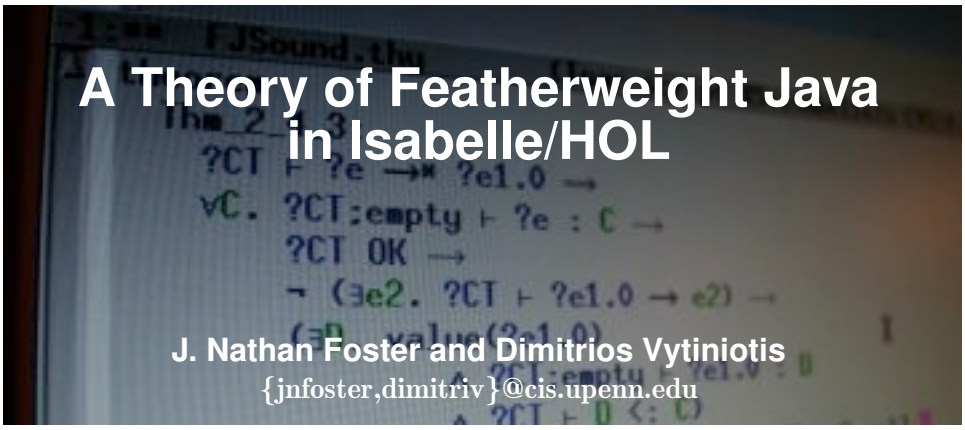

A Theory of Featherweight Java in Isabelle/HOL

J. Nathan Foster and Dimitrios Vytiniotis
{jnfoster,dimitriv}@cis.upenn.edu

Abstract

We formalize the type system, small-step operational semantics, and type soundness proof for Featherweight Java [1], a simple object calculus, in Isabelle/HOL [2].

Contents

1 `FJDefs`: Basic Definitions **2**
1.1 Syntax . . . 2
1.1.1 Type definitions . . . 2
1.1.2 Constants . . . 3
1.1.3 Expressions . . . 3
1.1.4 Methods . . . 3
1.1.5 Constructors . . . 3
1.1.6 Classes . . . 3
1.1.7 Class Tables . . . 4
1.2 Sub-expression Relation . . . 4
1.3 Values . . . 4
1.4 Substitution . . . 4
1.5 Lookup . . . 5
1.6 Variable Definition Accessors . . . 5
1.7 Subtyping Relation . . . 5
1.8 `fields` Relation . . . 6
1.9 `mtype` Relation . . . 6
1.10 `mbody` Relation . . . 7
1.11 Typing Relation . . . 7
1.12 Method Typing Relation . . . 10
1.13 Class Typing Relation . . . 11
1.14 Class Table Typing Relation . . . 11
1.15 Evaluation Relation . . . 11

2 FJAux: Auxiliary Lemmas **12**
2.1 Non-FJ Lemmas . . . 12
2.1.1 Lists . . . 12
2.1.2 Maps . . . 13
2.2 FJ Lemmas . . . 14
2.2.1 Substitution . . . 14
2.2.2 Lookup . . . 15
2.2.3 Functional . . . 17
2.2.4 Subtyping and Typing . . . 18
2.2.5 Sub-Expressions . . . 22

3 FJSound: Type Soundness **22**
3.1 Method Type and Body Connection . . . 22
3.2 Method Types and Field Declarations of Subtypes . . . 23
3.3 Substitution Lemma . . . 24
3.4 Weakening Lemma . . . 29
3.5 Method Body Typing Lemma . . . 30
3.6 Subject Reduction Theorem . . . 30
3.7 Multi-Step Subject Reduction Theorem . . . 34
3.8 Progress . . . 35
3.9 Type Soundness Theorem . . . 40

4 Executing FeatherweightJava programs **42**
4.1 A simple example . . . 44

1 FJDefs: Basic Definitions

theory *FJDefs*
imports *Main*
begin

1.1 Syntax

We use a named representation for terms: variables, method names, and class names, are all represented as `nats`. We use the finite maps defined in `Map.thy` to represent typing contexts and the static class table. This section defines the representations of each syntactic category (expressions, methods, constructors, classes, class tables) and defines several constants (`Object` and `this`).

1.1.1 Type definitions

type-synonym *varName* = *nat*
type-synonym *methodName* = *nat*
type-synonym *className* = *nat*

record *varDef* =
  vdName :: *varName*
  vdType :: *className*
type-synonym *varCtx* = *varName* ⇀ *className*

1.1.2 Constants

definition
  Object :: *className* **where**
  Object = *0*

definition
  this :: *varName* **where**
  this == *0*

1.1.3 Expressions

datatype *exp* =
    Var varName
  | *FieldProj exp varName*
  | *MethodInvk exp methodName exp list*
  | *New className exp list*
  | *Cast className exp*

1.1.4 Methods

record *methodDef* =
  mReturn :: *className*
  mName :: *methodName*
  mParams :: *varDef list*
  mBody :: *exp*

1.1.5 Constructors

record *constructorDef* =
  kName :: *className*
  kParams :: *varDef list*
  kSuper :: *varName list*
  kInits :: *varName list*

1.1.6 Classes

record *classDef* =
  cName :: *className*
  cSuper :: *className*
  cFields :: *varDef list*
  cConstructor :: *constructorDef*
  cMethods :: *methodDef list*

1.1.7 Class Tables

type-synonym $classTable = className \rightharpoonup classDef$

1.2 Sub-expression Relation

The sub-expression relation, written $t \in subexprs(s)$, is defined as the reflexive and transitive closure of the immediate subexpression relation.

inductive-set
  $isubexprs :: (exp * exp)\ set$
  and $isubexprs' :: [exp,exp] \Rightarrow bool\ \ (\langle\text{-} \in isubexprs'(\text{-}')\rangle\ [80,80]\ 80)$
where
  $e' \in isubexprs(e) \equiv (e',e) \in isubexprs$
| *se-field* : $e \in isubexprs(FieldProj\ e\ fi)$
| *se-invkrecv* : $e \in isubexprs(MethodInvk\ e\ m\ es)$
| *se-invkarg* : $[\![\ ei \in set\ es\ ]\!] \Longrightarrow ei \in isubexprs(MethodInvk\ e\ m\ es)$
| *se-newarg* : $[\![\ ei \in set\ es\ ]\!] \Longrightarrow ei \in isubexprs(New\ C\ es)$
| *se-cast* : $e \in isubexprs(Cast\ C\ e)$

abbreviation
  $subexprs :: [exp,exp] \Rightarrow bool\ \ (\langle\text{-} \in subexprs'(\text{-}')\rangle\ [80,80]\ 80)$ **where**
  $e' \in subexprs(e) \equiv (e',e) \in isubexprs\hat{}*$

1.3 Values

A *value* is an expression of the form `new C`$(overline vs)$, where $\overline{vs}$ is a list of values.

inductive
  $vals :: [exp\ list] \Rightarrow bool\ (\langle vals'(\text{-}')\rangle\ [80]\ 80)$
  and $val :: [exp] \Rightarrow bool\ (\langle val'(\text{-}')\rangle\ [80]\ 80)$
where
  vals-nil : $vals([])$
| *vals-cons* : $[\![\ val(vh);\ vals(vt)\ ]\!] \Longrightarrow vals((vh\ \#\ vt))$
| *val* : $[\![\ vals(vs)\ ]\!] \Longrightarrow val(New\ C\ vs)$

1.4 Substitution

The substitutions of a list of expressions ds for a list of variables xs in another expression e or a list of expressions es are defined in the obvious way, and written $(ds/xs)e$ and $[ds/xs]es$ respecitvely.

primrec $substs :: (varName \rightharpoonup exp) \Rightarrow exp \Rightarrow exp$
  and $subst\text{-}list1 :: (varName \rightharpoonup exp) \Rightarrow exp\ list \Rightarrow exp\ list$
  and $subst\text{-}list2 :: (varName \rightharpoonup exp) \Rightarrow exp\ list \Rightarrow exp\ list$ **where**
  $substs\ \sigma\ (Var\ x) = (case\ (\sigma(x))\ of\ None \Rightarrow (Var\ x) \mid Some\ p \Rightarrow p)$
| $substs\ \sigma\ (FieldProj\ e\ f) = FieldProj\ (substs\ \sigma\ e)\ f$
| $substs\ \sigma\ (MethodInvk\ e\ m\ es) = MethodInvk\ (substs\ \sigma\ e)\ m\ (subst\text{-}list1\ \sigma\ es)$
| $substs\ \sigma\ (New\ C\ es) = New\ C\ (subst\text{-}list2\ \sigma\ es)$

```
| substs σ (Cast C e) =          Cast C (substs σ e)
| subst-list1 σ [] = []
| subst-list1 σ (h # t) = (substs σ h) # (subst-list1 σ t)
| subst-list2 σ [] = []
| subst-list2 σ (h # t) = (substs σ h) # (subst-list2 σ t)
```

abbreviation
substs-syn :: [*exp list*] ⇒ [*varName list*] ⇒ [*exp*] ⇒ *exp*
(‹'(-'/-')-› [80,80,80] 80) **where**
(*ds*/*xs*)*e* ≡ *substs* (*map-upds Map.empty xs ds*) *e*

abbreviation
subst-list-syn :: [*exp list*] ⇒ [*varName list*] ⇒ [*exp list*] ⇒ *exp list*
(‹'[-'/-']-› [80,80,80] 80) **where**
[*ds*/*xs*]*es* ≡ *map* (*substs* (*map-upds Map.empty xs ds*)) *es*

1.5 Lookup

The fuction *lookup f l* function returns an option containing the first element of *l* satisfying *f*, or `None` if no such element exists

primrec *lookup* :: 'a *list* ⇒ ('a ⇒ *bool*) ⇒ 'a *option*
where
lookup [] *P* = *None*
| *lookup* (*h*#*t*) *P* = (*if P h then Some h else lookup t P*)

primrec *lookup2* :: 'a *list* ⇒ 'b *list* ⇒ ('a ⇒ *bool*) ⇒ 'b *option*
where
lookup2 [] *l2 P* = *None*
| *lookup2* (*h1*#*t1*) *l2 P* = (*if P h1 then Some*(*hd l2*) *else lookup2 t1* (*tl l2*) *P*)

1.6 Variable Definition Accessors

This section contains several helper functions for reading off the names and types of variable definitions (e.g., in field and method parameter declarations).

definition
varDefs-names :: *varDef list* ⇒ *varName list* **where**
varDefs-names = *map vdName*

definition
varDefs-types :: *varDef list* ⇒ *className list* **where**
varDefs-types = *map vdType*

1.7 Subtyping Relation

The subtyping relation, written $CT \vdash C{<:}D$ is just the reflexive and transitive closure of the immediate subclass relation. (For the sake of simplicity,

we define subtyping directly instead of using the reflexive and transitive closure operator.) The subtyping relation is extended to lists of classes, written $CT \vdash +Cs<:Ds$.

inductive
subtyping :: [*classTable*, *className*, *className*] ⇒ *bool* (‹- ⊢ - <: -› [80,80,80] 80)
where
s-refl : $CT \vdash C <: C$
| *s-trans* : ⟦ $CT \vdash C <: D$; $CT \vdash D <: E$ ⟧ ⟹ $CT \vdash C <: E$
| *s-super* : ⟦ $CT(C) = Some(CDef)$; *cSuper CDef* = *D* ⟧ ⟹ $CT \vdash C <: D$

abbreviation
neg-subtyping :: [*classTable*, *className*, *className*] ⇒ *bool* (‹- ⊢ - ¬<: -› [80,80,80] 80)
where $CT \vdash S \neg<: T \equiv \neg\ CT \vdash S <: T$

inductive
subtypings :: [*classTable*, *className list*, *className list*] ⇒ *bool* (‹- ⊢+ - <: -› [80,80,80] 80)
where
ss-nil : $CT \vdash+ [] <: []$
| *ss-cons* : ⟦ $CT \vdash C0 <: D0$; $CT \vdash+ Cs <: Ds$ ⟧ ⟹ $CT \vdash+ (C0 \# Cs) <: (D0 \# Ds)$

1.8 `fields` Relation

The `fields` relation, written $\texttt{fields}(CT, C) = Cf$, relates Cf to C when Cf is the list of fields declared directly or indirectly (i.e., by a superclass) in C.

inductive
fields :: [*classTable*, *className*, *varDef list*] ⇒ *bool* (‹*fields*'(-,-') = -› [80,80,80] 80)
where
f-obj:
fields(*CT*,*Object*) = []
| *f-class*:
⟦ $CT(C) = Some(CDef)$; *cSuper CDef* = *D*; *cFields CDef* = *Cf*; *fields*(*CT*,*D*) = *Dg*; *DgCf* = *Dg* @ *Cf* ⟧
⟹ *fields*(*CT*,*C*) = *DgCf*

1.9 `mtype` Relation

The `mtype` relation, written $\texttt{mtype}(CT, m, C) = Cs \to C_0$ relates a class C, method name m, and the arrow type $Cs \to C_0$. It either returns the type of the declaration of m in C, if any such declaration exists, and otherwise returning the type of m from C's superclass.

inductive

mtype :: [*classTable*, *methodName*, *className*, *className list*, *className*] ⇒ *bool* (‹*mtype*′(-,-,-′) = - → -› [*80*,*80*,*80*,*80*] *80*)
where
mt-class:
⟦ *CT*(*C*) = *Some*(*CDef*);
lookup (*cMethods CDef*) (λ*md*.(*mName md* = *m*)) = *Some*(*mDef*);
varDefs-types (*mParams mDef*) = *Bs*;
mReturn mDef = *B* ⟧
⟹ *mtype*(*CT*,*m*,*C*) = *Bs* → *B*

| *mt-super*:
⟦ *CT*(*C*) = *Some* (*CDef*);
lookup (*cMethods CDef*) (λ*md*.(*mName md* = *m*)) = *None*;
cSuper CDef = *D*;
mtype(*CT*,*m*,*D*) = *Bs* → *B* ⟧
⟹ *mtype*(*CT*,*m*,*C*) = *Bs* → *B*

1.10 `mbody` Relation

The `mtype` relation, written $\texttt{mbody}(CT, m, C) = xs.e_0$ relates a class C, method name m, and the names of the parameters xs and the body of the method e_0. It either returns the parameter names and body of the declaration of m in C, if any such declaration exists, and otherwise the parameter names and body of m from C's superclass.

inductive
mbody :: [*classTable*, *methodName*, *className*, *varName list*, *exp*] ⇒ *bool* (‹*mbody*′(-,-,-′) = - . -› [*80*,*80*,*80*,*80*] *80*)
where
mb-class:
⟦ *CT*(*C*) = *Some*(*CDef*);
lookup (*cMethods CDef*) (λ*md*.(*mName md* = *m*)) = *Some*(*mDef*);
varDefs-names (*mParams mDef*) = *xs*;
mBody mDef = *e* ⟧
⟹ *mbody*(*CT*,*m*,*C*) = *xs* . *e*

| *mb-super*:
⟦ *CT*(*C*) = *Some*(*CDef*);
lookup (*cMethods CDef*) (λ*md*.(*mName md* = *m*)) = *None*;
cSuper CDef = *D*;
mbody(*CT*,*m*,*D*) = *xs* . *e* ⟧
⟹ *mbody*(*CT*,*m*,*C*) = *xs* . *e*

1.11 Typing Relation

The typing relation, written $CT; \Gamma \vdash e : C$ relates an expression e to its type C, under the typing context Γ. The multi-typing relation, written $CT; \Gamma \vdash +es : Cs$ relates lists of expressions to lists of types.

inductive

typings :: [*classTable*, *varCtx*, *exp list*, *className list*] ⇒ *bool* (‹-;- ⊢+ - : ->
[80,80,80,80] 80)
and *typing* :: [*classTable*, *varCtx*, *exp*, *className*] ⇒ *bool* (‹-;- ⊢ - : -› [80,80,80,80]
80)
where
ts-nil : $CT;\Gamma \vdash+ [] : []$

| *ts-cons* :
$[\![\ CT;\Gamma \vdash e0 : C0;\ CT;\Gamma \vdash+ es : Cs\ ]\!]$
$\Longrightarrow CT;\Gamma \vdash+ (e0\ \#\ es) : (C0\ \#\ Cs)$

| *t-var* :
$[\![\ \Gamma(x) = Some\ C\ ]\!] \Longrightarrow CT;\Gamma \vdash (Var\ x) : C$

| *t-field* :
$[\![\ CT;\Gamma \vdash e0 : C0;$
$fields(CT,C0) = Cf;$
$lookup\ Cf\ (\lambda fd.(vdName\ fd = fi)) = Some(fDef);$
$vdType\ fDef = Ci\ ]\!]$
$\Longrightarrow CT;\Gamma \vdash FieldProj\ e0\ fi : Ci$

| *t-invk* :
$[\![\ CT;\Gamma \vdash e0 : C0;$
$mtype(CT,m,C0) = Ds \to C;$
$CT;\Gamma \vdash+ es : Cs;$
$CT \vdash+ Cs <: Ds;$
$length\ es = length\ Ds\ ]\!]$
$\Longrightarrow CT;\Gamma \vdash MethodInvk\ e0\ m\ es : C$

| *t-new* :
$[\![\ fields(CT,C) = Df;$
$length\ es = length\ Df;$
$varDefs\text{-}types\ Df = Ds;$
$CT;\Gamma \vdash+ es : Cs;$
$CT \vdash+ Cs <: Ds\ ]\!]$
$\Longrightarrow CT;\Gamma \vdash New\ C\ es : C$

| *t-ucast* :
$[\![\ CT;\Gamma \vdash e0 : D;$
$CT \vdash D <: C\ ]\!]$
$\Longrightarrow CT;\Gamma \vdash Cast\ C\ e0 : C$

| *t-dcast* :
$[\![\ CT;\Gamma \vdash e0 : D;$
$CT \vdash C <: D;\ C \neq D\ ]\!]$
$\Longrightarrow CT;\Gamma \vdash Cast\ C\ e0 : C$

| *t-scast* :
$[\![\ CT;\Gamma \vdash e0 : D;$

$CT \vdash C \neg<: D;$
$CT \vdash D \neg<: C \rrbracket$
$\Longrightarrow CT;\Gamma \vdash Cast\ C\ e0 : C$

We occasionally find the following induction principle, which only mentions the typing of a single expression, more useful than the mutual induction principle generated by Isabelle, which mentions the typings of single expressions and of lists of expressions.

lemma *typing-induct*:
assumes $CT;\Gamma \vdash e : C$ (**is** *?T*)
and $\bigwedge C\ CT\ \Gamma\ x.\ \Gamma\ x = Some\ C \Longrightarrow P\ CT\ \Gamma\ (Var\ x)\ C$
and $\bigwedge C0\ CT\ Cf\ Ci\ \Gamma\ e0\ fDef\ fi.\ [\![CT;\Gamma \vdash e0 : C0;\ P\ CT\ \Gamma\ e0\ C0;\ fields(CT,C0) = Cf;\ lookup\ Cf\ (\lambda fd.\ vdName\ fd = fi) = Some\ fDef;\ vdType\ fDef = Ci]\!] \Longrightarrow P\ CT\ \Gamma\ (FieldProj\ e0\ fi)\ Ci$
and $\bigwedge C\ C0\ CT\ Cs\ Ds\ \Gamma\ e0\ es\ m.\ [\![CT;\Gamma \vdash e0 : C0;\ P\ CT\ \Gamma\ e0\ C0;\ mtype(CT,m,C0) = Ds \to C;\ CT;\Gamma \vdash+ es : Cs;\ \bigwedge i\ .\ [\![\ i < length\ es\ ]\!] \Longrightarrow\ P\ CT\ \Gamma\ (es!i)\ (Cs!i);\ CT \vdash+ Cs <: Ds;\ length\ es = length\ Ds]\!] \Longrightarrow P\ CT\ \Gamma\ (MethodInvk\ e0\ m\ es)\ C$
and $\bigwedge C\ CT\ Cs\ Df\ Ds\ \Gamma\ es.\ [\![fields(CT,C) = Df;\ length\ es = length\ Df;\ varDefs\text{-}types\ Df = Ds;\ CT;\Gamma \vdash+ es : Cs;\ \bigwedge i.\ [\![\ i < length\ es\ ]\!] \Longrightarrow P\ CT\ \Gamma\ (es!i)\ (Cs!i);\ CT \vdash+ Cs <: Ds]\!] \Longrightarrow P\ CT\ \Gamma\ (New\ C\ es)\ C$
and $\bigwedge C\ CT\ D\ \Gamma\ e0.\ [\![CT;\Gamma \vdash e0 : D;\ P\ CT\ \Gamma\ e0\ D;\ CT \vdash D <: C]\!] \Longrightarrow P\ CT\ \Gamma\ (Cast\ C\ e0)\ C$
and $\bigwedge C\ CT\ D\ \Gamma\ e0.\ [\![CT;\Gamma \vdash e0 : D;\ P\ CT\ \Gamma\ e0\ D;\ CT \vdash C <: D;\ C \neq D]\!] \Longrightarrow P\ CT\ \Gamma\ (Cast\ C\ e0)\ C$
and $\bigwedge C\ CT\ D\ \Gamma\ e0.\ [\![CT;\Gamma \vdash e0 : D;\ P\ CT\ \Gamma\ e0\ D;\ CT \vdash C \neg<: D;\ CT \vdash D \neg<: C]\!] \Longrightarrow P\ CT\ \Gamma\ (Cast\ C\ e0)\ C$
shows $P\ CT\ \Gamma\ e\ C$ (**is** *?P*)
proof −
fix *es Cs*
let $?IH = CT;\Gamma \vdash+ es : Cs \longrightarrow (\forall i < length\ es.\ \ P\ CT\ \Gamma\ (es!i)\ (Cs!i))$
have $?IH \wedge (?T \longrightarrow ?P)$
proof(*induct rule*:*typings-typing.induct*)
case (*ts-nil CT* Γ) **show** *?case* **by** *auto*
next
case (*ts-cons CT* Γ *e0 C0 es Cs*)
show *?case* **proof**
fix *i*
show $i < length\ (e0\#es) \longrightarrow P\ CT\ \Gamma\ ((e0\#es)!i)\ ((C0\#Cs)!i)$ **using** *ts-cons* **by**(*cases i*, *auto*)
qed
next
case *t-var* **then show** *?case* **using** *assms* **by** *auto*
next
case *t-field* **then show** *?case* **using** *assms* **by** *auto*
next
case *t-invk* **then show** *?case* **using** *assms* **by** *auto*
next
case *t-new* **then show** *?case* **using** *assms* **by** *auto*
next

case *t-ucast* **then show** *?case* **using** *assms* **by** *auto*
next
case *t-dcast* **then show** *?case* **using** *assms* **by** *auto*
next
case *t-scast* **then show** *?case* **using** *assms* **by** *auto*
qed
thus *?thesis* **using** *assms* **by** *auto*
qed

1.12 Method Typing Relation

A method definition md, declared in a class C, is well-typed, written $CT \vdash md$`OK IN` C if its body is well-typed and it has the same type (i.e., overrides) any method with the same name declared in the superclass of C.

inductive
method-typing :: [*classTable*, *methodDef*, *className*] ⇒ *bool* (‹- ⊢ - *OK IN* -› [*80*,*80*,*80*] *80*)
where
m-typing:
⟦ *CT*(*C*) = *Some*(*CDef*);
cName CDef = *C*;
cSuper CDef = *D*;
mName mDef = *m*;
lookup (*cMethods CDef*) (λ*md*.(*mName md* = *m*)) = *Some*(*mDef*);
mReturn mDef = *C0*; *mParams mDef* = *Cxs*; *mBody mDef* = *e0*;
varDefs-types Cxs = *Cs*;
varDefs-names Cxs = *xs*;
Γ = (*map-upds Map.empty xs Cs*)(*this* ↦ *C*);
CT;Γ ⊢ *e0* : *E0*;
CT ⊢ *E0* <: *C0*;
∀ *Ds D0*. (*mtype*(*CT*,*m*,*D*) = *Ds* → *D0*) ⟶ (*Cs*=*Ds* ∧ *C0*=*D0*) ⟧
⟹ *CT* ⊢ *mDef OK IN C*

inductive
method-typings :: [*classTable*, *methodDef list*, *className*] ⇒ *bool* (‹- ⊢+ - *OK IN* -› [*80*,*80*,*80*] *80*)
where
ms-nil :
CT ⊢+ [] *OK IN C*

| *ms-cons* :
⟦ *CT* ⊢ *m OK IN C*;
CT ⊢+ *ms OK IN C* ⟧
⟹ *CT* ⊢+ (*m* # *ms*) *OK IN C*

1.13 Class Typing Relation

A class definition cd is well-typed, written $CT \vdash cd\texttt{OK}$ if its constructor initializes each field, and all of its methods are well-typed.

inductive
class-typing :: [*classTable*, *classDef*] ⇒ *bool* (‹- ⊢ - *OK*› [*80*,*80*] *80*)
where
t-class: ⟦ *cName CDef* = *C*;
cSuper CDef = *D*;
cConstructor CDef = *KDef*;
cMethods CDef = *M*;
kName KDef = *C*;
kParams KDef = (*Dg*@*Cf*);
kSuper KDef = *varDefs-names Dg*;
kInits KDef = *varDefs-names Cf*;
fields(*CT*,*D*) = *Dg*;
CT ⊢+ *M OK IN C* ⟧
⟹ *CT* ⊢ *CDef OK*

1.14 Class Table Typing Relation

A class table is well-typed, written CT `OK` if for every class name C, the class definition mapped to by CT is is well-typed and has name C.

inductive
ct-typing :: *classTable* ⇒ *bool* (‹- *OK*› *80*)
where
ct-all-ok:
⟦ *Object* ∉ *dom*(*CT*);
∀ *C CDef*. *CT*(*C*) = *Some*(*CDef*) ⟶ (*CT* ⊢ *CDef OK*) ∧ (*cName CDef* = *C*) ⟧
⟹ *CT OK*

1.15 Evaluation Relation

The single-step and multi-step evaluation relations are written $CT \vdash e \to e'$ and $CT \vdash e \to^* e'$ respectively.

inductive
reduction :: [*classTable*, *exp*, *exp*] ⇒ *bool* (‹- ⊢ - → -› [*80*,*80*,*80*] *80*)
where

r-field:
⟦ *fields*(*CT*,*C*) = *Cf*;
lookup2 Cf es (λ*fd*.(*vdName fd* = *fi*)) = *Some*(*ei*) ⟧
⟹ *CT* ⊢ *FieldProj* (*New C es*) *fi* → *ei*

| *r-invk*:
⟦ *mbody*(*CT*,*m*,*C*) = *xs* . *e0*;

$\quad substs\ ((map\text{-}upds\ Map.empty\ xs\ ds)(this \mapsto (New\ C\ es)))\ e0 = e0' \rrbracket$
$\Longrightarrow CT \vdash MethodInvk\ (New\ C\ es)\ m\ ds \to e0'$

| *r-cast*:
$\llbracket\ CT \vdash C <: D\ \rrbracket$
$\Longrightarrow CT \vdash Cast\ D\ (New\ C\ es) \to New\ C\ es$

| *rc-field*:
$\llbracket\ CT \vdash e0 \to e0'\ \rrbracket$
$\Longrightarrow CT \vdash FieldProj\ e0\ f \to FieldProj\ e0'\ f$

| *rc-invk-recv*:
$\llbracket\ CT \vdash e0 \to e0'\ \rrbracket$
$\Longrightarrow CT \vdash MethodInvk\ e0\ m\ es \to MethodInvk\ e0'\ m\ es$

| *rc-invk-arg*:
$\llbracket\ CT \vdash ei \to ei'\ \rrbracket$
$\Longrightarrow CT \vdash MethodInvk\ e0\ m\ (el@ei\#er) \to MethodInvk\ e0\ m\ (el@ei'\#er)$

| *rc-new-arg*:
$\llbracket\ CT \vdash ei \to ei'\ \rrbracket$
$\Longrightarrow CT \vdash New\ C\ (el@ei\#er) \to New\ C\ (el@ei'\#er)$

| *rc-cast*:
$\llbracket\ CT \vdash e0 \to e0'\ \rrbracket$
$\Longrightarrow CT \vdash Cast\ C\ e0 \to Cast\ C\ e0'$

inductive
$reductions :: [classTable,\ exp,\ exp] \Rightarrow bool\ (\langle\text{-} \vdash \text{-} \to* \text{-}\rangle\ [80,80,80]\ 80)$
where
rs-refl: $CT \vdash e \to* e$
| *rs-trans*: $\llbracket\ CT \vdash e \to e';\ CT \vdash e' \to* e''\ \rrbracket \Longrightarrow\ CT \vdash e \to* e''$

end

2 FJAux: Auxiliary Lemmas

theory *FJAux* **imports** *FJDefs*
begin

2.1 Non-FJ Lemmas

2.1.1 Lists

lemma *mem-ith*:
assumes $ei \in set\ es$
shows $\exists\ el\ er.\ es = el@ei\#er$
using *assms*
proof(*induct es*)

```
  case Nil thus ?case by auto
next
  case (Cons esh est)
  { assume esh = ei
    with Cons have ?case by blast
  } moreover {
    assume esh ≠ ei
    with Cons have ei ∈ set est by auto
    with Cons obtain el er where esh # est = (esh#el) @ (ei#er) by auto
    hence ?case by blast }
  ultimately show ?case by blast
qed

lemma ith-mem: ⋀i. ⟦ i < length es ⟧ ⟹ es!i ∈ set es
proof(induct es)
  case Nil thus ?case by auto
next
  case (Cons h t) thus ?case by(cases i, auto)
qed
```

2.1.2 Maps

```
lemma map-shuffle:
  assumes length xs = length ys
  shows [xs[↦]ys,x↦y] = [(xs@[x])[↦](ys@[y])]
  using assms
by (induct xs ys rule:list-induct2) (auto simp add:map-upds-append1)

lemma map-upds-index:
  assumes length xs = length As
  and [xs[↦]As]x = Some Ai
  shows ∃ i.(As!i = Ai)
        ∧ (i < length As)
        ∧ (∀ (Bs::'c list).((length Bs = length As)
                        ⟶ ([xs[↦]Bs] x = Some (Bs !i))))
  (is ∃ i. ?P i xs As
   is ∃ i.(?P1 i As) ∧ (?P2 i As) ∧ (∀ Bs::('c list).(?P3 i xs As Bs)))
  using assms
proof(induct xs As rule:list-induct2)
  assume [][↦][] x = Some Ai
  moreover have ¬[][↦][] x = Some Ai by auto
  ultimately show ∃ i. ?P i [] [] by contradiction
next
  fix xa xs y ys
  assume length-xs-ys: length xs = length ys
    and IH: [xs [↦] ys] x = Some Ai ⟹ ∃ i. ?P i xs ys
    and map-eq-Some: [xa # xs [↦] y # ys] x = Some Ai
  then have map-decomp: [xa#xs [↦] y#ys] = [xa↦y] ++ [xs[↦]ys] by fastforce
  show ∃ i. ?P i (xa#xs) (y # ys)
```

```
  proof(cases [xs[↦]ys]x)
    case(Some Ai')
    hence ([xa ↦y] ++ [xs[↦]ys]) x = Some Ai' by(rule map-add-find-right)
    hence P: [xs[↦]ys] x = Some Ai using map-eq-Some Some by simp
    from IH[OF P] obtain i where
      R1: ys ! i = Ai
      and R2: i < length ys
      and pre-r3: ∀(Bs::'c list). ?P3 i xs ys Bs by fastforce
    { fix Bs::'c list
      assume length-Bs: length Bs = length (y#ys)
      then obtain n where length (y#ys) = Suc n by auto
    with length-Bs obtain b bs where Bs-def: Bs = b#bs by (auto simp add:length-Suc-conv)
      with length-Bs have length ys = length bs by simp
        with pre-r3 have ([xa↦b] ++ [xs[↦]bs]) x = Some (bs!i) by(auto simp
only:map-add-find-right)
      with pre-r3 Bs-def length-Bs have ?P3 (i+1) (xa#xs) (y#ys) Bs by simp }
    with R1 R2 have ?P (i+1) (xa#xs) (y#ys) by auto
    thus ?thesis ..
  next
    case None
     with map-decomp map-eq-Some have [xa↦y] x = Some Ai by (auto simp
only:map-add-SomeD)
   hence ai-def: y = Ai and x-eq-xa:x=xa by (auto simp only:map-upd-Some-unfold)

    { fix Bs::'c list
      assume length-Bs: length Bs = length (y#ys)
      then obtain n where length (y#ys) = Suc n by auto
    with length-Bs obtain b bs where Bs-def: Bs = b#bs by (auto simp add:length-Suc-conv)
      with length-Bs have length ys = length bs by simp
      hence dom([xs[↦]ys]) = dom([xs[↦]bs]) by auto
      with None have [xs[↦]bs] x = None by (auto simp only:domIff)
       moreover from x-eq-xa have sing-map: [xa↦b] x = Some b by (auto simp
only:map-upd-Some-unfold)
    ultimately have ([xa↦b] ++ [xs[↦]bs]) x = Some b by (auto simp only:map-add-Some-iff)
      with Bs-def have ?P3 0 (xa#xs) (y#ys) Bs by simp }
    with ai-def have ?P 0 (xa#xs) (y#ys) by auto
    thus ?thesis ..
  qed
qed
```

2.2 FJ Lemmas

2.2.1 Substitution

```
lemma subst-list1-eq-map-substs :
  ∀σ. subst-list1 σ l = map (substs σ) l
   by (induct l, simp-all)

lemma subst-list2-eq-map-substs :
  ∀σ. subst-list2 σ l = map (substs σ) l
```

by (*induct l*, *simp-all*)

2.2.2 Lookup

lemma *lookup-functional*:
assumes *lookup l f = o1*
and *lookup l f = o2*
shows *o1 = o2*
using *assms* **by** (*induct l*) *auto*

lemma *lookup-true*:
lookup l f = Some r ⟹ f r
proof(*induct l*)
case *Nil* **thus** *?case* **by** *simp*
next
case(*Cons h t*) **thus** *?case* **by**(*cases f h*) (*auto simp add*:*lookup.simps*)
qed

lemma *lookup-hd*:
⟦ *length l > 0*; *f* (*l*!*0*) ⟧ ⟹ *lookup l f = Some* (*l*!*0*)
by (*induct l*) *auto*

lemma *lookup-split*: *lookup l f = None* ∨ (∃ *h*. *lookup l f = Some h*)
by (*induct l*) *simp-all*

lemma *lookup-index*:
assumes *lookup l1 f = Some e*
shows ⋀*l2*. ∃ *i* < (*length l1*). *e = l1*!*i* ∧ ((*length l1 = length l2*) ⟶ *lookup2 l1 l2 f = Some* (*l2*!*i*))
using *assms*
proof(*induct l1*)
case *Nil* **thus** *?case* **by** *auto*
next
case (*Cons h1 t1*)
{ assume *asm*:*f h1*
hence *0*<*length* (*h1* # *t1*) ∧ *e* = (*h1* # *t1*) ! *0*
using *Cons* **by** (*auto simp add*:*lookup.simps*)
moreover {
assume *length* (*h1* # *t1*) = *length l2*
hence *length l2 = Suc* (*length t1*) **by** *auto*
then obtain *h2 t2* **where** *l2-def*:*l2 = h2*#*t2* **by** (*auto simp add*: *length-Suc-conv*)
hence *lookup2* (*h1* # *t1*) *l2 f = Some* (*l2* ! *0*) **using** *asm* **by**(*auto simp add*: *lookup2.simps*)
}
ultimately have *?case* **by** *auto*
} moreover {
assume *asm*:¬ (*f h1*)
hence *lookup t1 f = Some e*
using *Cons* **by** (*auto simp add*:*lookup.simps*)

```
    then obtain i where
      i<length t1
      and e = t1 ! i
      and  ih:(length t1 = length (tl l2) ⟶ lookup2 t1 (tl l2) f = Some ((tl l2) !
i))
      using Cons by blast
    hence Suc i < length (h1#t1) ∧ e = (h1#t1)!(Suc i) using Cons by auto
    moreover {
      assume length (h1 # t1) = length l2
      hence lens:length l2 = Suc (length t1) by auto
    then obtain h2 t2 where l2-def:l2 = h2#t2 by (auto simp add: length-Suc-conv)
      hence lookup2 t1 t2 f = Some (t2 ! i) using ih l2-def lens by auto
      hence lookup2 (h1 # t1) l2 f = Some (l2!(Suc i))
        using asm l2-def by(auto simp add: lookup2.simps)
    }
    ultimately have ?case by auto
  }
  ultimately show ?case by auto
qed

lemma lookup2-index:
  ⋀l2. ⟦ lookup2 l1 l2 f = Some e;
  length l1 = length l2 ⟧ ⟹ ∃ i < (length l2). e = (l2!i) ∧ lookup l1 f = Some
(l1!i)
proof(induct l1)
  case Nil thus ?case by auto
next
  case (Cons h1 t1)
  hence length l2 = Suc (length t1) by auto
  then obtain h2 t2 where l2-def:l2 = h2#t2 by (auto simp add: length-Suc-conv)
  { assume asm:f h1
    hence e = h2 using Cons l2-def by (auto simp add:lookup2.simps)
    hence 0<length (h2#t2) ∧ e = (h2#t2) ! 0 ∧ lookup (h1 # t1) f = Some
((h1 # t1) ! 0)
      using asm by (auto simp add:lookup.simps)
    hence ?case using l2-def by auto
  } moreover {
    assume asm:¬ (f h1)
    hence ∃ i<length t2. e = t2 ! i ∧ lookup t1 f = Some (t1 ! i) using Cons l2-def
by auto
    then obtain i where i<length t2 ∧  e = t2 ! i ∧ lookup t1 f = Some (t1 ! i)
by auto
    hence (Suc i) < length(h2#t2) ∧ e = ((h2#t2) ! (Suc i)) ∧ lookup (h1#t1) f
= Some ((h1#t1) ! (Suc i))
      using asm by (force simp add: lookup.simps)
    hence ?case using l2-def by auto
  }
  ultimately show ?case by auto
qed
```

lemma *lookup-append*:
assumes *lookup l f = Some r*
shows *lookup (l@l') f = Some r*
using *assms* **by**(*induct l*) *auto*

lemma *method-typings-lookup*:
assumes *lookup-eq-Some*: *lookup M f = Some mDef*
and *M-ok*: *CT ⊢+ M OK IN C*
shows *CT ⊢ mDef OK IN C*
using *lookup-eq-Some M-ok*
proof(*induct M*)
case *Nil* **thus** *?case* **by** *fastforce*
next
case (*Cons h t*) **thus** *?case* **by**(*cases f h, auto elim*:*method-typings.cases simp add*:*lookup.simps*)
qed

2.2.3 Functional

These lemmas prove that several relations are actually functions

lemma *mtype-functional*:
assumes *mtype*(*CT*,*m*,*C*) = *Cs* → *C0*
and *mtype*(*CT*,*m*,*C*) = *Ds* → *D0*
shows *Ds*=*Cs* ∧ *D0*=*C0*
using *assms* **by** *induct* (*auto elim*:*mtype.cases*)

lemma *mbody-functional*:
assumes *mb1*: *mbody*(*CT*,*m*,*C*) = *xs* . *e0*
and *mb2*: *mbody*(*CT*,*m*,*C*) = *ys* . *d0*
shows *xs* = *ys* ∧ *e0* = *d0*
using *assms* **by** *induct* (*auto elim*:*mbody.cases*)

lemma *fields-functional*:
assumes *fields*(*CT*,*C*) = *Cf*
and *CT OK*
shows ⋀*Cf'*. ⟦ *fields*(*CT*,*C*) = *Cf'* ⟧ ⟹ *Cf* = *Cf'*
using *assms*
proof *induct*
case (*f-obj CT*)
hence *CT*(*Object*) = *None* **by** (*auto elim*: *ct-typing.cases*)
thus *?case* **using** *f-obj* **by** (*auto elim*: *fields.cases*)
next
case (*f-class CT C CDef D Cf Dg DgCf DgCf'*)
hence *f-class-inv*:
(*CT C = Some CDef*) ∧ (*cSuper CDef = D*) ∧ (*cFields CDef = Cf*)
and *CT OK* **by** *fastforce+*
hence *c-not-obj*:*C* ≠ *Object* **by** (*force elim*:*ct-typing.cases*)
from *f-class* **have** *flds*:*fields*(*CT*,*C*) = *DgCf'* **by** *fastforce*

then obtain Dg' **where**
$fields(CT,D) = Dg'$
and $DgCf' = Dg' @ Cf$
using *f-class-inv c-not-obj* **by** (*auto elim*:*fields.cases*)
hence $Dg' = Dg$ **using** *f-class* **by** *auto*
thus *?case* **using** ‹$DgCf = Dg @ Cf$› **and** ‹$DgCf' = Dg' @ Cf$› **by** *force*
qed

2.2.4 Subtyping and Typing

lemma *typings-lengths*: **assumes** $CT;\Gamma \vdash+ es:Cs$ **shows** $length\ es = length\ Cs$
using *assms* **by**(*induct es Cs*) (*auto elim*:*typings.cases*)

lemma *typings-index*:
assumes $CT;\Gamma \vdash+ es:Cs$
shows $\bigwedge i.\ [\![\ i < length\ es\ ]\!] \Longrightarrow CT;\Gamma \vdash (es!i) : (Cs!i)$
proof −
have $length\ es = length\ Cs$ **using** *assms* **by** (*auto simp*: *typings-lengths*)
thus $\bigwedge i.\ [\![\ i < length\ es\ ]\!] \Longrightarrow CT;\Gamma \vdash (es!i) : (Cs!i)$
using *assms*
proof(*induct es Cs rule*:*list-induct2*)
case *Nil* **thus** *?case* **by** *auto*
next
case (*Cons esh est hCs tCs i*)
thus *?case* **by**(*cases i*) (*auto elim*:*typings.cases*)
qed
qed

lemma *subtypings-index*:
assumes $CT \vdash+ Cs <: Ds$
shows $\bigwedge i.\ [\![\ i < length\ Cs\ ]\!] \Longrightarrow CT \vdash (Cs!i) <: (Ds!i)$
using *assms*
proof *induct*
case *ss-nil* **thus** *?case* **by** *auto*
next
case (*ss-cons hCs CT tCs hDs tDs i*)
thus *?case* **by**(*cases i*, *auto*)
qed

lemma *subtyping-append*:
assumes $CT \vdash+ Cs <: Ds$
and $CT \vdash C <: D$
shows $CT \vdash+ (Cs@[C]) <: (Ds@[D])$
using *assms*
by (*induct rule*:*subtypings.induct*) (*auto simp add*:*subtypings.intros elim*:*subtypings.cases*)

lemma *typings-append*:
assumes $CT;\Gamma \vdash+ es : Cs$

```
  and CT;Γ ⊢ e : C
  shows CT;Γ ⊢+ (es@[e]) : (Cs@[C])
proof −
  have length es = length Cs using assms by(simp-all add:typings-lengths)
  thus CT;Γ ⊢+ (es@[e]) : (Cs@[C]) using assms
  proof(induct es Cs rule:list-induct2)
    have CT;Γ ⊢+ []:[] by(simp add:typings-typing.ts-nil)
    moreover from assms have CT;Γ ⊢ e : C by simp
   ultimately show CT;Γ ⊢+ ([]@[e]) : ([]@[C]) by (auto simp add:typings-typing.ts-cons)
  next
    fix x xs y ys
    assume length xs = length ys
      and IH: ⟦CT;Γ ⊢+ xs : ys; CT;Γ ⊢ e : C⟧ ⟹ CT;Γ ⊢+ (xs @ [e]) : (ys @
[C])
      and x-xs-typs: CT;Γ ⊢+ (x # xs) : (y # ys)
      and e-typ: CT;Γ ⊢ e : C
      from x-xs-typs have x-typ: CT;Γ ⊢ x : y and CT;Γ ⊢+ xs : ys by(auto
elim:typings.cases)
    with IH e-typ have CT;Γ ⊢+ (xs@[e]) : (ys@[C]) by simp
   with x-typ have CT;Γ ⊢+ ((x#xs)@[e]) : ((y#ys)@[C]) by (auto simp add:typings-typing.ts-cons)
   thus CT;Γ ⊢+ ((x # xs) @ [e]) : ((y # ys) @ [C]) by(auto simp add:typings-typing.ts-cons)
  qed
qed

lemma ith-typing: ⋀Cs. ⟦ CT;Γ ⊢+ (es@(h#t)) : Cs ⟧ ⟹ CT;Γ ⊢ h : (Cs!(length
es))
proof(induct es, auto elim:typings.cases)
qed

lemma ith-subtyping: ⋀Ds. ⟦ CT ⊢+ (Cs@(h#t)) <: Ds ⟧ ⟹ CT ⊢ h <: (Ds!(length
Cs))
proof(induct Cs, auto elim:subtypings.cases)
qed

lemma subtypings-refl: CT ⊢+ Cs <: Cs
by(induct Cs, auto simp add: subtyping.s-refl subtypings.intros)

lemma subtypings-trans: ⋀Ds Es. ⟦ CT ⊢+ Cs <: Ds;  CT ⊢+ Ds <: Es ⟧ ⟹
CT ⊢+ Cs <: Es
proof(induct Cs)
  case Nil thus ?case
    by (auto elim:subtypings.cases simp add:subtypings.ss-nil)
next
  case (Cons hCs tCs)
  then obtain hDs tDs
    where h1:CT ⊢ hCs <: hDs and t1:CT ⊢+ tCs <: tDs and Ds = hDs#tDs
    by (auto elim:subtypings.cases)
  then obtain hEs tEs
    where h2:CT ⊢ hDs <: hEs and t2:CT ⊢+ tDs <: tEs and Es = hEs#tEs
```

```
    using Cons by (auto elim:subtypings.cases)
  moreover from subtyping.s-trans[OF h1 h2] have CT ⊢ hCs <: hEs by fastforce
  moreover with t1 t2 have CT ⊢+ tCs <: tEs using Cons by simp-all
  ultimately show ?case by (auto simp add:subtypings.intros)
qed

lemma ith-typing-sub:
  ⋀Cs. ⟦ CT;Γ ⊢+ (es@(h#t)) : Cs;
     CT;Γ ⊢ h' : Ci';
     CT ⊢ Ci' <: (Cs!(length es)) ⟧
  ⟹ ∃ Cs'. (CT;Γ ⊢+ (es@(h'#t)) : Cs' ∧ CT ⊢+ Cs' <: Cs)
proof(induct es)
case Nil
 then obtain hCs tCs
    where ts: CT;Γ ⊢+ t : tCs
    and Cs-def: Cs = hCs # tCs by(auto elim:typings.cases)
 from Cs-def Nil have CT ⊢ Ci' <: hCs by auto
 with Cs-def have CT ⊢+ (Ci'#tCs) <: Cs by(auto simp add:subtypings.ss-cons
subtypings-refl)
 moreover from ts Nil have CT;Γ ⊢+ (h'#t) : (Ci'#tCs) by(auto simp add:typings-typing.ts-cons)
 ultimately show ?case by auto
next
case (Cons eh et)
then obtain hCs tCs
  where CT;Γ ⊢ eh : hCs
  and CT;Γ ⊢+ (et@(h#t)) : tCs
  and Cs-def: Cs = hCs # tCs
  by(auto elim:typings.cases)
moreover with Cons obtain tCs'
  where CT;Γ ⊢+ (et@(h'#t)) : tCs'
  and CT ⊢+ tCs' <: tCs
  by auto
ultimately have
  CT;Γ ⊢+ (eh#(et@(h'#t))) : (hCs#tCs')
  and CT ⊢+ (hCs#tCs') <: Cs
  by(auto simp add:typings-typing.ts-cons subtypings.ss-cons subtyping.s-refl)
thus ?case by auto
qed

lemma mem-typings:
  ⋀Cs. ⟦ CT;Γ ⊢+ es:Cs; ei ∈ set es⟧ ⟹ ∃ Ci. CT;Γ ⊢ ei:Ci
proof(induct es)
  case Nil thus ?case by auto
next
  case (Cons eh et) thus ?case
    by(cases ei=eh, auto elim:typings.cases)
qed

lemma typings-proj:
```

```
  assumes CT;Γ ⊢+ ds : As
      and CT ⊢+ As <: Bs
      and length ds = length As
      and length ds = length Bs
      and i < length ds
    shows CT;Γ ⊢ ds!i : As!i and CT ⊢ As!i <: Bs!i
  using assms by (auto simp add:typings-index subtypings-index)

lemma subtypings-length:
  CT ⊢+ As <: Bs ⟹ length As = length Bs
  by(induct rule:subtypings.induct) simp-all

lemma not-subtypes-aux:
  assumes CT ⊢ C <: Da
  and C ≠ Da
  and CT C = Some CDef
  and cSuper CDef = D
  shows CT ⊢ D <: Da
  using assms
by (induct rule:subtyping.induct) (auto intro:subtyping.intros)

lemma not-subtypes:
  assumes CT ⊢ A <: C
  shows ⋀D. ⟦ CT ⊢ D ¬<: C;  CT ⊢ C ¬<: D⟧ ⟹ CT ⊢ A ¬<: D
  using assms
proof(induct rule:subtyping.induct)
  case s-refl thus ?case by auto
next
  case (s-trans CT C D E Da)
  have da-nsub-d:CT ⊢ Da ¬<: D
  proof(rule ccontr)
    assume ¬ CT ⊢ Da ¬<: D
    hence da-sub-d:CT ⊢ Da <: D by auto
    have d-sub-e:CT ⊢ D <: E using s-trans by fastforce
      thus False using s-trans by (force simp add:subtyping.s-trans[OF da-sub-d
d-sub-e])
  qed
  have d-nsub-da:CT ⊢ D ¬<: Da using s-trans by auto
  from da-nsub-d d-nsub-da s-trans show CT ⊢ C ¬<: Da by auto
next
  case (s-super CT C CDef D Da)
  have C ≠ Da proof(rule ccontr)
    assume ¬ C ≠ Da
    hence C = Da by auto
    hence CT ⊢ Da <: D using s-super by(auto simp add: subtyping.s-super)
    thus False using s-super by auto
  qed
  thus ?case using s-super by (auto simp add: not-subtypes-aux)
qed
```

2.2.5 Sub-Expressions

lemma *isubexpr-typing*:
assumes *e1* ∈ *isubexprs*(*e0*)
shows ⋀*C*. ⟦ *CT*;*Map.empty* ⊢ *e0* : *C* ⟧ ⟹ ∃ *D*. *CT*;*Map.empty* ⊢ *e1* : *D*
using *assms*
by (*induct rule*:*isubexprs.induct*) (*auto elim*:*typing.cases simp add*:*mem-typings*)

lemma *subexpr-typing*:
assumes *e1* ∈ *subexprs*(*e0*)
shows ⋀*C*. ⟦ *CT*;*Map.empty* ⊢ *e0* : *C* ⟧ ⟹ ∃ *D*. *CT*;*Map.empty* ⊢ *e1* : *D*
using *assms*
by (*induct rule*:*rtrancl.induct*) (*auto*, *force simp add*:*isubexpr-typing*)

lemma *isubexpr-reduct*:
assumes *d1* ∈ *isubexprs*(*e1*)
shows ⋀*d2*. ⟦ *CT* ⊢ *d1* → *d2* ⟧ ⟹ ∃ *e2*. *CT* ⊢ *e1* → *e2*
using *assms mem-ith*
by *induct*
(*auto elim*:*isubexprs.cases intro*:*reduction.intros*,
force intro:*reduction.intros*,
force intro:*reduction.intros*)

lemma *subexpr-reduct*:
assumes *d1* ∈ *subexprs*(*e1*)
shows ⋀*d2*. ⟦ *CT* ⊢ *d1* → *d2* ⟧ ⟹ ∃ *e2*. *CT* ⊢ *e1* → *e2*
using *assms*
by (*induct rule*:*rtrancl.induct*) (*auto*, *force simp add*: *isubexpr-reduct*)

end

3 FJSound: Type Soundness

theory *FJSound* **imports** *FJAux*
begin

Type soundness is proved using the standard technique of progress and subject reduction. The numbered lemmas and theorems in this section correspond to the same results in the ACM TOPLAS paper.

3.1 Method Type and Body Connection

lemma *mtype-mbody*:
fixes *Cs* :: *nat list*
assumes *mtype*(*CT*,*m*,*C*) = *Cs* → *C0*
shows ∃ *xs e*. *mbody*(*CT*,*m*,*C*) = *xs* . *e* ∧ *length xs* = *length Cs*
using *assms*
proof(*induct rule*:*mtype.induct*)
case(*mt-class C0 Cs C CDef CT m mDef*)

thus *?case*
by (*force simp add*:*varDefs-types-def varDefs-names-def elim*:*mtype.cases intro*:*mbody.mb-class*)
next
case(*mt-super CT C0 CDef m D Cs C*)
then obtain *xs e* **where** *mbody*(*CT*,*m*,*D*) = *xs* . *e* **and** *length xs* = *length Cs* **by** *auto*
thus *?case* **using** *mt-super* **by** (*auto intro*:*mbody.mb-super*)
qed

lemma *mtype-mbody-length*:
assumes *mt*:*mtype*(*CT*,*m*,*C*) = *Cs* → *C0*
and *mb*:*mbody*(*CT*,*m*,*C*) = *xs* . *e*
shows *length xs* = *length Cs*
proof −
from *mtype-mbody*[*OF mt*] **obtain** *xs'* *e'*
where *mb2*: *mbody*(*CT*,*m*,*C*) = *xs'* . *e'*
and *length xs'* = *length Cs*
by *auto*
with *mbody-functional*[*OF mb mb2*] **show** *?thesis* **by** *auto*
qed

3.2 Method Types and Field Declarations of Subtypes

lemma *A-1-1*:
assumes *CT* ⊢ *C* <: *D* **and** *CT OK*
shows (*mtype*(*CT*,*m*,*D*) = *Cs* → *C0*) ⟹ (*mtype*(*CT*,*m*,*C*) = *Cs* → *C0*)
using *assms*
proof (*induct rule*:*subtyping.induct*)
case (*s-refl C CT*) **show** *?case* **by** *fact*
next
case (*s-trans C CT D E*) **thus** *?case* **by** *auto*
next
case (*s-super CT C CDef D*)
hence *CT* ⊢ *CDef OK* **and** *cName CDef* = *C*
by(*auto elim*:*ct-typing.cases*)
with *s-super* **obtain** *M*
where *M*: *CT* ⊢+ *M OK IN C* **and** *cMethods*: *cMethods CDef* = *M*
by(*auto elim*:*class-typing.cases*)
let *?lookup-m* = *lookup M* (λ*md*. (*mName md* =*m*))
show *?case*
proof(*cases* ∃ *mDef*. *?lookup-m* = *Some mDef*)
case *True*
then obtain *mDef* **where** *m*: *?lookup-m* = *Some mDef* **by**(*rule exE*)
hence *mDef-name*: *mName mDef* = *m* **by** (*rule lookup-true*)
have *CT* ⊢ *mDef OK IN C* **using** *M m* **by**(*auto simp add*:*method-typings-lookup*)
then obtain *CDef'* *m'* *D'* *Cs'* *C0'*
where *CT*: *CT C* = *Some CDef'*
and *cSuper CDef'* = *D'*

```
      and mName mDef = m'
      and mReturn: mReturn mDef = C0'
      and varDefs-types: varDefs-types (mParams mDef) = Cs'
      and ∀ Ds D0. (mtype(CT,m',D') = Ds → D0) ⟶ Cs'=Ds ∧ C0'=D0
    by (auto elim: method-typing.cases)
  with s-super mDef-name have CDef=CDef'
    and D=D'
    and m=m'
    and ∀ Ds D0. (mtype(CT,m,D) = Ds → D0) ⟶ Cs'=Ds ∧ C0' = D0
    by auto
  thus ?thesis using s-super cMethods m CT mReturn varDefs-types by (auto
intro:mtype.intros)
 next
  case False
  hence ?lookup-m = None by (simp add: lookup-split)
  then show ?thesis using s-super cMethods by (auto simp add:mtype.intros)
 qed
qed


lemma sub-fields:
 assumes CT ⊢ C <: D
 shows ⋀Dg. fields(CT,D) = Dg ⟹ ∃ Cf. fields(CT,C) = (Dg@Cf)
using assms
proof induct
 case (s-refl CT C)
 hence fields(CT,C) = (Dg@[]) by simp
 thus ?case ..
next
 case (s-trans CT C D E)
 then obtain Df Cf where fields(CT,C) = ((Dg@Df)@Cf) by force
 thus ?case by auto
next
 case (s-super CT C CDef D Dg)
 then obtain Cf where cFields CDef = Cf by force
 with s-super have fields(CT,C) = (Dg@Cf) by(simp add:f-class)
 thus ?case ..
qed
```

3.3 Substitution Lemma

```
lemma A-1-2:
 assumes CT OK
   and Γ = Γ1 ++ Γ2
   and Γ2 = [xs [↦] Bs]
   and length xs = length ds
   and length Bs = length ds
   and ∃ As. CT;Γ1 ⊢+ ds : As ∧ CT ⊢+ As <: Bs
 shows CT;Γ ⊢+ es:Ds ⟹ ∃ Cs. (CT;Γ1 ⊢+ ([ds/xs]es):Cs ∧ CT ⊢+ Cs <:
```

```
Ds) (is ?TYPINGS ⟹ ?P1)
    and CT;Γ ⊢ e:D ⟹ ∃ C. (CT;Γ1 ⊢ ((ds/xs)e):C ∧ CT ⊢ C <: D) (is
?TYPING ⟹ ?P2)
proof −
  let ?COMMON-ASMS = (CT OK) ∧ (Γ = Γ1 ++ Γ2) ∧ (Γ2 = [xs [↦] Bs]) ∧
(length Bs = length ds) ∧ (∃ As. CT;Γ1 ⊢+ ds : As ∧ CT ⊢+ As <: Bs)
  have RESULT: (?TYPINGS ⟶ ?COMMON-ASMS ⟶ ?P1)
             ∧ (?TYPING ⟶ ?COMMON-ASMS ⟶ ?P2)
  proof(induct rule:typings-typing.induct)
    case (ts-nil CT Γ)
    show ?case
    proof (rule impI)
      have (CT;Γ1 ⊢+ ([ds/xs][]):[]) ∧ (CT ⊢+ [] <: [])
        by (auto simp add: typings-typing.intros subtypings.intros)
      then show ∃ Cs.(CT;Γ1 ⊢+ ([ds/xs][]):Cs) ∧ (CT ⊢+ Cs <: []) by auto
    qed
  next
    case(ts-cons CT Γ e0 C0 es Cs')
    show ?case
    proof (rule impI)
      assume asms: (CT OK) ∧ (Γ = Γ1 ++ Γ2) ∧ (Γ2 = [xs [↦] Bs]) ∧ (length
Bs = length ds) ∧ (∃ As. CT;Γ1 ⊢+ ds : As ∧ CT ⊢+ As <: Bs)
      with ts-cons have e0-typ: CT;Γ ⊢ e0 : C0 by fastforce
      with ts-cons asms have
          ∃ C.(CT;Γ1 ⊢ (ds/xs) e0 : C) ∧ (CT ⊢ C <: C0)
        and ∃ Cs.(CT;Γ1 ⊢+ [ds/xs]es : Cs) ∧ (CT ⊢+ Cs <: Cs')
        by auto
      then obtain C Cs where
          (CT;Γ1 ⊢ (ds/xs) e0 : C) ∧ (CT ⊢ C <: C0)
        and (CT;Γ1 ⊢+ [ds/xs]es : Cs) ∧ (CT ⊢+ Cs <: Cs') by auto
      hence CT;Γ1 ⊢+ [ds/xs](e0#es) : (C#Cs)
        and CT ⊢+ (C#Cs) <: (C0#Cs')
        by (auto simp add: typings-typing.intros subtypings.intros)
      then show ∃ Cs. CT;Γ1 ⊢+ map (substs [xs [↦] ds]) (e0 # es) : Cs ∧ CT
⊢+ Cs <: (C0 # Cs')
        by auto
    qed
  next
    case (t-var Γ x C' CT)
    show ?case
    proof (rule impI)
      assume asms: (CT OK) ∧ (Γ = Γ1 ++ Γ2) ∧ (Γ2 = [xs [↦] Bs]) ∧ (length
Bs = length ds) ∧ (∃ As. CT;Γ1 ⊢+ ds : As ∧ CT ⊢+ As <: Bs)
      hence
        lengths: length ds = length Bs
        and G-def: Γ = Γ1 ++ Γ2
        and G2-def : Γ2 = [xs[↦]Bs] by auto
      from lengths G2-def have same-doms: dom([xs[↦]ds]) = dom(Γ2) by auto
      from asms show ∃ C. CT;Γ1 ⊢ substs [xs [↦] ds] (Var x) : C ∧ CT ⊢ C <:
```

```
C'
      proof (cases Γ2 x)
        case None
       with G-def t-var have G1-x: Γ1 x = Some C' by(simp add:map-add-Some-iff)
        from None same-doms have x ∉ dom([xs[↦]ds]) by (auto simp only:domIff)

        hence [xs[↦]ds]x = None by(auto simp only:map-add-Some-iff)
        hence (ds/xs)(Var x) = (Var x) by auto
        with G1-x have
          CT;Γ1 ⊢ (ds/xs)(Var x) : C' and CT ⊢ C' <: C'
          by (auto simp add:typings-typing.intros subtyping.intros)
        thus ?thesis by auto
      next
        case (Some Bi)
       with G-def t-var have c'-eq-bi: C' = Bi by (auto simp add: map-add-SomeD)
        from ‹length xs = length ds› asms have length xs = length Bs by simp
        with Some G2-def have ∃ i.(Bs!i = Bi) ∧ (i < length Bs) ∧
            (∀ l.((length l = length Bs) ⟶ ([xs[↦]l] x = Some (l!i))))
          by (auto simp add: map-upds-index)
        then obtain i where bs-i-proj: (Bs!i = Bi)
          and i-len: i < length Bs
            and P: (⋀(l::exp list).((length l = length Bs) ⟶ ([xs[↦]l] x = Some
(l!i))))
          by fastforce
        from lengths P have subst-x: ([xs[↦]ds]x = Some (ds!i)) by auto
        from asms obtain As where as-ex:CT;Γ1 ⊢+ ds : As ∧ CT ⊢+ As <: Bs
by fastforce
        hence length As = length Bs by (auto simp add: subtypings-length)
        hence proj-i:CT;Γ1 ⊢ ds!i : As!i ∧ CT ⊢ As!i <: Bs!i
          using i-len lengths as-ex by (auto simp add: typings-proj)
        hence CT;Γ1 ⊢ (ds/xs)(Var x) : As!i ∧ CT ⊢ As!i <: C'
          using c'-eq-bi bs-i-proj subst-x by auto
        thus ?thesis ..
      qed
    qed
  next
    case(t-field CT Γ e0 C0 Cf fi fDef Ci)
    show ?case
    proof(rule impI)
      assume asms: (CT OK) ∧ (Γ = Γ1 ++ Γ2) ∧
        (Γ2 = [xs [↦] Bs]) ∧ (length Bs = length ds) ∧ (∃ As. CT;Γ1 ⊢+ ds : As
∧ CT ⊢+ As <: Bs)
      from t-field have flds: fields(CT,C0) = Cf by fastforce
       from t-field asms obtain C where e0-typ: CT;Γ1 ⊢ (ds/xs)e0 : C and sub:
CT ⊢ C <: C0
        by auto
     from sub-fields[OF sub flds] obtain Dg where flds-C: fields(CT,C) = (Cf@Dg)
..
       from t-field have lookup-CfDg: lookup (Cf@Dg) (λfd. vdName fd = fi) =
```

Some fDef
by(*simp add*:*lookup-append*)
from *e0-typ flds-C lookup-CfDg t-field* **have** $CT;\Gamma 1 \vdash (ds/xs)(FieldProj\ e0\ fi) : Ci$
by(*simp add*:*typings-typing.intros*)
moreover have $CT \vdash Ci <: Ci$ **by** (*simp add*:*subtyping.intros*)
ultimately show $\exists C.\ CT;\Gamma 1 \vdash (ds/xs)(FieldProj\ e0\ fi) : C \wedge CT \vdash C <: Ci$ **by** *auto*
qed
next
case(*t-invk CT* Γ *e0 C0 m Ds C es Cs*)
show *?case*
proof(*rule impI*)
assume *asms*: $(CT\ OK) \wedge (\Gamma = \Gamma 1 +\!\!+ \Gamma 2) \wedge (\Gamma 2 = [xs\ [\mapsto]\ Bs]) \wedge (length\ Bs = length\ ds) \wedge (\exists As.\ CT;\Gamma 1 \vdash+\ ds : As \wedge CT \vdash+\ As <: Bs)$
hence *ct-ok*: $CT\ OK$ **..**
from *t-invk* **have** *mtyp*: $mtype(CT,m,C0) = Ds \to C$
and *subs*: $CT \vdash+\ Cs <: Ds$
and *lens*: $length\ es = length\ Ds$
by *auto*
from *t-invk asms* **obtain** C' **where**
e0-typ: $CT;\Gamma 1 \vdash (ds/xs)e0 : C'$ **and** *sub'*: $CT \vdash C' <: C0$ **by** *auto*
from *t-invk asms* **obtain** Cs' **where**
es-typ: $CT;\Gamma 1 \vdash+ [ds/xs]es : Cs'$ **and** *subs'*: $CT \vdash+ Cs' <: Cs$ **by** *auto*
have *subst-e*: $(ds/xs)(MethodInvk\ e0\ m\ es) = MethodInvk\ ((ds/xs)e0)\ m\ ([ds/xs]es)$
by(*auto simp add*: *subst-list1-eq-map-substs*)
from
e0-typ
A-1-1[*OF sub' ct-ok mtyp*]
es-typ
subtypings-trans[*OF subs' subs*]
lens
subst-e
have $CT;\Gamma 1 \vdash (ds/xs)(MethodInvk\ e0\ m\ es) : C$ **by**(*auto simp add*:*typings-typing.intros*)
moreover have $CT \vdash C <: C$ **by**(*simp add*:*subtyping.intros*)
ultimately show $\exists C'.\ CT;\Gamma 1 \vdash (ds/xs)(MethodInvk\ e0\ m\ es) : C' \wedge CT \vdash C' <: C$ **by** *auto*
qed
next
case(*t-new CT C Df es Ds* Γ *Cs*)
show *?case*
proof(*rule impI*)
assume *asms*: $(CT\ OK) \wedge (\Gamma = \Gamma 1 +\!\!+ \Gamma 2) \wedge (\Gamma 2 = [xs\ [\mapsto]\ Bs]) \wedge (length\ Bs = length\ ds) \wedge (\exists As.\ CT;\Gamma 1 \vdash+\ ds : As \wedge CT \vdash+\ As <: Bs)$
hence *ct-ok*: $CT\ OK$ **..**
from *t-new* **have**
subs: $CT \vdash+\ Cs <: Ds$

and *flds*: $fields(CT,C) = Df$
and *len*: $length\ es = length\ Df$
and *vdts*: $varDefs\text{-}types\ Df = Ds$
by *auto*
from *t-new asms* **obtain** Cs' **where**
$es\text{-}typ$: $CT;\Gamma 1 \vdash+ [ds/xs]es : Cs'$ **and** *subs'*: $CT \vdash+ Cs' <: Cs$ **by** *auto*
have *subst-e*: $(ds/xs)(New\ C\ es) = New\ C\ ([ds/xs]es)$
by(*auto simp add*: *subst-list2-eq-map-substs*)
from *es-typ subtypings-trans*[*OF subs' subs*] *flds subst-e len vdts*
have $CT;\Gamma 1 \vdash (ds/xs)(New\ C\ es) : C$ **by**(*auto simp add*:*typings-typing.intros*)
moreover have $CT \vdash C <: C$ **by**(*simp add*:*subtyping.intros*)
ultimately show $\exists C'.\ CT;\Gamma 1 \vdash (ds/xs)(New\ C\ es) : C' \wedge CT \vdash C' <: C$ **by** *auto*
qed
next
case(*t-ucast CT* Γ *e0 D C*)
show *?case*
proof(*rule impI*)
assume *asms*: $(CT\ OK) \wedge (\Gamma = \Gamma 1 +\!\!+ \Gamma 2) \wedge (\Gamma 2 = [xs\ [\mapsto]\ Bs]) \wedge (length\ Bs = length\ ds) \wedge (\exists As.\ CT;\Gamma 1 \vdash+ ds : As \wedge CT \vdash+ As <: Bs)$
from *t-ucast asms* **obtain** C' **where** *e0-typ*: $CT;\Gamma 1 \vdash (ds/xs)e0 : C'$
and *sub1*:$CT \vdash C' <: D$
and *sub2*:$CT \vdash D <: C$ **by** *auto*
from *sub1 sub2* **have** $CT \vdash C' <: C$ **by** (*rule s-trans*)
with *e0-typ* **have** $CT;\Gamma 1 \vdash (ds/xs)(Cast\ C\ e0) : C$ **by**(*auto simp add*: *typings-typing.intros*)
moreover have $CT \vdash C <: C$ **by** (*rule s-refl*)
ultimately show $\exists C'.\ CT;\Gamma 1 \vdash (ds/xs)(Cast\ C\ e0) : C' \wedge CT \vdash C' <: C$ **by** *auto*
qed
next
case(*t-dcast CT* Γ *e0 D C*)
show *?case*
proof(*rule impI*)
assume *asms*: $(CT\ OK) \wedge (\Gamma = \Gamma 1 +\!\!+ \Gamma 2) \wedge (\Gamma 2 = [xs\ [\mapsto]\ Bs]) \wedge (length\ Bs = length\ ds) \wedge (\exists As.\ CT;\Gamma 1 \vdash+ ds : As \wedge CT \vdash+ As <: Bs)$
from *t-dcast asms* **obtain** C' **where** *e0-typ*:$CT;\Gamma 1 \vdash (ds/xs)e0 : C'$ **by** *auto*
have $(CT \vdash C' <: C)\ \vee$
$(C \neq C' \wedge CT \vdash C <: C') \vee$
$(CT \vdash C\ \neg<:\ C' \wedge CT \vdash C'\ \neg<:\ C)$ **by** *blast*
moreover
{ assume $CT \vdash C' <: C$
with *e0-typ* **have** $CT;\Gamma 1 \vdash (ds/xs)\ (Cast\ C\ e0) : C$ **by** (*auto simp add*: *typings-typing.intros*)
}
moreover
{ assume $(C \neq C' \wedge CT \vdash C <: C')$
with *e0-typ* **have** $CT;\Gamma 1 \vdash (ds/xs)\ (Cast\ C\ e0) : C$ **by** (*auto simp add*: *typings-typing.intros*)

```
    }
    moreover
    { assume (CT ⊢ C ¬<: C' ∧ CT ⊢ C' ¬<: C)
       with e0-typ have CT;Γ1 ⊢ (ds/xs) (Cast C e0) : C by (auto simp add:
typings-typing.intros)
    }
    ultimately have CT;Γ1 ⊢ (ds/xs) (Cast C e0) : C by auto
    moreover have CT ⊢ C <: C by(rule s-refl)
   ultimately show ∃ C'. CT;Γ1 ⊢ (ds/xs)(Cast C e0) : C' ∧ CT ⊢ C' <: C by
auto
  qed
 next
  case(t-scast CT Γ e0 D C)
  show ?case
  proof(rule impI)
   assume asms: (CT OK) ∧ (Γ = Γ1 ++ Γ2) ∧ (Γ2 = [xs [↦] Bs]) ∧ (length
Bs = length ds) ∧ (∃ As. CT;Γ1 ⊢+ ds : As ∧ CT ⊢+ As <: Bs)
   from t-scast asms obtain C' where e0-typ:CT;Γ1 ⊢ (ds/xs)e0 : C'
     and sub1: CT ⊢ C' <: D
     and nsub1: CT ⊢ C ¬<: D
     and nsub2: CT ⊢ D ¬<: C by auto
   from not-subtypes[OF sub1 nsub1 nsub2] have CT ⊢ C' ¬<: C by fastforce
   moreover have CT ⊢ C ¬<: C' proof(rule ccontr)
     assume ¬ CT ⊢ C ¬<: C'
     hence CT ⊢ C <: C' by auto
     hence CT ⊢ C <: D using sub1 by(rule s-trans)
     with nsub1 show False by auto
   qed
   ultimately have CT;Γ1 ⊢ (ds/xs) (Cast C e0) : C using e0-typ by (auto simp
add: typings-typing.intros)
   thus ∃ C'. CT;Γ1 ⊢ (ds/xs)(Cast C e0) : C' ∧ CT ⊢ C' <: C by (auto simp
add: s-refl)
  qed
 qed
 thus ?TYPINGS ⟹ ?P1 and ?TYPING ⟹ ?P2 using assms by auto
qed
```

3.4 Weakening Lemma

This lemma is not in the same form as in TOPLAS, but rather as we need it in subject reduction

```
lemma A-1-3:
 shows (CT;Γ2 ⊢+ es : Cs) ⟹ (CT;Γ1++Γ2 ⊢+ es : Cs) (is ?P1 ⟹ ?P2)
 and CT;Γ2 ⊢ e : C ⟹ CT;Γ1++Γ2 ⊢ e : C (is ?Q1 ⟹ ?Q2)
proof −
 have (?P1 ⟶ ?P2) ∧ (?Q1 ⟶ ?Q2)
 by(induct rule:typings-typing.induct, auto simp add: map-add-find-right typings-typing.intros)

 thus ?P1 ⟹ ?P2 and ?Q1 ⟹ ?Q2 by auto
```

qed

3.5 Method Body Typing Lemma

lemma *A-1-4*:
assumes *ct-ok*: $CT\ OK$
and *mb*:$mbody(CT,m,C) = xs\ .\ e$
and *mt*:$mtype(CT,m,C) = Ds \to D$
shows $\exists\, D0\ C0.\ (CT \vdash C <: D0) \wedge$
$(CT \vdash C0 <: D) \wedge$
$(CT;[xs[\mapsto]]Ds](this \mapsto D0) \vdash e : C0)$
using *mb ct-ok mt* **proof**(*induct rule*: *mbody.induct*)
case (*mb-class CT C CDef m mDef xs e*)
hence
m-param:*varDefs-types* (*mParams mDef*) = *Ds*
and *m-ret*:*mReturn mDef* = *D*
and $CT \vdash CDef\ OK$
and *cName CDef* = *C*
by (*auto elim*:*mtype.cases ct-typing.cases*)
hence $CT \vdash+$ (*cMethods CDef*) *OK IN C* **by** (*auto elim*:*class-typing.cases*)
hence $CT \vdash mDef\ OK\ IN\ C$ **using** *mb-class* **by**(*auto simp add*:*method-typings-lookup*)
hence $\exists\ E0.\ ((CT;[xs[\mapsto]]Ds,this\mapsto C] \vdash e : E0) \wedge (CT \vdash E0 <: D))$
using *mb-class m-param m-ret* **by**(*auto elim*:*method-typing.cases*)
then obtain *E0*
where $CT;[xs[\mapsto]]Ds,this\mapsto C] \vdash e : E0$
and $CT \vdash E0 <: D$
and $CT \vdash C <: C$ **by** (*auto simp add*: *s-refl*)
thus *?case* **by** *blast*
next
case (*mb-super CT C CDef m Da xs e*)
hence *ct*: $CT\ OK$
and *IH*: $[\![CT\ OK;\ mtype(CT,m,Da) = Ds \to D]\!]$
$\Longrightarrow \exists\, D0\ C0.\ (CT \vdash Da <: D0) \wedge (CT \vdash C0 <: D)$
$\wedge\ (CT;[xs\ [\mapsto]\ Ds,\ this \mapsto D0] \vdash e{:}C0)$ **by** *fastforce+*
from *mb-super* **have** *c-sub-da*: $CT \vdash C <: Da$ **by** (*auto simp add*:*s-super*)
from *mb-super* **have** *mt*:$mtype(CT,m,Da) = Ds \to D$ **by** (*auto elim*: *mtype.cases*)
from *IH*[*OF ct mt*] **obtain** *D0 C0*
where *s1*: $CT \vdash Da <: D0$
and $CT \vdash C0 <: D$
and $CT;[xs\ [\mapsto]\ Ds,\ this \mapsto D0] \vdash e : C0$ **by** *auto*
thus *?case* **using** *s-trans*[*OF c-sub-da s1*] **by** *blast*
qed

3.6 Subject Reduction Theorem

theorem *Thm-2-4-1*:
assumes $CT \vdash e \to e'$
and $CT\ OK$
shows $\bigwedge C.\ [\![\ CT;\Gamma \vdash e : C\ ]\!]$
$\Longrightarrow \exists\, C'.\ (CT;\Gamma \vdash e' : C' \wedge CT \vdash C' <: C)$

using *assms*
proof(*induct rule*: *reduction.induct*)
case (*r-field CT Ca Cf es fi e'*)
hence $CT;\Gamma \vdash FieldProj\ (New\ Ca\ es)\ fi : C$
and *ct-ok*: $CT\ OK$
and *flds*: $fields(CT,Ca) = Cf$
and *lkup2*: $lookup2\ Cf\ es\ (\lambda fd.\ vdName\ fd = fi) = Some\ e'$ **by** *fastforce+*
then obtain $Ca'\ Cf'\ fDef$
where *new-typ*: $CT;\Gamma \vdash New\ Ca\ es : Ca'$
and *flds'*:$fields(CT,Ca') = Cf'$
and *lkup*: $lookup\ Cf'\ (\lambda fd.\ vdName\ fd = fi) = Some\ fDef$
and *C-def*: $vdType\ fDef = C$ **by** (*auto elim*: *typing.cases*)
hence *Ca-Ca'*: $Ca = Ca'$ **by** (*auto elim*:*typing.cases*)
with *flds'* **have** *Cf-Cf'*: $Cf = Cf'$ **by**(*simp add*:*fields-functional*[*OF flds ct-ok*])
from *new-typ* **obtain** $Cs\ Ds\ Cf''$
where $fields(CT,Ca') = Cf''$
and *es-typs*: $CT;\Gamma \vdash+\ es:Cs$
and *Ds-def*: $varDefs\text{-}types\ Cf'' = Ds$
and *length-Cf-es*: $length\ Cf'' = length\ es$
and *subs*: $CT \vdash+\ Cs <: Ds$
by(*auto elim*:*typing.cases*)
with *Ca-Ca'* **have** *Cf-Cf''*: $Cf = Cf''$ **by**(*auto simp add*:*fields-functional*[*OF flds ct-ok*])
from *length-Cf-es Cf-Cf'' lookup2-index*[*OF lkup2*] **obtain** i **where**
i-bound: $i < length\ es$
and $e' = es!i$
and $lookup\ Cf\ (\lambda fd.\ vdName\ fd = fi) = Some\ (Cf!i)$ **by** *auto*
moreover
with *C-def Ds-def lkup lkup2* **have** $Ds!i = C$
using *Ca-Ca' Cf-Cf' Cf-Cf'' i-bound length-Cf-es flds'*
by (*auto simp add*:*nth-map varDefs-types-def fields-functional*[*OF flds ct-ok*])
moreover with *subs es-typs* **have**
$CT;\Gamma \vdash (es!i):(Cs!i)$ **and** $CT \vdash (Cs!i) <: (Ds!i)$ **using** *i-bound*
by(*auto simp add*:*typings-index subtypings-index typings-lengths*)
ultimately show *?case* **by** *auto*
next
case(*r-invk CT m Ca xs e ds es e'*)
from *r-invk* **have** *mb*: $mbody(CT,m,Ca) = xs\ .\ e$ **by** *fastforce*
from *r-invk* **obtain** $Ca'\ Ds\ Cs$
where $CT;\Gamma \vdash New\ Ca\ es : Ca'$
and $mtype(CT,m,Ca') = Cs \to C$
and *ds-typs*: $CT;\Gamma \vdash+\ ds : Ds$
and *Ds-subs*: $CT \vdash+\ Ds <: Cs$
and *l1*: $length\ ds = length\ Cs$ **by**(*auto elim*:*typing.cases*)
hence *new-typ*: $CT;\Gamma \vdash New\ Ca\ es : Ca$
and *mt*: $mtype(CT,m,Ca) = Cs \to C$ **by** (*auto elim*:*typing.cases*)
from *ds-typs new-typ* **have** $CT;\Gamma \vdash+\ (ds\ @[New\ Ca\ es]) : (Ds\ @[Ca])$
by (*simp add*:*typings-append*)
moreover from *A-1-4*[*OF - mb mt*] *r-invk* **obtain** $Da\ E$

where $CT \vdash Ca <: Da$
and *E-sub-C*: $CT \vdash E <: C$
and *e0-typ1*: $CT;[xs[\mapsto]Cs,this\mapsto Da] \vdash e : E$ **by** *auto*
moreover with *Ds-subs* **have** $CT \vdash+ (Ds@[Ca]) <: (Cs@[Da])$
by(*auto simp add*:*subtyping-append*)
ultimately have *ex*: $\exists As.\ CT;\Gamma \vdash+ (ds\ @[New\ Ca\ es]) : As \wedge CT \vdash+ As <: (Cs@[Da])$
by *auto*
from *e0-typ1* **have** *e0-typ2*: $CT;(\Gamma +\!\!+ [xs[\mapsto]Cs,this\mapsto Da]) \vdash e : E$
by(*simp only*:*A-1-3*)
from *e0-typ2 mtype-mbody-length*[*OF mt mb*]
have *e0-typ3*: $CT;(\Gamma +\!\!+ [(xs@[this])[\mapsto](Cs@[Da])]) \vdash e : E$
by(*force simp only*:*map-shuffle*)
let $?\Gamma 1 = \Gamma$ **and** $?\Gamma 2 = [(xs@[this])[\mapsto](Cs@[Da])]$
have *g-def*: $(?\Gamma 1 +\!\!+ ?\Gamma 2) = (?\Gamma 1 +\!\!+ ?\Gamma 2)$ **and** *g2-def*: $?\Gamma 2 = ?\Gamma 2$ **by** *auto*
from *A-1-2*[*OF - g-def g2-def - - ex*] *e0-typ3 r-invk l1 mtype-mbody-length*[*OF mt mb*]
obtain E' **where** *e'-typ*: $CT;\Gamma \vdash substs\ [(xs@[this])[\mapsto](ds@[New\ Ca\ es])]\ e : E'$
and *E'-sub-E*: $CT \vdash E' <: E$ **by** *force*
moreover from *e'-typ l1 mtype-mbody-length*[*OF mt mb*]
have $CT;\Gamma \vdash substs\ [xs[\mapsto]ds,this\mapsto(New\ Ca\ es)]\ e : E'$
by(*auto simp only*:*map-shuffle*)
moreover from *E'-sub-E E-sub-C* **have** $CT \vdash E' <: C$ **by** (*rule subtyping.s-trans*)
ultimately show *?case* **using** *r-invk* **by** *auto*
next
case (*r-cast CT Ca D es*)
then obtain *Ca'*
where $C = D$
and $CT;\Gamma \vdash New\ Ca\ es : Ca'$ **by** (*auto elim*: *typing.cases*)
thus *?case* **using** *r-cast* **by** (*auto elim*: *typing.cases*)
next
case (*rc-field CT e0 e0' f*)
then obtain *C0 Cf fd* **where** $CT;\Gamma \vdash e0 : C0$
and *Cf-def*: $fields(CT,C0) = Cf$
and *fd-def*:*lookup Cf* $(\lambda fd.\ (vdName\ fd = f)) = Some\ fd$
and $vdType\ fd = C$
by (*auto elim*:*typing.cases*)
moreover with *rc-field* **obtain** C'
where $CT;\Gamma \vdash e0' : C'$
and $CT \vdash C' <: C0$ **by** *auto*
moreover from *sub-fields*[*OF - Cf-def*] **obtain** Cf'
where $fields(CT,C') = (Cf@Cf')$ **by** *rule* (*rule* ‹$CT \vdash C' <: C0$›)
moreover with *fd-def* **have** *lookup* $(Cf@Cf')$ $(\lambda fd.\ (vdName\ fd = f)) = Some\ fd$
by(*simp add*:*lookup-append*)
ultimately have $CT;\Gamma \vdash FieldProj\ e0'\ f : C$ **by**(*auto simp add*:*typings-typing.t-field*)
thus *?case* **by** (*auto simp add*:*subtyping.s-refl*)
next
case (*rc-invk-recv CT e0 e0' m es C*)
then obtain *C0 Ds Cs*

```
    where ct-ok:CT OK
    and CT;Γ ⊢ e0 : C0
    and mt:mtype(CT,m,C0) = Ds → C
    and CT;Γ ⊢+ es : Cs
    and length es = length Ds
    and CT ⊢+ Cs <: Ds
    by (auto elim:typing.cases)
  moreover with rc-invk-recv obtain C0'
    where CT;Γ ⊢ e0' : C0'
    and CT ⊢ C0' <: C0 by auto
  moreover with A-1-1[OF - ct-ok mt] have mtype(CT,m,C0') = Ds → C by simp
  ultimately have CT;Γ ⊢ MethodInvk e0' m es : C by(auto simp add:typings-typing.t-invk)
  thus ?case by (auto simp add:subtyping.s-refl)
next
  case (rc-invk-arg CT ei ei' e0 m el er C)
  then obtain Cs Ds C0
    where typs: CT;Γ ⊢+ (el@(ei#er)) : Cs
    and e0-typ: CT;Γ ⊢ e0 : C0
    and mt: mtype(CT,m,C0) = Ds → C
    and Cs-sub-Ds: CT ⊢+ Cs <: Ds
    and len: length (el@(ei#er)) = length Ds
    by(auto elim:typing.cases)
  hence CT;Γ ⊢ ei:(Cs!(length el)) by (simp add:ith-typing)
  with rc-invk-arg obtain Ci'
    where ei-typ: CT;Γ ⊢ ei':Ci'
    and Ci-sub: CT ⊢ Ci' <: (Cs!(length el))
    by auto
  from ith-typing-sub[OF typs ei-typ Ci-sub] obtain Cs'
    where es'-typs: CT;Γ ⊢+ (el@(ei'#er)) : Cs'
    and Cs'-sub-Cs: CT ⊢+ Cs' <: Cs by auto
  from len have length (el@(ei'#er)) = length Ds by simp
  with es'-typs subtypings-trans[OF Cs'-sub-Cs Cs-sub-Ds]  e0-typ mt have
    CT;Γ ⊢ MethodInvk e0 m (el@(ei'#er)) : C
    by(auto simp add:typings-typing.t-invk)
  thus ?case by (auto simp add:subtyping.s-refl)
next
  case (rc-new-arg CT ei ei' Ca el er C)
  then obtain Cs Df Ds
    where typs: CT;Γ ⊢+ (el@(ei#er)) : Cs
    and flds: fields(CT,C) = Df
    and len: length (el@(ei#er)) = length Df
    and Ds-def: varDefs-types Df = Ds
    and Cs-sub-Ds: CT ⊢+ Cs <: Ds
    and C-def: Ca = C
    by(auto elim:typing.cases)
  hence CT;Γ ⊢ ei:(Cs!(length el)) by (simp add:ith-typing)
  with rc-new-arg obtain Ci'
    where ei-typ: CT;Γ ⊢ ei':Ci'
    and Ci-sub: CT ⊢ Ci' <: (Cs!(length el))
```

by *auto*
from *ith-typing-sub*[*OF typs ei-typ Ci-sub*] **obtain** *Cs′*
where *es′-typs*: $CT;\Gamma \vdash+ (el@(ei'\#er)) : Cs'$
and *Cs′-sub-Cs*: $CT \vdash+ Cs' <: Cs$ **by** *auto*
from *len* **have** *length* $(el@(ei'\#er)) = length\ Df$ **by** *simp*
with *es′-typs subtypings-trans*[*OF Cs′-sub-Cs Cs-sub-Ds*] *flds Ds-def C-def* **have**
$CT;\Gamma \vdash New\ Ca\ (el@(ei'\#er)) : C$
by(*auto simp add*:*typings-typing.t-new*)
thus *?case* **by** (*auto simp add*:*subtyping.s-refl*)
next
case (*rc-cast CT e0 e0′ C Ca*)
then obtain *D*
where $CT;\Gamma \vdash e0 : D$
and *Ca-def*: $Ca = C$
by(*auto elim*:*typing.cases*)
with *rc-cast* **obtain** D'
where *e0′-typ*: $CT;\Gamma \vdash e0':D'$ **and** $CT \vdash D' <: D$
by *auto*
have $(CT \vdash D' <: C)\ \vee$
$(C \neq D' \wedge CT \vdash C <: D') \vee$
$(CT \vdash C \neg<: D' \wedge CT \vdash D' \neg<: C)$ **by** *blast*
moreover {
assume $CT \vdash D' <: C$
with *e0′-typ* **have** $CT;\Gamma \vdash Cast\ C\ e0' : C$ **by** (*auto simp add*: *typings-typing.t-ucast*)
} moreover {
assume $(C \neq D' \wedge CT \vdash C <: D')$
with *e0′-typ* **have** $CT;\Gamma \vdash Cast\ C\ e0' : C$ **by** (*auto simp add*: *typings-typing.t-dcast*)
} moreover {
assume $(CT \vdash C \neg<: D' \wedge CT \vdash D' \neg<: C)$
with *e0′-typ* **have** $CT;\Gamma \vdash Cast\ C\ e0' : C$ **by** (*auto simp add*: *typings-typing.t-scast*)
} ultimately have $CT;\Gamma \vdash Cast\ C\ e0' : C$ **by** *auto*
thus *?case* **using** *Ca-def* **by** (*auto simp add*:*subtyping.s-refl*)
qed

3.7 Multi-Step Subject Reduction Theorem

corollary *Cor-2-4-1-multi*:
assumes $CT \vdash e \rightarrow* e'$
and *CT OK*
shows $\bigwedge C.\ [\![\ CT;\Gamma \vdash e : C\ ]\!] \Longrightarrow \exists C'.\ (CT;\Gamma \vdash e' : C' \wedge CT \vdash C' <: C)$
using *assms*
proof *induct*
case (*rs-refl CT e C*) **thus** *?case* **by** (*auto simp add*:*subtyping.s-refl*)
next
case(*rs-trans CT e e′ e″ C*)
hence *e-typ*: $CT;\Gamma \vdash e : C$
and *e-step*: $CT \vdash e \rightarrow e'$
and *ct-ok*: *CT OK*
and *IH*: $\bigwedge D.\ [\![CT;\Gamma \vdash e' : D;\ CT\ OK]\!] \Longrightarrow \exists E.\ CT;\Gamma \vdash e'' : E \wedge CT \vdash E <:$

```
D
    by auto
  from Thm-2-4-1[OF e-step ct-ok e-typ] obtain D where e'-typ: CT;Γ ⊢ e' : D
and D-sub-C: CT ⊢ D <: C by auto
  with IH[OF e'-typ ct-ok] obtain E where CT;Γ ⊢ e'': E and E-sub-D: CT ⊢ E
<: D by auto
  moreover from s-trans[OF E-sub-D D-sub-C] have CT ⊢ E <: C by auto
  ultimately show ?case by auto
qed
```

3.8 Progress

The two "progress lemmas" proved in the TOPLAS paper alone are not quite enough to prove type soundness. We prove an additional lemma showing that every well-typed expression is either a value or contains a potential redex as a sub-expression.

```
theorem Thm-2-4-2-1:
  assumes CT;Map.empty ⊢ e : C
  and FieldProj (New C0 es) fi ∈ subexprs(e)
  shows ∃ Cf fDef. fields(CT, C0) = Cf ∧ lookup Cf (λfd. (vdName fd = fi)) =
Some fDef
proof −
  obtain Ci where CT;Map.empty ⊢ (FieldProj (New C0 es) fi) : Ci
    using assms by (force simp add:subexpr-typing)
  then obtain Cf fDef C0'
    where CT;Map.empty ⊢ (New C0 es) : C0'
    and fields(CT,C0') = Cf
    and lookup Cf (λfd. (vdName fd = fi)) = Some fDef
    by (auto elim:typing.cases)
  thus ?thesis by (auto elim:typing.cases)
qed

lemma Thm-2-4-2-2:
  fixes es ds :: exp list
  assumes CT;Map.empty ⊢ e : C
  and MethodInvk (New C0 es) m ds ∈ subexprs(e)
  shows ∃ xs e0. mbody(CT,m,C0) = xs . e0 ∧ length xs = length ds
proof −
  obtain D where CT;Map.empty ⊢ MethodInvk (New C0 es) m ds : D
    using assms by (force simp add:subexpr-typing)
  then obtain C0' Cs
    where CT;Map.empty ⊢ (New C0 es) : C0'
    and mt:mtype(CT,m,C0') = Cs → D
    and length ds = length Cs
    by (auto elim:typing.cases)
  with mtype-mbody[OF mt] show ?thesis by (force elim:typing.cases)
qed
```

lemma *closed-subterm-split*:
assumes $CT;\Gamma \vdash e : C$ **and** $\Gamma = Map.empty$
shows
$((\exists\, C0\ es\ fi.\ (FieldProj\ (New\ C0\ es)\ fi) \in subexprs(e))$
$\vee\ (\exists\, C0\ es\ m\ ds.\ (MethodInvk\ (New\ C0\ es)\ m\ ds) \in subexprs(e))$
$\vee\ (\exists\, C0\ D\ es.\ (Cast\ D\ (New\ C0\ es)) \in subexprs(e))$
$\vee\ val(e))$ (**is** *?F e* $\vee$ *?M e* $\vee$ *?C e* $\vee$ *?V e* **is** *?IH e*)
using *assms*
proof(*induct CT* Γ *e C rule:typing-induct*)
case *1* **thus** *?case* **using** *assms* **by** *auto*
next
case (*2 C CT* Γ *x*) **thus** *?case* **by** *auto*
next
case (*3 C0 Ct Cf Ci* Γ *e0 fDef fi*)
have *s1*: *e0* $\in$ *subexprs*(*FieldProj e0 fi*) **by**(*auto simp add:isubexprs.intros*)
from *3* **have** *?IH e0* **by** *auto*
moreover
{ assume *?F e0*
then obtain *C0 es fi'* **where** *s2*: *FieldProj* (*New C0 es*) *fi'* $\in$ *subexprs*(*e0*) **by** *auto*
from *rtrancl-trans*[*OF s2 s1*] **have** *?case* **by** *auto*
} moreover {
assume *?M e0*
then obtain *C0 es m ds* **where** *s2*: *MethodInvk* (*New C0 es*) *m ds* $\in$ *subexprs*(*e0*) **by** *auto*
from *rtrancl-trans*[*OF s2 s1*] **have** *?case* **by** *auto*
} moreover {
assume *?C e0*
then obtain *C0 D es* **where** *s2*: *Cast D* (*New C0 es*) $\in$ *subexprs*(*e0*) **by** *auto*
from *rtrancl-trans*[*OF s2 s1*] **have** *?case* **by** *auto*
} moreover {
assume *?V e0*
then obtain *C0 es* **where** *e0* = (*New C0 es*) **and** *vals*(*es*) **by** (*force elim:val.cases*)
hence *?case* **by**(*force intro:isubexprs.intros*)
}
ultimately show *?case* **by** *blast*
next
case (*4 C C0 CT Cs Ds* Γ *e0 es m*)
have *s1*: *e0* $\in$ *subexprs*(*MethodInvk e0 m es*) **by**(*auto simp add:isubexprs.intros*)
from *4* **have** *?IH e0* **by** *auto*
moreover
{ assume *?F e0*
then obtain *C0 es fi* **where** *s2*: *FieldProj* (*New C0 es*) *fi* $\in$ *subexprs*(*e0*) **by** *auto*
from *rtrancl-trans*[*OF s2 s1*] **have** *?case* **by** *auto*
} moreover {
assume *?M e0*
then obtain *C0 es' m' ds* **where** *s2*: *MethodInvk* (*New C0 es'*) *m' ds* $\in$ *subexprs*(*e0*) **by** *auto*

```
    from rtrancl-trans[OF s2 s1] have ?case by auto
  } moreover {
    assume ?C e0
    then obtain C0 D es where s2: Cast D (New C0 es) ∈ subexprs(e0) by auto
    from rtrancl-trans[OF s2 s1] have ?case by auto
  } moreover {
    assume ?V e0
   then obtain C0 es' where e0 = (New C0 es') and vals(es') by (force elim:val.cases)
    hence ?case by(force intro:isubexprs.intros)
  }
  ultimately show ?case by blast
next
  case (5 C CT Cs Df Ds Γ es)
  hence
    length es = length Cs
    ⋀ i. ⟦i < length es; CT;Γ ⊢ (es!i) : (Cs!i); Γ = Map.empty⟧ ⟹ ?IH (es!i)
    and CT;Γ ⊢+ es : Cs
    by (auto simp add:typings-lengths)
  hence (∃ i < length es. (?F (es!i) ∨ ?M (es!i) ∨ ?C (es!i))) ∨ (vals(es)) (is ?Q
es)
  proof(induct es Cs rule:list-induct2)
    case Nil thus ?Q [] by(auto intro:vals-val.intros)
    next
    case (Cons h t Ch Ct)
      with 5 have h-t-typs: CT;Γ ⊢+ (h#t) : (Ch#Ct)
        and OIH: ⋀ i. ⟦i < length (h#t); CT;Γ ⊢ ((h#t)!i) : ((Ch#Ct)!i); Γ =
Map.empty⟧ ⟹ ?IH ((h#t)!i)
        and G-def: Γ = Map.empty
        by auto
      from h-t-typs have
        h-typ: CT;Γ ⊢ (h#t)!0 : (Ch#Ct)!0
        and t-typs: CT;Γ ⊢+ t : Ct
        by(auto elim:typings.cases)
      { fix i assume i < length t
        hence s-i: Suc i < length (h#t) by auto
       from OIH[OF s-i] have ⟦i < length t; CT;Γ ⊢ (t!i) : (Ct!i); Γ = Map.empty⟧
⟹ ?IH (t!i) by auto }
      with t-typs have ?Q t using Cons by auto
      moreover {
        assume ∃ i < length t. (?F (t!i) ∨ ?M (t!i) ∨ ?C (t!i))
        then obtain i
          where i < length t
          and ?F (t!i) ∨ ?M (t!i) ∨ ?C (t!i) by force
        hence (Suc i < length (h#t)) ∧ (?F ((h#t)!(Suc i)) ∨ ?M ((h#t)!(Suc i))
∨ ?C ((h#t)!(Suc i))) by auto
       hence ∃ i < length (h#t). (?F ((h#t)!i) ∨ ?M ((h#t)!i) ∨ ?C ((h#t)!i)) ..
        hence ?Q (h#t) by auto
      } moreover {
        assume v-t: vals(t)
```

```
      from OIH[OF - h-typ G-def] have ?IH h by auto
      moreover
      { assume ?F h ∨ ?M h ∨ ?C h
        hence ?F ((h#t)!0) ∨ ?M ((h#t)!0) ∨ ?C ((h#t)!0) by auto
        hence ?Q (h#t) by force
      } moreover {
        assume ?V h
        with v-t have vals((h#t)) by (force intro:vals-val.intros)
        hence ?Q(h#t) by auto
      } ultimately have ?Q(h#t) by blast
    } ultimately show ?Q(h#t) by blast
  qed
  moreover {
    assume ∃ i<length es. ?F (es!i) ∨ ?M (es!i) ∨ ?C(es!i)
     then obtain i where i-len: i < length es and r: ?F (es!i) ∨ ?M (es!i) ∨
?C(es!i) by force
      from ith-mem[OF i-len] have s1:es!i ∈ subexprs(New C es) by(auto in-
tro:isubexprs.se-newarg)
    { assume ?F (es!i)
      then obtain C0 es' fi where s2: FieldProj (New C0 es') fi ∈ subexprs(es!i)
by auto
      from rtrancl-trans[OF s2 s1] have ?F(New C es) ∨ ?M(New C es) ∨ ?C(New
C es) by auto
    } moreover {
      assume ?M (es!i)
       then obtain C0 es' m' ds where s2: MethodInvk (New C0 es') m' ds ∈
subexprs(es!i) by force
      from rtrancl-trans[OF s2 s1] have ?F(New C es) ∨ ?M(New C es) ∨ ?C(New
C es) by auto
    } moreover {
      assume ?C (es!i)
       then obtain C0 D es' where s2: Cast D (New C0 es') ∈ subexprs(es!i) by
auto
      from rtrancl-trans[OF s2 s1] have ?F(New C es) ∨ ?M(New C es) ∨ ?C(New
C es) by auto
    } ultimately have ?F(New C es) ∨ ?M(New C es) ∨ ?C(New C es) using r
by blast
    hence ?case by auto
  } moreover {
    assume vals(es)
    hence ?case by(auto intro:vals-val.intros)
  } ultimately show ?case by blast
next
  case (6 C CT D Γ e0)
  have s1: e0 ∈ subexprs(Cast C e0) by(auto simp add:isubexprs.intros)
from 6 have ?IH e0 by auto
moreover
{ assume ?F e0
   then obtain C0 es fi where s2: FieldProj (New C0 es) fi ∈ subexprs(e0) by
```

```
auto
    from rtrancl-trans[OF s2 s1] have ?case by auto
  } moreover {
    assume ?M e0
   then obtain C0 es m ds where s2: MethodInvk (New C0 es) m ds ∈ subexprs(e0)
by auto
    from rtrancl-trans[OF s2 s1] have ?case by auto
  } moreover {
    assume ?C e0
    then obtain C0 D' es where s2: Cast D' (New C0 es) ∈ subexprs(e0) by auto
    from rtrancl-trans[OF s2 s1] have ?case by auto
  } moreover {
    assume ?V e0
   then obtain C0 es' where e0 = (New C0 es') and vals(es') by (force elim:val.cases)
    hence ?case by(force intro:isubexprs.intros)
  }
  ultimately show ?case by blast
next
  case (7 C CT D Γ e0)
  have s1: e0 ∈ subexprs(Cast C e0) by(auto simp add:isubexprs.intros)
  from 7 have ?IH e0 by auto
  moreover
  { assume ?F e0
     then obtain C0 es fi where s2: FieldProj (New C0 es) fi ∈ subexprs(e0) by
auto
    from rtrancl-trans[OF s2 s1] have ?case by auto
  } moreover {
    assume ?M e0
   then obtain C0 es m ds where s2: MethodInvk (New C0 es) m ds ∈ subexprs(e0)
by auto
    from rtrancl-trans[OF s2 s1] have ?case by auto
  } moreover {
    assume ?C e0
    then obtain C0 D' es where s2: Cast D' (New C0 es) ∈ subexprs(e0) by auto
    from rtrancl-trans[OF s2 s1] have ?case by auto
  } moreover {
    assume ?V e0
   then obtain C0 es' where e0 = (New C0 es') and vals(es') by (force elim:val.cases)
    hence ?case by(force intro:isubexprs.intros)
  }
  ultimately show ?case by blast
next
  case (8 C CT D Γ e0)
  have s1: e0 ∈ subexprs(Cast C e0) by(auto simp add:isubexprs.intros)
  from 8 have ?IH e0 by auto
  moreover
  { assume ?F e0
     then obtain C0 es fi where s2: FieldProj (New C0 es) fi ∈ subexprs(e0) by
auto
```

from *rtrancl-trans*[*OF s2 s1*] **have** *?case* **by** *auto*
} **moreover** {
assume *?M e0*
then obtain *C0 es m ds* **where** *s2*: *MethodInvk* (*New C0 es*) *m ds* ∈ *subexprs*(*e0*)
by *auto*
from *rtrancl-trans*[*OF s2 s1*] **have** *?case* **by** *auto*
} **moreover** {
assume *?C e0*
then obtain *C0 D′ es* **where** *s2*: *Cast D′* (*New C0 es*) ∈ *subexprs*(*e0*) **by** *auto*
from *rtrancl-trans*[*OF s2 s1*] **have** *?case* **by** *auto*
} **moreover** {
assume *?V e0*
then obtain *C0 es′* **where** *e0* = (*New C0 es′*) **and** *vals*(*es′*) **by** (*force elim*:*val.cases*)
hence *?case* **by**(*force intro*:*isubexprs.intros*)
}
ultimately show *?case* **by** *blast*
qed

3.9 Type Soundness Theorem

theorem *Thm-2-4-3*:
assumes *e-typ*: *CT*;*Map.empty* ⊢ *e* : *C*
and *ct-ok*: *CT OK*
and *multisteps*: *CT* ⊢ *e* →∗ *e1*
and *no-step*: ¬(∃ *e2*. *CT* ⊢ *e1* → *e2*)
shows (*val*(*e1*) ∧ (∃ *D*. *CT*;*Map.empty* ⊢ *e1* : *D* ∧ *CT* ⊢ *D* <: *C*))
∨ (∃ *D C es*. (*Cast D* (*New C es*) ∈ *subexprs*(*e1*) ∧ *CT* ⊢ *C* ¬<: *D*))
proof −
from *assms Cor-2-4-1-multi*[*OF multisteps ct-ok e-typ*] **obtain** *C1*
where *e1-typ*: *CT*;*Map.empty* ⊢ *e1* : *C1*
and *C1-sub-C*: *CT* ⊢ *C1* <: *C* **by** *auto*
from *e1-typ* **have** ((∃ *C0 es fi*. (*FieldProj* (*New C0 es*) *fi*) ∈ *subexprs*(*e1*))
∨ (∃ *C0 es m ds*. (*MethodInvk* (*New C0 es*) *m ds*) ∈ *subexprs*(*e1*))
∨ (∃ *C0 D es*. (*Cast D* (*New C0 es*)) ∈ *subexprs*(*e1*))
∨ *val*(*e1*)) (**is** *?F e1* ∨ *?M e1* ∨ *?C e1* ∨ *?V e1*) **by** (*simp add*: *closed-subterm-split*)
moreover
{ **assume** *?F e1*
then obtain *C0 es fi* **where** *fp*: *FieldProj* (*New C0 es*) *fi* ∈ *subexprs*(*e1*) **by** *auto*
then obtain *Ci* **where** *CT*;*Map.empty* ⊢ *FieldProj* (*New C0 es*) *fi* : *Ci* **using**
e1-typ **by**(*force simp add*:*subexpr-typing*)
then obtain *C0′* **where** *new-typ*: *CT*;*Map.empty* ⊢ *New C0 es* : *C0′* **by** (*force*
elim: *typing.cases*)
hence *C0* = *C0′* **by** (*auto elim*:*typing.cases*)
with *new-typ* **obtain** *Df* **where** *f1*: *fields*(*CT*,*C0*) = *Df* **and** *lens*: *length es* =
length Df **by**(*auto elim*:*typing.cases*)
from *Thm-2-4-2-1*[*OF e1-typ fp*] **obtain** *Cf fDef*
where *f2*: *fields*(*CT*,*C0*) = *Cf*
and *lkup*: *lookup Cf* (λ*fd*. *vdName fd* = *fi*) = *Some*(*fDef*) **by** *force*
moreover from *fields-functional*[*OF f1 ct-ok f2*] *lens* **have** *length es* = *length Cf*

by *auto*
moreover from *lookup-index*[*OF lkup*] **obtain** *i* **where**
i<*length Cf*
and *fDef* = *Cf* ! *i*
and (*length Cf* = *length es*) ⟶ *lookup2 Cf es* (λ*fd*. *vdName fd* = *fi*) = *Some* (*es* ! *i*) **by** *auto*
ultimately have *lookup2 Cf es* (λ*fd*. *vdName fd* = *fi*) = *Some* (*es*!*i*) **by** *auto*
with *f2* **have** *CT* ⊢ *FieldProj*(*New C0 es*) *fi* → (*es*!*i*) **by**(*auto intro*:*reduction.intros*)
with *fp* **have** ∃ *e2*. *CT* ⊢ *e1* → *e2* **by**(*simp add*:*subexpr-reduct*)
with *no-step* **have** *?thesis* **by** *auto*
} **moreover** {
assume *?M e1*
then obtain *C0 es m ds* **where** *mi*:*MethodInvk* (*New C0 es*) *m ds* ∈ *subexprs*(*e1*) **by** *auto*
then obtain *D* **where** *CT*;*Map.empty* ⊢ *MethodInvk* (*New C0 es*) *m ds* : *D* **using** *e1-typ* **by**(*force simp add*:*subexpr-typing*)
then obtain *C0*′ *Es E*
where *m-typ*: *CT*;*Map.empty* ⊢ *New C0 es* : *C0*′
and *mtype*(*CT*,*m*,*C0*′) = *Es* → *E*
and *length ds* = *length Es*
by (*auto elim*:*typing.cases*)
from *Thm-2-4-2-2*[*OF e1-typ mi*] **obtain** *xs e0* **where** *mb*: *mbody*(*CT*, *m*, *C0*) = *xs* . *e0* **and** *length xs* = *length ds* **by** *auto*
hence *CT* ⊢ (*MethodInvk* (*New C0 es*) *m ds*) → (*substs*[*xs*[↦]*ds*,*this*↦(*New C0 es*)]*e0*) **by**(*auto simp add*:*reduction.intros*)
with *mi* **have** ∃ *e2*. *CT* ⊢ *e1* → *e2* **by**(*simp add*:*subexpr-reduct*)
with *no-step* **have** *?thesis* **by** *auto*
} **moreover** {
assume *?C e1*
then obtain *C0 D es* **where** *c-def*: *Cast D* (*New C0 es*) ∈ *subexprs*(*e1*) **by** *auto*
then obtain *D*′ **where** *CT*;*Map.empty* ⊢ *Cast D* (*New C0 es*) : *D*′ **using** *e1-typ* **by** (*force simp add*:*subexpr-typing*)
then obtain *C0*′ **where** *new-typ*: *CT*;*Map.empty* ⊢ *New C0 es* : *C0*′ **and** *D-eq-D*′: *D* = *D*′ **by** (*auto elim*:*typing.cases*)
hence *C0-eq-C0*′: *C0* = *C0*′ **by**(*auto elim*:*typing.cases*)
hence *?thesis* **proof**(*cases CT* ⊢ *C0* <: *D*)
case *True*
hence *CT* ⊢ *Cast D* (*New C0 es*) → (*New C0 es*) **by**(*auto simp add*:*reduction.intros*)
with *c-def* **have** ∃ *e2*. *CT* ⊢ *e1* → *e2* **by** (*simp add*:*subexpr-reduct*)
with *no-step* **show** *?thesis* **by** *auto*
next
case *False*
with *c-def* **show** *?thesis* **by** *auto*
qed
} **moreover** {
assume *?V e1*
hence *?thesis* **using** *assms* **by**(*auto simp add*:*Cor-2-4-1-multi*)
} **ultimately show** *?thesis* **by** *blast*
qed

end

theory *Execute*
imports *FJSound*
begin

4 Executing FeatherweightJava programs

We execute FeatherweightJava programs using the predicate compiler.

code-pred (*modes*: *i* => *i* => *i* => *bool*,
i => *i* => *o* => *bool as supertypes-of*) *subtyping* **.**

thm *subtyping.equation*

The reduction relation requires that we inverse the (@) function. Therefore, we define a new predicate append and derive introduction rules.

definition *append* **where** *append xs ys zs* = (*zs* = *xs* @ *ys*)

lemma [*code-pred-intro*]: *append* [] *xs xs*
unfolding *append-def* **by** *simp*

lemma [*code-pred-intro*]: *append xs ys zs* ⟹ *append* (*x*#*xs*) *ys* (*x*#*zs*)
unfolding *append-def* **by** *simp*

With this at hand, we derive new introduction rules for the reduction relation:

lemma *rc-invk-arg'*: *CT* ⊢ *ei* → *ei'* ⟹ *append el* (*ei* # *er*) *e'* ⟹ *append el* (*ei'* # *er*) *e''* ⟹
CT ⊢ *MethodInvk e m e'* → *MethodInvk e m e''*
unfolding *append-def* **by** *simp* (*rule reduction.intros*(*6*))

lemma *rc-new-arg'*: *CT* ⊢ *ei* → *ei'* ⟹ *append el* (*ei* # *er*) *e* ⟹ *append el* (*ei'* # *er*) *e'*
==> *CT* ⊢ *New C e* → *New C e'*
unfolding *append-def* **by** *simp* (*rule reduction.intros*(*7*))

lemmas [*code-pred-intro*] = *reduction.intros*(*1*−*5*)
rc-invk-arg' rc-new-arg' reduction.intros(*8*)

code-pred (*modes*: *i* => *i* => *i* => *bool*, *i* => *i* => *o* => *bool as reduce*) *reduction*
proof −
case *append*
from *this* **show** *thesis*
unfolding *append-def* **by** (*cases xa*) *fastforce*+

next
case *reduction*
from *reduction.prems* **show** *thesis*
proof (*cases rule*: *reduction.cases*)
case *r-field*
with *reduction*(*1*) **show** *thesis* **by** *fastforce*
next
case *r-invk*
with *reduction*(*2*) **show** *thesis* **by** *fastforce*
next
case *r-cast*
with *reduction*(*3*) **show** *thesis* **by** *fastforce*
next
case *rc-field*
with *reduction*(*4*) **show** *thesis* **by** *fastforce*
next
case *rc-invk-recv*
with *reduction*(*5*) **show** *thesis* **by** *fastforce*
next
case *rc-invk-arg*
with *reduction*(*6*) **show** *thesis*
unfolding *append-def* **by** *fastforce*
next
case *rc-new-arg*
with *reduction*(*7*) **show** *thesis*
unfolding *append-def* **by** *fastforce*
next
case *rc-cast*
with *reduction*(*8*) **show** *thesis* **by** *fastforce*
qed
qed

thm *reduction.equation*

code-pred *reductions* **.**

thm *reductions.equation*

We also make the class typing executable: this requires that we derive rules for *method-typing*.

definition *method-typing-aux*
where
method-typing-aux CT m D Cs C = (¬ (∀ *Ds D0*. *mtype*(*CT*,*m*,*D*) = *Ds* → *D0* ⟶ *Cs* = *Ds* ∧ *C* = *D0*))

lemma *method-typing-aux*:
(∀ *Ds D0*. *mtype*(*CT*,*m*,*D*) = *Ds* → *D0* ⟶ *Cs* = *Ds* ∧ *C* = *D0*) = (¬ *method-typing-aux CT m D Cs C*)
unfolding *method-typing-aux-def* **by** *auto*

lemma [*code-pred-intro*]:
mtype(*CT*,*m*,*D*) = *Ds* → *D0* ⟹ *Cs* ≠ *Ds* ⟹ *method-typing-aux CT m D Cs C*
unfolding *method-typing-aux-def* **by** *auto*

lemma [*code-pred-intro*]:
mtype(*CT*,*m*,*D*) = *Ds* → *D0* ⟹ *C* ≠ *D0* ⟹ *method-typing-aux CT m D Cs C*
unfolding *method-typing-aux-def* **by** *auto*

declare *method-typing.intros*[*unfolded method-typing-aux*, *code-pred-intro*]

declare *class-typing.intros*[*unfolded append-def*[*symmetric*], *code-pred-intro*]

code-pred (*modes*: *i* => *i* => *bool*) *class-typing*
proof −
case *class-typing*
from *class-typing.cases*[*OF class-typing.prems*, *of thesis*] *this*(*1*) **show** *thesis*
unfolding *append-def* **by** *fastforce*
next
case *method-typing*
from *method-typing.cases*[*OF method-typing.prems*, *of thesis*] *this*(*1*) **show** *thesis*
unfolding *append-def method-typing-aux-def* **by** *fastforce*
next
case *method-typing-aux*
from *this* **show** *thesis*
unfolding *method-typing-aux-def* **by** *auto*
qed

4.1 A simple example

We now execute a simple FJ example program:

abbreviation *A* :: *className*
where *A* == *Suc 0*

abbreviation *B* :: *className*
where *B* == *2*

abbreviation *cPair* :: *className*
where *cPair* == *3*

definition *classA-Def* :: *classDef*
where
classA-Def = (| *cName* = *A*, *cSuper* = *Object*, *cFields* = [], *cConstructor* = (|*kName* = *A*, *kParams* = [], *kSuper* = [], *kInits* = []|), *cMethods* = []|)

definition
classB-Def = (| *cName* = *B*, *cSuper* = *Object*, *cFields* = [], *cConstructor* =

$(\!|kName = B,\ kParams = [],\ kSuper = [],\ kInits = []|\!),\ cMethods = []|\!)$

abbreviation *ffst* :: *varName*
where
ffst == *4*

abbreviation *fsnd* :: *varName*
where
fsnd == *5*

abbreviation *setfst* :: *methodName*
where
setfst == *6*

abbreviation *newfst* :: *varName*
where
newfst == *7*

definition *classPair-Def* :: *classDef*
where
classPair-Def = (| *cName* = *cPair*, *cSuper* = *Object*,
cFields = [(| *vdName* = *ffst*, *vdType* = *Object* |), (| *vdName* = *fsnd*, *vdType* = *Object* |)],
cConstructor = (| *kName* = *cPair*, *kParams* = [(| *vdName* = *ffst*, *vdType* = *Object* |), (| *vdName* = *fsnd*, *vdType* = *Object* |)], *kSuper* = [], *kInits* = [*ffst*, *fsnd*]|)
,
cMethods = [(| *mReturn* = *cPair*, *mName* = *setfst*, *mParams* = [(|*vdName* = *newfst*, *vdType* = *Object* |)],
mBody = *New cPair* [*Var newfst*, *FieldProj* (*Var this*) *fsnd*] |)]|)

definition *exampleProg* :: *classTable*
where *exampleProg* = (((%*x*. *None*)(*A* := *Some classA-Def*))(*B* := *Some classB-Def*))(*cPair* := *Some classPair-Def*)

value *exampleProg* ⊢ *classA-Def OK*
value *exampleProg* ⊢ *classB-Def OK*
value *exampleProg* ⊢ *classPair-Def OK*

values {*x*. *exampleProg* ⊢ *MethodInvk* (*New cPair* [*New A* [], *New B* []]) *setfst* [*New B* []] →∗ *x*}
values {*x*. *exampleProg* ⊢ *FieldProj* (*FieldProj* (*New cPair* [*New cPair* [*New A* [], *New B* []], *New A* []]) *ffst*) *fsnd* →∗ *x*}

end
theory *Featherweight-Java*
imports *FJSound Execute*

begin

end

References

[1] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.

[2] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283. 2002. http://www.in.tum.de/~nipkow/LNCS2283/.