sub-impl* **by** *blast*

lemma *method-subsumption-impl*:
(Γ ⊢ *e* : σ ⟶ *True*)
∧ (Γ ⊢ *m* : *s in* σ ⟶ τ <: σ ⟶ Γ ⊢ *m* : *s in* τ)
∧ (Γ ⊢ *ms* :: *ss in* σ ⟶ τ <: σ ⟶ Γ ⊢ *ms* :: *ss in* τ)
apply (*induct rule*: *wte-wtm-wtms.induct*)
apply *simp-all*
apply *clarify*
apply (*frule sub-obj-right-inv*) **apply** (*erule exE*) **apply** *clarify*
apply (*rule wt-mtd*)
apply (*rule wte-sub*) **apply** *simp* **apply** *force*
using *lookup-sub* **apply** *blast*
apply *clarify* **apply** (*rule wt-nil*)
apply *clarify*
apply (*rule wt-cons*)
apply *simp*
apply *simp*
done

lemma *method-subsumption*:
⟦ Γ ⊢ *m* : *s in* σ; τ <: σ ⟧ ⟹ Γ ⊢ *m* : *s in* τ
using *method-subsumption-impl* **by** *blast*

lemma *methods-subsumption*:
⟦ Γ ⊢ *ms* :: *ss in* σ; τ <: σ ⟧ ⟹ Γ ⊢ *ms* :: *ss in* τ
using *method-subsumption-impl* **by** *blast*

lemma *lookup-sig-implies-lookup-impl*:
(Γ ⊢ *e* : τ ⟶ *True*)
∧ (Γ ⊢ *m* : *s in A* ⟶ *mname m* = *ms-name s*)
∧ (Γ ⊢ *ms* :: *ss in A* ⟶ *lookup-sig ss l* = *Some* (*Sig l* τ)
⟶ (∃ *b*. *lookup ms l* = *Some* (*Method l b*)))
apply (*induct rule*: *wte-wtm-wtms.induct*)
apply *simp* **apply** *simp* **apply** *simp* **apply** *simp* **apply** *simp* **apply** *simp* **apply** *simp*
apply *simp* **apply** *simp* **apply** (*simp add*: *mname-def ms-name-def*) **apply** *simp*
apply *clarify*
apply (*case-tac m*)
apply (*case-tac s*)
apply (*case-tac l* = *nat*)
apply (*simp add*: *mname-def ms-name-def*)
apply (*erule impE*)
apply (*simp add*: *mname-def ms-name-def*)

apply (*erule exE*)
apply (*simp add*: *mname-def ms-name-def*)
done

lemma *lookup-sig-implies-lookup*:
$[\![\, \Gamma \vdash ms :: ss\ in\ A;\ lookup\text{-}sig\ ss\ l = Some\ (Sig\ l\ \tau)\ ]\!]$
$\Longrightarrow (\exists\ b.\ lookup\ ms\ l = Some\ (Method\ l\ b))$
using *lookup-sig-implies-lookup-impl* **by** *blast*

9.5 Subject Reduction

inductive-cases *app-reduce*: $App\ e1\ e2 \longrightarrow e'$
inductive-cases *cast-reduce*: $e\langle\sigma\Rightarrow\tau\rangle \longrightarrow e'$

lemma *subject-reduction-impl*:
$(\Gamma \vdash e : \tau \longrightarrow \Gamma = empty \wedge e \longrightarrow e' \longrightarrow empty \vdash e' : \tau)$
$\wedge (\Gamma \vdash m : \sigma\ in\ A \longrightarrow True)$
$\wedge (\Gamma \vdash ms :: \sigma s\ in\ A' \longrightarrow True)$
apply (*induct rule*: *wte-wtm-wtms.induct*)
apply *simp-all*
apply *force*
apply *clarify* **apply** (*cases rule*: *reduces.cases*) **apply** *simp+*
apply *clarify* **apply** (*cases rule*: *reduces.cases*) **apply** *simp+*
apply *clarify* **defer**
apply *clarify* **apply** *simp* **apply** (*rule wte-sub*) **apply** *simp* **apply** *simp*
apply *clarify* **defer**
apply *clarify* **defer**
apply *clarify* **defer**
apply *clarify* **apply** (*cases rule*: *reduces.cases*) **apply** *simp+*
proof −
— Beta
fix Γ::*env* **and** $\sigma\ \tau\ e1\ e2$
assume *wte1*: $empty \vdash e1 : \sigma \rightarrow \tau$
and *wte2*: $empty \vdash e2 : \sigma$
and *red*: $App\ e1\ e2 \longrightarrow e'$
from *red* **show** $empty \vdash e' : \tau$
proof (*rule app-reduce*)
fix $\tau''\ b$ **assume** *vv*: *Values e2* **and** *ep*: $e' = \{0 \rightarrow e2\}b$ **and** *e1*: $e1 = \lambda{:}\tau''.\ b$
from *wte1 e1* **have** *wte1b*: $empty \vdash (\lambda{:}\tau''.\ b) : \sigma \rightarrow \tau$ **by** *simp*
from *wte1b* **obtain** *L*
where *st*: $\sigma <: \tau''$
and *fL*: *finite L*
and *wtb*: $\forall x.\ x \notin L \longrightarrow [x \mapsto \tau''] \vdash \{0 \rightarrow FVar\ x\}b : \tau$
using *wte-lambda-inv*[*of empty* $(\lambda{:}\tau''.\ b)\ \sigma \rightarrow \tau$] **by** *blast*
let $?X = Suc\ (max\ (setmax\ L)\ (setmax\ (FV\ b)))$
have *xgel*: $setmax\ L < ?X$ **by** *auto*
have *xgeb*: $setmax\ (FV\ b) < ?X$ **by** *auto*

— Set up for and apply the substitution lemma
from *fL xgel* **have** *xL*: $?X \notin L$ **by** (*rule greaterthan-max-is-fresh*)

with *wtb* **have** *wtb2*: $[?X \mapsto \tau''] \vdash \{0{\to}FVar\ ?X\}b : \tau$ **by** *blast*
have *gxs*: $[?X \mapsto \tau'']\ ?X = Some\ \tau''$ **by** *simp*
have *fg*: *finite-env* $[?X \mapsto \tau'']$ **using** *finite-env-def* **by** *simp*
have *fgp*: *finite-env empty* **using** *finite-env-def* **by** *simp*
have *gxgp*: $[?X \mapsto \tau''] - ?X \subset empty$ **by** (*simp add*: *remove-bind-def*)
from *wte2 st* **have** *wte2b*: $empty \vdash e2 : \tau''$ **by** (*rule wte-sub*)
from *wtb2 gxs fg fgp gxgp wte2b*
have *wtb*: $empty \vdash [?X{\to}e2](\{0{\to}FVar\ ?X\}b) : \tau$
using *substitution* **by** *blast*

— Use the substitution decomposition lemma
have *finb*: *finite* $(FV\ b)$ **by** (*rule finite-FV*)
from *finb xgeb* **have** *xb*: $?X \notin FV\ b$ **by** (*rule greaterthan-max-is-fresh*)
from *xb* **have** $\{0{\to}e2\}b = [?X{\to}e2](\{0{\to}FVar\ ?X\}b)$
by (*rule decompose-subst*)
with *wtb ep* **show** $empty \vdash e' : \tau$ **by** *simp*
next — Delta
fix *c* **assume** *d*: $\delta\ c\ e2 = Some\ e'$
and *ve2*: *Values e2*
and *e1*: $e1 = (Const\ c)$
from *wte1 e1* **obtain** $\sigma'\ \tau'$ **where** *tc*: $TypeOf\ c = \sigma' \to \tau'$
and *ss*: $\sigma <: \sigma'$ **and** *tt*: $\tau' <: \tau$
apply *simp* **using** *wte-const-inv* **by** *blast*
from *wte2 ss* **have** *wte2b*: $empty \vdash e2 : \sigma'$ **by** (*rule wte-sub*)

from *tc wte2b ve2* **obtain** v'' **where** *dd*: $\delta\ c\ e2 = Some\ v''$
and *wtvp*: $empty \vdash v'' : \tau'$ **using** *delta-typability* **by** *blast*
from *d dd wtvp* **have** *wtep*: $empty \vdash e' : \tau'$ **by** *simp*
from *wtep tt* **show** $empty \vdash e' : \tau$ **by** (*rule wte-sub*)
next — ApCst
fix $\nu\ \varrho\ \sigma 1\ \sigma'\ \tau 1\ f$
let $?arg = mcast\ e2\ \varrho\ \sigma 1$
assume *ep*: $e' = mcast\ (App\ f\ ?arg)\ \tau 1\ \nu$
and *e1*: $e1 = f\langle\sigma 1 \to \tau 1 \Rightarrow \varrho \to \nu\rangle$

from *wte1 e1* **have** *wte1a*: $empty \vdash f\langle\sigma 1 \to \tau 1 \Rightarrow \varrho \to \nu\rangle : \sigma \to \tau$ **by** *simp*
from *wte1a* **have** *rnst*: $\varrho \to \nu <: \sigma \to \tau$
and *wtf*: $empty \vdash f : \sigma 1 \to \tau 1$
and *s1t1rv*: $\sigma 1 \to \tau 1 \sim \varrho \to \nu$
and *s1t1rvne*: $\sigma 1 \to \tau 1 \neq \varrho \to \nu$
using *wte-cast-inv*[*of empty f* $\sigma 1 \to \tau 1\ \ \varrho \to \nu$] **by** *auto*
from *rnst* **have** *sr*: $\sigma <: \varrho$ **by** *auto*
from *rnst* **have** *vt*: $\nu <: \tau$ **by** *auto*

from *wte2 sr* **have** *e2b*: $empty \vdash e2 : \varrho$ **by** (*rule wte-sub*)
from *e2b s1t1rv s1t1rvne* **have** *wtce2*: $empty \vdash ?arg : \sigma 1$
apply (*simp add*: *mcast-def*)
apply (*case-tac* $\varrho = \sigma 1$)
apply *auto* **apply** (*rule wte-cast*) **apply** *auto*
using *consistent-symmetric* **apply** *force*

apply (*rule wte-cast*) **apply** *auto* **using** *consistent-symmetric* **apply** *blast*
done

from *wtf wtce2* **have** *wtap*: *empty* $\vdash$ *App f ?arg* : $\tau 1$ **by** (*rule wte-app*)
from *s1t1rv* **have** *t1n*: $\tau 1 \sim \nu$ **by** *auto*
from *wtap t1n* **have** *empty* $\vdash$ *mcast* (*App f ?arg*) $\tau 1$ ν : ν
apply (*simp add*: *mcast-def*)
apply (*case-tac* $\tau 1 = \nu$)
apply *auto* **apply** (*rule wte-cast*)
apply *auto* **apply** (*rule wte-cast*) **apply** *auto*
done
with *ep vt* **show** *empty* $\vdash$ e' : τ **apply** *simp* **apply** (*rule wte-sub*) **by** *auto*
qed
next — Cast
fix Γ σ τ e
assume *wte*: *empty* $\vdash$ e : σ **and** *IH*: *empty* = *empty* $\wedge$ $e \longrightarrow e' \longrightarrow$ *empty* $\vdash$ e' :
σ
and *st*: $\sigma \sim \tau$ **and** *nst*: $\sigma \neq \tau$
and *red*: $e\langle\sigma\Rightarrow\tau\rangle \longrightarrow e'$
from *red* **show** *empty* $\vdash$ e' : τ
proof (*rule cast-reduce*)
fix ϱ σ' v
assume *rtstr*: $\varrho \lesssim \tau$
and *nrt*: $\varrho \neq \tau$ **and** *ep*: $e' = $ *mcast* v ϱ ($\varrho \leftharpoonup \tau$)
and *e*: $e = v\langle\varrho\Rightarrow\sigma'\rangle$
from *wte e* **have** *wtcv*: *empty* $\vdash$ $v\langle\varrho\Rightarrow\sigma'\rangle$: σ **by** *simp*
from *wtcv* **have** *ss*: σ' <: σ
and *rsp*: $\varrho \sim \sigma'$
and *nrsp*: $\varrho \neq \sigma'$
and *wtv*: *empty* $\vdash$ v : ϱ
using *wte-cast-inv*[*of empty* v ϱ σ' σ] **by** *auto*

have *rrt*: $\varrho \sim \varrho \leftharpoonup \tau$ **by** (*rule consistent-merge*)
from *wtv rrt* **have** *wtcv*: *empty* $\vdash$ *mcast* v ϱ ($\varrho \leftharpoonup \tau$) : $\varrho \leftharpoonup \tau$
apply (*simp add*: *mcast-def*)
apply (*case-tac* $\varrho = \varrho \leftharpoonup \tau$)
apply *simp* **apply** *clarify*
apply (*rule wte-cast*) **apply** *auto*
done
from *rtstr* **have** *rtt2*: $\varrho \leftharpoonup \tau$ <: τ
using *restrict-sub-merge* **by** *blast*
from *wtcv rtt2* **have** *wtcv2*: *empty* $\vdash$ *mcast* v ϱ ($\varrho \leftharpoonup \tau$) : τ **by** (*rule wte-sub*)
with *ep* **show** *empty* $\vdash$ e' : τ **by** *simp*
next
fix σ **assume** *e*: $e = e'\langle\tau\Rightarrow\sigma\rangle$
from *wte e* **show** *empty* $\vdash$ e' : τ
using *wte-cast-inv* **by** *blast*
qed
next — Sel
fix Γ τ e l *ss*

assume *wte*: *empty* ⊢ *e* : *ObjT ss*
and *X*: *empty* = *empty* ∧ *e* −→ *e*′ ⟶ *empty* ⊢ *e*′ : *ObjT ss*
and *lm*: *lookup-sig ss l* = *Some* (*Sig l* τ)
and *red*: *Invoke e l* −→ *e*′
from *red* **show** *empty* ⊢ *e*′ : τ
apply (*cases rule*: *reduces.cases*)
apply *simp* **apply** *simp* **defer apply** *simp* **apply** *simp* **defer apply** *simp* **apply** *simp* **apply** *simp*
proof −
fix τ′ *b l*′ *ms*′
assume *a*: (*Invoke e l*, *e*′) = (*Invoke* (*Obj ms*′ τ′) *l*′, *App b* (*Obj ms*′ τ′))
and *lm2*: *lookup ms*′ *l*′ = *Some* (*Method l*′ *b*)
from *wte a* **have** *wto*: *empty* ⊢ *Obj ms*′ τ′ : *ObjT ss* **by** *simp*
from *wto* **obtain** *tt* **where** *t*: τ′ = *ObjT tt* **and** *ttss*: *tt* <:: *ss*
and *wtms*: *empty* ⊢ *ms*′ :: *tt in ObjT tt*
using *wte-obj-inv* **by** *blast*

from *wtms lm2* **obtain** τ′ **where**
wtm: *empty* ⊢ (*Method l*′ *b*) : (*Sig l*′ τ′) *in ObjT tt*
using *lookup-wtm* **by** *force*
from *wtm* **have** *wtb*: *empty* ⊢ *b* : *ObjT tt* → τ′
using *wtm-inv*[*of empty l*′ *b* τ′ *tt*] **by** *blast*
from *wtm* **have** *lm2*: *lookup-sig tt l*′ = *Some* (*Sig l*′ τ′)
using *wtm-inv*[*of empty l*′ *b* τ′ *tt*] **by** *blast*

from *wtms* **have** *wto2*: *empty* ⊢ *Obj ms*′ (*ObjT tt*) : *ObjT tt* **by** (*rule wte-obj*)
from *wtb wto2* **have** *wta*: *empty* ⊢ *App b* (*Obj ms*′ (*ObjT tt*)) : τ′ **by** (*rule wte-app*)

from *a lm lm2 ttss* **have** τ = τ′
apply *simp* **using** *lookup-sub* **by** *simp*
with *wta a t* **show** *empty* ⊢ *e*′ : τ **by** *simp*
next — SelCst
fix σ′ τ′ *obj l*′ *ss tt*
assume *a*: (*Invoke e l*, *e*′) = (*Invoke* (*obj*⟨*ObjT ss*⇒*ObjT tt*⟩) *l*′, *mcast* (*Invoke obj l*′) σ′ τ′)
and *lookup-sig ss l*′ = *Some* (*Sig l*′ σ′)
and *lookup-sig tt l*′ = *Some* (*Sig l*′ τ′)
show *empty* ⊢ *e*′ : τ **sorry**
qed
next — Upd
fix Γ *e l m s ss*
assume *wte*: *empty* ⊢ *e* : *ObjT ss*
and *wtm*: *empty* ⊢ *m* : *s in ObjT ss*
and *lm*: *lookup-sig ss l* = *Some s*
and *red*: (*Update e m*) −→ *e*′
and ¬ (*empty* = *empty* ∧ *e* −→ *e*′)
from *red* **show** *empty* ⊢ *e*′ : *ObjT ss*
apply (*cases rule*: *reduces.cases*)
apply *simp* **apply** *simp* **apply** *simp* **defer apply** *simp* **apply** *simp* **defer apply**

simp **apply** *simp*
  proof –
    fix τ' *m′ ms*
    assume *a*: (*Update e m*, *e′*) = (*Update* (*Obj ms* τ') *m′*, *Obj* (*replace ms m′*) τ')
    from *wte a* **have** *wto*: *empty* ⊢ *Obj ms* τ' : *ObjT ss* **by** *simp*
    from *lm* **have** *snl*: *ms-name s* = *l* **by** (*rule lookup-name-result*)
    from *wtm* **have** *mnsn*: *mname m* = *ms-name s* **by** (*rule method-sig-name*)
    from *snl mnsn lm* **have** *lm2*: *lookup-sig ss* (*mname m*) = *Some s* **by** *simp*
    from *wto* **obtain** *tt* **where** *t*: τ' = *ObjT tt* **and** *ttss*: *tt* <:: *ss*
      and *wtms*: *empty* ⊢ *ms* :: *tt in ObjT tt*
      using *wte-obj-inv* **by** *blast*
    from *ttss* **have** *osom*: *ObjT tt* <: *ObjT ss* **by** *auto*
   from *wtm osom* **have** *wtm2*: *empty* ⊢ *m* : *s in ObjT tt* **by** (*rule method-subsumption*)
    from *lm2 ttss* **have** *lm3*: *lookup-sig tt* (*mname m*) = *Some s*
      using *lookup-sub* **by** *simp*
    from *wtms wtm2 lm3* **have** *empty* ⊢ *replace ms m* :: *tt in ObjT tt*
      by (*rule replace-wt*)
    hence *wto2*: *empty* ⊢ *Obj* (*replace ms m*) (*ObjT tt*) : *ObjT tt* **by** (*rule wte-obj*)
    from *wto2 osom* **have** *empty* ⊢ *Obj* (*replace ms m*) (*ObjT tt*) : *ObjT ss* **by** (*rule wte-sub*)
    with *a t* **show** *?thesis* **by** *simp*
  next — UpdCst
    fix σ' τ' *b obj l m′ ss′ tt*
     assume (*Update e m*, *e′*) = (*Update* (*obj*⟨*ObjT ss′*⇒*ObjT tt*⟩) (*Method l b*), (*Update obj m′*)⟨*ObjT ss′*⇒*ObjT tt*⟩)
      and *lookup-sig ss′ l* = *Some* (*Sig l* σ')
      and *lookup-sig tt l* = *Some* (*Sig l* τ')
      and *m′* = *Method l* (*b*⟨*ObjT tt* → τ'⇒*ObjT ss′* → σ'⟩)
    show *empty* ⊢ *e′* : *ObjT ss* **sorry**
  qed
qed

lemma *subject-reduction*:
  assumes *wte*: $\Gamma \vdash e : \tau$ **and** *g*: Γ = *empty* **and** *red*: $e \longrightarrow e'$
  shows *empty* ⊢ $e' : \tau$
  using *wte g red subject-reduction-impl* **by** *simp*

9.6 The Decomposition Lemma

consts *welltyped-ctx* :: (*env* × *ctx* × *ty* × *ty*) *set*
syntax *welltyped-ctx* :: *env* ⇒ *ctx* ⇒ *ty* ⇒ *ty* ⇒ *bool* (- ⊢ - : -⇒- [*52*,*52*,*52*,*52*] *51*)
translations $\Gamma \vdash E : \sigma \Rightarrow \tau$ == (Γ, *E*, σ, τ) ∈ *welltyped-ctx*
inductive *welltyped-ctx* **intros**
 WTHole: $\Gamma \vdash$ *Hole* : $\tau \Rightarrow \tau$
 WTAppL: ⟦ $\Gamma \vdash E : \sigma \Rightarrow (\varrho \to \tau)$; $\Gamma \vdash e : \varrho$ ⟧
 ⟹ $\Gamma \vdash$ *AppL E e* : $\sigma \Rightarrow \tau$
 WTAppR: ⟦ $\Gamma \vdash e : \varrho \to \tau$; $\Gamma \vdash E : \sigma \Rightarrow \varrho$ ⟧
 ⟹ $\Gamma \vdash$ *AppR e E* : $\sigma \Rightarrow \tau$
 WTCSub: ⟦ $\Gamma \vdash E : \sigma \Rightarrow \varrho$; ϱ <: ϱ' ⟧
 ⟹ $\Gamma \vdash E : \sigma \Rightarrow \varrho'$

```
WTCSel: ⟦ Γ ⊢ E : σ ⇒ ObjT ss; lookup-sig ss l = Some (Sig l τ) ⟧
 ⟹Γ ⊢ InvokeC E l : σ ⇒ τ
WTCUpd: ⟦Γ ⊢ E : σ ⇒ ObjT ss;
        Γ ⊢ m : s in ObjT ss;
        lookup-sig ss l = Some s ⟧
       ⟹ Γ ⊢ UpdateC E m : σ ⇒ ObjT ss
WTCCast: ⟦ Γ ⊢ E : σ ⇒ ϱ; ϱ ∼ ϱ'; ϱ ≠ ϱ' ⟧
 ⟹ Γ ⊢ CastC E ϱ ϱ' : σ ⇒ ϱ'


constdefs bad-cast :: expr ⇒ bool
 bad-cast e ≡ (∃ v ϱ σ σ' τ. e = (v⟨ϱ⇒σ⟩)⟨σ'⇒τ⟩ ∧ SimpleValues v
        ∧ ¬(ϱ ≲ τ) )

lemma welltyped-decomposition-impl:
 (Γ ⊢ e : τ ⟶
   Γ = empty ⟶ Values e
      ∨ (∃ σ E r. e = E[r] ∧ Γ ⊢ E : σ ⇒ τ ∧ E ∈ wf-ctx
                          ∧ Γ ⊢ r : σ ∧ (redex r ∨ bad-cast r)))
 ∧ (Γ ⊢ m : s in A ⟶ True)
 ∧ (Γ ⊢ ms :: ss in A' ⟶ True)
 apply (induct rule: wte-wtm-wtms.induct)
 apply simp apply simp apply simp
 apply (rule impI) defer
 apply (rule impI) apply (case-tac Values e)
  apply simp
  apply (erule impE) apply simp
  apply (erule disjE) apply simp
  apply (erule exE)+ apply simp
  apply (rule-tac x=σ' in exI)
  apply (rule-tac x=E in exI)
  apply (rule-tac x=r in exI)
  apply simp
  apply clarify apply (rule WTCSub) apply simp apply simp
 apply (rule impI) apply (case-tac Values e)
  apply (erule impE) apply simp
  apply simp apply (case-tac e) apply simp apply simp apply simp
  apply simp apply simp apply simp
  apply (rule-tac x=τ in exI)
  apply (rule-tac x=Hole in exI)
  apply (rule-tac x=Cast (Cast expr ty1 ty2) σ τ in exI)
  apply simp apply (rule conjI)
  apply (rule WTHole) apply (rule conjI) apply (rule WFHole)
  apply (rule conjI) apply (rule wte-cast) apply simp apply simp apply simp
  apply (simp add: redex-def)
  apply (case-tac ty1 = τ) using Remove apply blast
  apply (case-tac (ty1 ≲ τ ∧ ty1 ≠ τ))
  using Merge apply blast
  apply (simp add: bad-cast-def)
```

apply *simp* **apply** *simp* **apply** *simp* **apply** *simp*
apply (*erule exE*)+ **apply** (*erule conjE*)+
apply (*rule disjI2*) **apply** *clarify*
apply (*rule-tac* $x=\sigma'$ **in** *exI*)
apply (*rule-tac* $x=$*CastC E* σ τ **in** *exI*)
apply (*rule-tac* $x=r$ **in** *exI*)
apply *simp* **apply** (*rule conjI*) **apply** (*rule WTCCast*) **apply** *simp* **apply** *simp*
apply *simp* **apply** (*rule WFCastC*) **apply** *simp*
apply (*rule impI*) **defer**
apply (*rule impI*) **defer**
apply *simp*
apply *simp*
apply *simp*
apply *simp*
proof −
fix Γ σ τ *e1 e2*
assume *wte1*: $\Gamma \vdash$ *e1* : $\sigma \rightarrow \tau$
and *IH1*: Γ = *empty* $\longrightarrow$
Values e1 $\vee$
($\exists\,\sigma'$ *E r*.
e1 = *E*[*r*] $\wedge$
$\Gamma \vdash$ *E* : $\sigma'\Rightarrow\sigma \rightarrow \tau$ $\wedge$ *E* $\in$ *wf-ctx* $\wedge$ $\Gamma \vdash r : \sigma'$ $\wedge$ (*redex r* $\vee$ *bad-cast r*))
and *wte2*: $\Gamma \vdash$ *e2* : σ
and *IH2*: Γ = *empty* $\longrightarrow$
Values e2 $\vee$
($\exists\,\sigma'$ *E r*.
e2 = *E*[*r*] $\wedge$ $\Gamma \vdash$ *E* : $\sigma'\Rightarrow\sigma$ $\wedge$ *E* $\in$ *wf-ctx* $\wedge$ $\Gamma \vdash r : \sigma'$ $\wedge$ (*redex r* $\vee$ *bad-cast r*))
and *g*: Γ = *empty*
show *Values* (*App e1 e2*) $\vee$
($\exists\,\sigma$ *E r*. *App e1 e2* = *E*[*r*] $\wedge$ $\Gamma \vdash$ *E* : $\sigma\Rightarrow\tau$ $\wedge$ *E* $\in$ *wf-ctx* $\wedge$ $\Gamma \vdash r : \sigma$ $\wedge$ (*redex r* $\vee$ *bad-cast r*))
proof (*cases Values e1*)
assume *ve1*: *Values e1*
show *?thesis*
proof (*cases Values e2*)
assume *ve2*: *Values e2*
have *h*: *App e1 e2* = *Hole*[*App e1 e2*] **by** *simp*
have *wth*: *empty* $\vdash$ *Hole* : $\tau\Rightarrow\tau$ **by** (*rule WTHole*)
from *wte1 wte2 g* **have** *wta*: *empty* $\vdash$ *App e1 e2* : τ **apply** *simp* **by** (*rule wte-app*)

from *wte1 ve1 g* **have** ($\exists$ σ' *e*. *e1* = λ:σ'. *e*) $\vee$ ($\exists$ *c*. *e1* = *Const c*)
$\vee$ ($\exists$ σ' τ' v' ϱ ν. *e1* = $v'\langle\sigma'\rightarrow\tau' \Rightarrow \varrho\rightarrow\nu\rangle$)
apply *simp* **apply** (*rule canonical-form-fun*) **by** *auto*
moreover { **assume** *x*: $\exists$ σ' *e*. *e1* = λ:σ'. *e*
— Beta
from *x* **obtain** σ' *b* **where** *e1*: *e1* = λ:σ'. *b* **by** *blast*
from *e1 ve2* **have** *App e1 e2* $-\!\!\rightarrow$ {*0*$\rightarrow$*e2*}*b* **apply** *simp* **by** (*rule Beta*)
hence *r*: *redex* (*App e1 e2*) **using** *redex-def* **by** *blast*

```
        have wfh: Hole ∈ wf-ctx by (rule WFHole)
        from h wth wfh wta r g have ?thesis by blast
      } moreover { assume x: ∃ c. e1 = Const c
          — Delta
        from x obtain c where e1: e1 = Const c by blast
        from wte1 e1 obtain σ' τ' where tc: TypeOf c = σ' → τ'
          and ss: σ <: σ' and tt: τ' <: τ
          apply simp using wte-const-inv by blast
        from wte2 have wte2b: Γ ⊢ e2 : σ' by (rule wte-sub)
        from tc wte2b ve2 g obtain v'' where dd: δ c e2 = Some v''
          using delta-typability by blast
        from dd ve2 e1 have App e1 e2 −→ v'' apply simp by (rule Delta)
        hence r: redex (App e1 e2) using redex-def by blast
        have wfh: Hole ∈ wf-ctx by (rule WFHole)
        with h wth wfh wta r g have ?thesis by blast
      } moreover { assume x: (∃ σ' τ' v ϱ ν. e1 = v⟨σ'→τ' ⇒ ϱ→ν⟩)
        — ApCst
        from x obtain σ' τ' f ϱ ν where e1: e1 = f⟨σ'→τ' ⇒ ϱ→ν⟩ by blast
        from ve1 e1 have sf: SimpleValues f by simp
        from e1 sf ve2 have App e1 e2 −→ mcast (App f (mcast e2 ϱ σ')) τ' ν
          apply simp by (rule ApCst)
        hence r: redex (App e1 e2) using redex-def by blast
        have wfh: Hole ∈ wf-ctx by (rule WFHole)
        from h wth wfh wta r g have ?thesis by blast
      } ultimately show ?thesis by blast
    next
      assume ve2: ¬ Values e2
      from ve2 IH2 g obtain σ' E r where e2: e2 = E[r]
        and wtE: Γ ⊢ E : σ'⇒σ and wfE: E ∈ wf-ctx
        and wtr: Γ ⊢ r : σ' and rr: (redex r ∨ bad-cast r)
        by blast
      from e2 have App e1 e2 = (AppR e1 E)[r] by simp
      moreover from wte1 wtE g have empty ⊢ AppR e1 E : σ' ⇒ τ
        apply simp apply (rule WTAppR) apply auto done
      moreover from ve1 wfE have AppR e1 E ∈ wf-ctx by (rule WFAppR)
      moreover note wtr rr g
      ultimately show ?thesis by blast
    qed
  next
    assume ve1: ¬ Values e1
    from ve1 IH1 g obtain σ' E r where e1: e1 = E[r]
      and wtE: Γ ⊢ E : σ'⇒σ→τ and wfE: E ∈ wf-ctx and wtr: Γ ⊢ r : σ' and rr:
(redex r ∨ bad-cast r)
      by blast
    from e1 have App e1 e2 = (AppL E e2)[r] by simp
    moreover from wtE wte2 g have empty ⊢ AppL E e2 : σ' ⇒ τ
      apply simp apply (rule WTAppL) apply auto done
    moreover from wfE have AppL E e2 ∈ wf-ctx by (rule WFAppL)
    moreover note wtr rr g
    ultimately show ?thesis by blast
```

qed
next
fix Γ τ *e l msigs*
assume *wte*: $\Gamma \vdash e : ObjT\ msigs$
and *IH*: $\Gamma = empty \longrightarrow$
Values e $\vee$
($\exists\sigma\ E\ r.$
$e = E[r] \wedge$
$\Gamma \vdash E : \sigma \Rightarrow ObjT\ msigs \wedge E \in wf\text{-}ctx \wedge \Gamma \vdash r : \sigma \wedge (redex\ r \vee bad\text{-}cast\ r))$
and *lm*: *lookup-sig msigs l* = *Some* (*Sig l* τ)
and *g*: $\Gamma = empty$
show *Values* (*Invoke e l*) $\vee$
($\exists\sigma\ E\ r.$
$Invoke\ e\ l = E[r] \wedge \Gamma \vdash E : \sigma \Rightarrow \tau \wedge E \in wf\text{-}ctx \wedge \Gamma \vdash r : \sigma \wedge (redex\ r \vee bad\text{-}cast\ r))$
proof (*cases Values e*)
assume *ve*: *Values e*
have *a*: *Invoke e l* = *Hole*[*Invoke e l*] **by** *simp*
have *wth*: $\Gamma \vdash Hole : \tau \Rightarrow \tau$ **by** (*rule WTHole*)
have *wfh*: $Hole \in wf\text{-}ctx$ **by** (*rule WFHole*)
from *wte lm* **have** *wtr*: $\Gamma \vdash Invoke\ e\ l : \tau$ **by** (*rule wte-sel*)
from *wte ve g* **have** *x*: $(\exists\ ms\ \tau.\ e = Obj\ ms\ \tau) \vee (\exists\ ms\ \tau\ rr\ tt.\ e = (Obj\ ms\ \tau)\langle ObjT\ rr \Rightarrow ObjT\ tt\rangle)$
apply *simp* **apply** (*rule canonical-form-obj*) **by** *auto*
moreover { **assume** *x*: $\exists\ ms\ \tau.\ e = Obj\ ms\ \tau$
from *x* **obtain** *ms* τ' **where** *e*: $e = Obj\ ms\ \tau'$ **by** *blast*
from *wte e* **have** *wto*: $\Gamma \vdash Obj\ ms\ \tau' : ObjT\ msigs$ **by** *simp*
from *wto* **obtain** *tt* **where** *tp*: $\tau' = ObjT\ tt$ **and** *ssm*: *tt* <:: *msigs*
and *wtms*: $\Gamma \vdash ms :: tt\ in\ ObjT\ tt$ **using** *wte-obj-inv* **by** *blast*
from *ssm lm* **have** *lm2*: *lookup-sig tt l* = *Some* (*Sig l* τ)
using *lookup-sub* **by** *blast*
from *wtms lm2* **have** *x*: $\exists\ b.$ *lookup ms l* = *Some* (*Method l b*)
by (*rule lookup-sig-implies-lookup*)
from *x* **obtain** *b* **where** *lmb*: *lookup ms l* = *Some* (*Method l b*) **by** *blast*
from *lmb* **have** *red*: *Invoke* (*Obj ms* τ') *l* $\longrightarrow$ *App b* (*Obj ms* τ') **by** (*rule Sel*)
from *red e* **have** *r*: *redex* (*Invoke e l*) **using** *redex-def* **by** *blast*
from *a wth wfh wtr r* **have** *?thesis* **by** *blast*
} **moreover** { **assume** *x*: $\exists\ ms\ \tau\ rr\ tt.\ e = (Obj\ ms\ \tau)\langle ObjT\ rr \Rightarrow ObjT\ tt\rangle$
have *?thesis* **sorry**
} **ultimately show** *?thesis* **by** *blast*
next
assume *ve*: $\neg$ *Values e*
from *ve IH g* **obtain** σ' *E r* **where** *e*: $e = E[r]$
and *wtE*: $\Gamma \vdash E : \sigma' \Rightarrow ObjT\ msigs$ **and** *wfE*: $E \in wf\text{-}ctx$
and *wtr*: $\Gamma \vdash r : \sigma'$ **and** *rr*: (*redex r* $\vee$ *bad-cast r*) **by** *blast*
from *e* **have** *Invoke e l* = (*InvokeC E l*)[*r*] **by** *simp*
moreover from *wtE g lm* **have** $empty \vdash InvokeC\ E\ l : \sigma' \Rightarrow \tau$
apply *simp* **apply** (*rule WTCSel*) **apply** *auto* **done**
moreover from *wfE* **have** $InvokeC\ E\ l \in wf\text{-}ctx$ **by** (*rule WFInvoke*)
moreover note *wtr rr g*

```
    ultimately show ?thesis by blast
  qed
next
  fix Γ σ e l m msigs
  assume wte: Γ ⊢ e : ObjT msigs
    and IH: Γ = empty ⟶
        Values e ∨
        (∃σ E r.
            e = E[r] ∧
            Γ ⊢ E : σ⇒ObjT msigs ∧ E ∈ wf-ctx ∧ Γ ⊢ r : σ ∧ (redex r ∨ bad-cast r))
    and wtm: Γ ⊢ m : σ in ObjT msigs
    and lm: lookup-sig msigs l = Some σ
    and g: Γ = empty
  show Values (Update e m) ∨
          (∃σ E r. (Update e m) = E[r] ∧
            Γ ⊢ E : σ⇒ObjT msigs ∧ E ∈ wf-ctx ∧ Γ ⊢ r : σ ∧ (redex r ∨ bad-cast r))
  proof (cases Values e)
    assume ve: Values e
    have a: Update e m = Hole[Update e m] by simp
    have wth: Γ ⊢ Hole : ObjT msigs ⇒ ObjT msigs by (rule WTHole)
    have wfh: Hole ∈ wf-ctx by (rule WFHole)
    from wte wtm lm have wtr: Γ ⊢ Update e m : ObjT msigs by (rule wte-upd)
    from wte ve g have x: (∃ ms τ. e = Obj ms τ) ∨ (∃ ms τ rr tt. e = (Obj ms
τ)⟨ObjT rr ⇒ ObjT tt⟩)
      apply simp apply (rule canonical-form-obj) by auto
    moreover { assume x: ∃ ms τ. e = Obj ms τ
      from x obtain ms τ' where e: e = Obj ms τ' by blast
      from e have red: Update e m ⟶ Obj (replace ms m) τ' apply simp by (rule
Upd)
      from red e have r: redex (Update e m) using redex-def by blast
      from a wth wfh wtr r have ?thesis by blast
    } moreover { assume x: ∃ ms τ rr tt. e = (Obj ms τ)⟨ObjT rr ⇒ ObjT tt⟩
      have ?thesis sorry
    } ultimately show ?thesis by blast
  next
    assume ve: ¬ Values e
    from ve IH g obtain σ' E r where e: e = E[r]
      and wtE: Γ ⊢ E : σ'⇒ObjT msigs and wfE: E ∈ wf-ctx
      and wtr: Γ ⊢ r : σ' and rr: (redex r ∨ bad-cast r) by blast
    from e have Update e m = (UpdateC E m)[r] by simp
    moreover from wtE wtm g lm have empty ⊢ UpdateC E m : σ' ⇒ ObjT msigs
      apply simp apply (rule WTCUpd) apply auto done
    moreover from wfE have UpdateC E m ∈ wf-ctx by (rule WFUpdate)
    moreover note wtr rr g
    ultimately show ?thesis by blast
  qed
qed

lemma welltyped-decomposition:
  empty ⊢ e : τ ⟹ Values e
```

$\lor$ ($\exists$ σ E r. $e = E[r]$ $\land$ *empty* $\vdash E : \sigma \Rightarrow \tau$ $\land$ $E \in$ *wf-ctx*
$\land$ *empty* $\vdash r : \sigma$ $\land$ (*redex* r $\lor$ *bad-cast* r))
using *welltyped-decomposition-impl* **apply** *blast* **done**

9.7 Subterm Typing

lemma *subterm-typing-impl*:
($\Gamma \vdash e : \tau \longrightarrow (\forall\ E\ r.\ e = E[r] \longrightarrow (\exists\ \sigma.\ \Gamma \vdash E : \sigma \Rightarrow \tau \land \Gamma \vdash r : \sigma))$)
$\land$ ($\Gamma \vdash m : s$ *in* $A \longrightarrow$ *True*)
$\land$ ($\Gamma \vdash ms :: ss$ *in* $A' \longrightarrow$ *True*)
apply (*induct rule*: *wte-wtm-wtms.induct*)
apply *clarify*
apply (*rule-tac* $x{=}\tau$ **in** *exI*)
apply (*case-tac* E)
using *wte-var WTHole* **apply** *force*
apply *simp* **apply** *simp* **apply** *simp* **apply** *simp* **apply** *simp*
apply *clarify*
apply (*rule-tac* $x{=}$*TypeOf* c **in** *exI*)
apply (*case-tac* E) **using** *wte-const WTHole* **apply** *force*
apply *simp* **apply** *simp* **apply** *simp* **apply** *simp* **apply** *simp*
apply *clarify*
apply (*case-tac* E)
apply (*rule-tac* $x{=}\sigma{\to}\tau$ **in** *exI*)
apply *simp* **using** *wte-abs WTHole* **apply** *force*
apply *simp* **apply** *simp* **apply** *simp* **apply** *simp* **apply** *simp*
apply *clarify*
apply (*case-tac* E)
apply (*rule-tac* $x{=}\tau$ **in** *exI*) **using** *wte-app WTHole* **apply** *force*
apply (*erule-tac* $x{=}$*ctx* **in** *allE*)
apply (*erule-tac* $x{=}$*ctx* **in** *allE*)
apply (*erule-tac* $x{=}r$ **in** *allE*)
apply (*erule-tac* $x{=}r$ **in** *allE*)
apply *simp* **using** *WTAppL* **apply** *blast*
apply (*erule-tac* $x{=}$*ctx* **in** *allE*)
apply (*erule-tac* $x{=}$*ctx* **in** *allE*)
apply (*erule-tac* $x{=}r$ **in** *allE*)
apply (*erule-tac* $x{=}r$ **in** *allE*)
apply *simp* **using** *WTAppR* **apply** *blast* **apply** *simp* **apply** *simp* **apply** *simp*
apply *clarify*
apply (*erule-tac* $x{=}E$ **in** *allE*)
apply (*erule-tac* $x{=}r$ **in** *allE*)
apply *simp*
apply (*erule exE*) **apply** *clarify*
apply (*rule-tac* $x{=}\sigma'$ **in** *exI*) **apply** *clarify*
apply (*rule WTCSub*) **apply** *simp* **apply** *simp*
apply *clarify*
apply (*case-tac* E)
apply (*rule-tac* $x{=}\tau$ **in** *exI*) **using** *wte-cast WTHole* **apply** *force*
apply *simp* **apply** *simp* **apply** *simp* **apply** *simp*
apply (*erule-tac* $x{=}$*ctx* **in** *allE*)

apply (*erule-tac x=r* **in** *allE*)
apply *simp* **using** *WTCCast* **apply** *blast*
apply *clarify*
apply (*case-tac E*)
apply (*rule-tac x=*τ **in** *exI*) **using** *wte-sel WTHole* **apply** *force*
apply *simp* **apply** *simp*
apply (*erule-tac x=ctx* **in** *allE*)
apply (*erule-tac x=r* **in** *allE*)
apply *simp* **apply** (*erule exE*) **apply** *clarify*
apply (*rule-tac x=*σ **in** *exI*) **apply** (*rule conjI*)
apply (*rule WTCSel*) **apply** *simp* **apply** *simp* **apply** *simp*
apply *simp* **apply** *simp*
apply *clarify*
apply (*case-tac E*)
apply (*rule-tac x=ObjT ss* **in** *exI*)
apply (*rule conjI*)
apply *simp* **apply** (*rule WTHole*)
apply *simp* **apply** *clarify* **apply** (*rule wte-upd*) **apply** *simp*
apply *simp* **apply** *simp* **apply** *simp* **apply** *simp* **apply** *simp*
apply (*erule-tac x=ctx* **in** *allE*)
apply (*erule-tac x=r* **in** *allE*)
apply *simp* **apply** (*erule exE*) **apply** *clarify*
apply (*rule-tac x=*σ **in** *exI*) **apply** (*rule conjI*)
apply (*rule WTCUpd*) **apply** *simp* **apply** *simp* **apply** *simp*
apply *simp* **apply** *simp*
apply *clarify*
apply (*case-tac E*)
apply (*rule-tac x=ObjT ss* **in** *exI*)
apply *simp* **using** *wte-obj WTHole* **apply** *force*
apply *simp* **apply** *simp* **apply** *simp* **apply** *simp* **apply** *simp*
apply *simp+*
done

lemma *subterm-typing*:
$\Gamma \vdash E[r] : \tau \Longrightarrow \exists\ \sigma.\ \Gamma \vdash E : \sigma \Rightarrow \tau \wedge \Gamma \vdash r : \sigma$
using *subterm-typing-impl* **by** *simp*

lemma *fill-ctx-welltyped*[*rule-format*]:
$\Gamma \vdash E : \sigma \Rightarrow \tau \Longrightarrow \forall\ r.\ \Gamma \vdash r : \sigma \longrightarrow \Gamma \vdash$ *fill E r* $: \tau$
apply (*induct rule*: *welltyped-ctx.induct*)
apply *simp*
using *wte-app* **apply** *force*
using *wte-app* **apply** *force*
using *wte-sub* **apply** *blast*
apply *clarify* **apply** *simp* **apply** (*rule wte-sel*) **apply** *blast* **apply** *blast*
apply *clarify* **apply** *simp* **apply** (*rule wte-upd*) **apply** *blast* **apply** *blast*
apply *blast*
using *wte-cast* **apply** *force*
done

9.8 Progress and Preservation

```
constdefs BadCast :: expr ⇒ bool
  BadCast e ≡ ∃ (E::ctx) r. e = E[r] ∧ bad-cast r

lemma progress:
  assumes wte: empty ⊢ e : τ
  shows Values e ∨ (∃ e'. e ⟼ e') ∨ BadCast e
proof −
  show ?thesis
  proof (cases Values e)
    assume Values e thus ?thesis by simp
  next assume ¬ Values e
    with wte have x: ∃ σ E r. e = E[r] ∧ empty ⊢ E : σ ⇒ τ ∧ E ∈ wf-ctx
        ∧ empty ⊢ r : σ  ∧ (redex r ∨ bad-cast r)
      using welltyped-decomposition[of e τ] by simp
    from x obtain σ E r where eE: e = E[r] and wtc: empty ⊢ E : σ ⇒ τ
      and wfE: E ∈ wf-ctx and wtr: empty ⊢ r : σ
      and rrb: redex r ∨ bad-cast r by blast
    { assume rr: redex r
      from rr obtain r' where red: r −→ r' using redex-def by blast
      from wfE red have E[r] ⟼ E[r'] by (rule Step)
      with eE have ?thesis by blast
    } moreover { assume b: bad-cast r
      with eE have BadCast e apply (simp add: BadCast-def) by auto
      hence ?thesis by blast
    } moreover note rrb
    ultimately show ?thesis by blast
  qed
qed

lemma preservation:
  assumes s: e ⟼ e'
  and wte: empty ⊢ e : τ
  shows empty ⊢ e' : τ
using s
proof (cases rule: eval-step.cases)
  fix E r r'
  assume a: (e, e') = (E[r], E[r'])
    and wfE: E ∈ wf-ctx
    and rr: r −→ r'
  from a wte obtain σ where wtc: empty ⊢ E : σ ⇒ τ
    and wtr: empty ⊢ r : σ using subterm-typing by blast
  from wtr rr
  have wtrp: empty ⊢ r' : σ using subject-reduction by blast
  from wtc wtrp have empty ⊢ fill E r' : τ by (rule fill-ctx-welltyped)
  with a show ?thesis by simp
qed
```

9.9 The Main Theorem

constdefs *finished* :: *expr* ⇒ *bool*
finished $e \equiv \neg(\exists\ e'.\ e \longmapsto e')$

syntax *eval-step-rtrancl* :: *expr* ⇒ *expr* ⇒ *bool* (**infixl** $\longmapsto^*$ *51*)
translations $e \longmapsto^* e' == (e,e') \in$ *eval-step**

lemma *type-safety-fobj*:
assumes *et*: *empty* ⊢ $e : \tau$
and *ee*: $e \longmapsto^* e'$
shows *empty* ⊢ $e' : \tau \wedge$ (*Values* $e' \vee$ *BadCast* $e' \vee \neg$ (*finished* e'))
using *ee et*
proof (*induct rule*: *rtrancl.induct*)
fix *a* **assume** *wta*: *empty* ⊢ $a : \tau$
from *wta* **have** *Values* $a \vee (\exists\ e'.\ a \longmapsto e') \vee$ *BadCast* a **by** (*rule progress*)
with *wta* **show** *empty* ⊢ $a : \tau \wedge$ (*Values* $a \vee$ *BadCast* $a \vee \neg$ (*finished* a))
using *finished-def* **by** *auto*
next
fix *a b c*
assume *IH*: *empty* ⊢ $a : \tau \Longrightarrow$ *empty* ⊢ $b : \tau \wedge$ (*Values* $b \vee$ *BadCast* $b \vee \neg$(*finished* b))
and *bc*: $b \longmapsto c$ **and** *wta*: *empty* ⊢ $a : \tau$
from *wta IH* **have** *wtb*: *empty* ⊢ $b : \tau$ **by** *simp*
from *bc wtb* **have** *wtc*: *empty* ⊢ $c : \tau$ **by** (*rule preservation*)
from *wtc* **have** *Values* $c \vee (\exists\ e'.\ c \longmapsto e') \vee$ *BadCast* c **by** (*rule progress*)
with *wtc* **show** *empty* ⊢ $c : \tau \wedge$ (*Values* $c \vee$ *BadCast* $c \vee \neg$ (*finished* c))
using *finished-def* **by** *auto*
qed

lemma *compilation-total-impl*:
$(\Gamma \vdash_G e : \tau \longrightarrow (\exists\ e'.\ \Gamma \vdash e \Rightarrow e' : \tau))$
$\wedge\ (\Gamma \vdash_G m : s\ in\ \tau \longrightarrow (\exists\ m'.\ \Gamma \vdash m \Rightarrow m' : s\ in\ \tau))$
$\wedge\ (\Gamma \vdash_G ms :: ss\ in\ \tau \longrightarrow (\exists\ ms'.\ \Gamma \vdash ms \Rightarrow ms' :: ss\ in\ \tau))$
sorry

lemma *compilation-total*:
$\Gamma \vdash_G e : \tau \Longrightarrow \exists\ e'.\ \Gamma \vdash e \Rightarrow e' : \tau$
using *compilation-total-impl* **by** *blast*

theorem *type-safety*:
assumes *c*: *empty* ⊢ $e \Rightarrow e' : \tau$
and *ee*: $e' \longmapsto^* e''$
and *te*: *finished* e''
shows (*Values* $e'' \vee$ *BadCast* e'') $\wedge$ *empty* ⊢ $e'' : \tau$
proof −
from *c* **have** *et*: *empty* ⊢ $e' : \tau$ **by** (*rule compilation-sound*)
from *et ee te* **show** *?thesis* **using** *type-safety-fobj* **by** *blast*
qed

References

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.

[2] A. Charguéraud, B. C. Pierce, and S. Weirich. Proof engineering: Practical techniques for mechanized metatheory, Sept. 2006. Submitted for publication.

[3] A. D. Gordon. A mechanisation of name-carrying syntax up to alpha-conversion. In *HUG '93: Proceedings of the 6th International Workshop on Higher Order Logic Theorem Proving and its Applications*, pages 413–425, London, UK, 1994. Springer-Verlag.

[4] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[5] R. Pollack. Closure under alpha-conversion. In *TYPES '93: Proceedings of the international workshop on Types for proofs and programs*, pages 313–332, Secaucus, NJ, USA, 1994. Springer-Verlag New York, Inc.

[6] J. G. Siek and W. Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, September 2006.