

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	3
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	6
1.8	fields Relation	6
1.9	mtype Relation	6
1.10	mbody Relation	7
1.11	Typing Relation	8
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	45

1 FJDefs: Basic Definitions

```
theory FJDefs imports Main
```

```
begin
```

```
lemmas in-set-code[code-unfold] = mem-iff[symmetric, THEN eq-reflection]
```

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

```
types varName = nat
types methodName = nat
types className = nat
record varDef =
  vdName :: varName
  vdType :: className
types varCtx = varName  $\rightarrow$  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

types *classTable* = *className* \rightarrow *classDef*

1.2 Sub-expression Relation

The sub-expression relation, written $t \in \text{subexprs}(s)$, 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* ($- \in \text{isubexprs}'(-)$ [80,80] 80)

where

$e' \in \text{isubexprs}(e) \equiv (e',e) \in \text{isubexprs}$
| *se-field* : $e \in \text{isubexprs}(\text{FieldProj } e \text{ fi})$
| *se-invkrecv* : $e \in \text{isubexprs}(\text{MethodInvk } e \text{ m es})$
| *se-invkarg* : $\llbracket ei \in \text{set es} \rrbracket \Longrightarrow ei \in \text{isubexprs}(\text{MethodInvk } e \text{ m es})$
| *se-newarg* : $\llbracket ei \in \text{set es} \rrbracket \Longrightarrow ei \in \text{isubexprs}(\text{New } C \text{ es})$
| *se-cast* : $e \in \text{isubexprs}(\text{Cast } C \text{ e})$

abbreviation

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

1.3 Values

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

inductive

vals :: [*exp list*] \Rightarrow *bool* (*vals'*(-) [80] 80)
and *val* :: [*exp*] \Rightarrow *bool* (*val'*(-) [80] 80)

where

vals-nil : *vals*([])
| *vals-cons* : $\llbracket \text{val}(vh); \text{vals}(vt) \rrbracket \Longrightarrow \text{vals}((vh \# vt))$
| *val* : $\llbracket \text{vals}(vs) \rrbracket \Longrightarrow \text{val}(\text{New } C \text{ vs})$

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 $[ds/xs]es$ respectively.

consts

subst :: (*varName* \rightarrow *exp*) \Rightarrow *exp* \Rightarrow *exp*
subst-list1 :: (*varName* \rightarrow *exp*) \Rightarrow *exp list* \Rightarrow *exp list*
subst-list2 :: (*varName* \rightarrow *exp*) \Rightarrow *exp list* \Rightarrow *exp list*

primrec

$$\begin{aligned}
\text{substs } \sigma \text{ (Var } x) &= && (\text{case } (\sigma(x)) \text{ of None } \Rightarrow (\text{Var } x) \mid \text{Some } p \Rightarrow p) \\
\text{substs } \sigma \text{ (FieldProj } e \ f) &= && \text{FieldProj (substs } \sigma \ e) \ f \\
\text{substs } \sigma \text{ (MethodInvk } e \ m \ es) &= && \text{MethodInvk (substs } \sigma \ e) \ m \ (\text{subst-list1 } \sigma \ es) \\
\text{substs } \sigma \text{ (New } C \ es) &= && \text{New } C \ (\text{subst-list2 } \sigma \ es) \\
\text{substs } \sigma \text{ (Cast } C \ e) &= && \text{Cast } C \ (\text{substs } \sigma \ e) \\
\text{subst-list1 } \sigma \ [] &= && [] \\
\text{subst-list1 } \sigma \ (h \ # \ t) &= && (\text{substs } \sigma \ h) \ # \ (\text{subst-list1 } \sigma \ t) \\
\text{subst-list2 } \sigma \ [] &= && [] \\
\text{subst-list2 } \sigma \ (h \ # \ t) &= && (\text{substs } \sigma \ h) \ # \ (\text{subst-list2 } \sigma \ t)
\end{aligned}$$
abbreviation

$$\begin{aligned}
\text{substs-syn} &:: [\text{exp list}] \Rightarrow [\text{varName list}] \Rightarrow [\text{exp}] \Rightarrow \text{exp} \\
&(\text{'[-'/-']- [80,80,80] 80}) \textbf{ where} \\
&[\text{ds/xs}]e \equiv \text{substs (map-upds empty xs ds)} \ e
\end{aligned}$$
abbreviation

$$\begin{aligned}
\text{subst-list-syn} &:: [\text{exp list}] \Rightarrow [\text{varName list}] \Rightarrow [\text{exp list}] \Rightarrow \text{exp list} \\
&(\text{'[-'/-']- [80,80,80] 80}) \textbf{ where} \\
&[\text{ds/xs}]es \equiv \text{map (substs (map-upds empty xs ds)) } \ es
\end{aligned}$$

1.5 Lookup

The function *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 \Rightarrow ('a \Rightarrow bool) \Rightarrow 'a option
where

$$\begin{aligned}
&\text{lookup } [] \ P = \text{None} \\
&| \text{lookup } (h\#\#t) \ P = (\text{if } P \ h \ \text{then } \text{Some } h \ \text{else } \text{lookup } \ t \ P)
\end{aligned}$$

primrec *lookup2* :: 'a list \Rightarrow 'b list \Rightarrow ('a \Rightarrow bool) \Rightarrow 'b option
where

$$\begin{aligned}
&\text{lookup2 } [] \ l2 \ P = \text{None} \\
&| \text{lookup2 } (h1\#\#t1) \ l2 \ P = (\text{if } P \ h1 \ \text{then } \text{Some}(hd \ l2) \ \text{else } \text{lookup2 } \ t1 \ (tl \ l2) \ P)
\end{aligned}$$

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

$$\begin{aligned}
\text{varDefs-names} &:: \text{varDef list} \Rightarrow \text{varName list} \textbf{ where} \\
\text{varDefs-names} &= \text{map } \text{vdName}
\end{aligned}$$
definition

$$\begin{aligned}
\text{varDefs-types} &:: \text{varDef list} \Rightarrow \text{className list} \textbf{ where} \\
\text{varDefs-types} &= \text{map } \text{vdType}
\end{aligned}$$

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

$subtyping :: [classTable, className, className] \Rightarrow bool \ (- \vdash - <: - \ [80,80,80] \ 80)$

where

$s-refl : CT \vdash C <: C$
 $| s-trans : \llbracket CT \vdash C <: D; CT \vdash D <: E \rrbracket \Longrightarrow CT \vdash C <: E$
 $| s-super : \llbracket CT(C) = Some(CDef); cSuper CDef = D \rrbracket \Longrightarrow CT \vdash C <: D$

abbreviation

$neg-subtyping :: [classTable, className, className] \Rightarrow bool \ (- \vdash - \neg <: - \ [80,80,80] \ 80)$

where $CT \vdash S \neg <: T \equiv \neg CT \vdash S <: T$

inductive

$subtypings :: [classTable, className list, className list] \Rightarrow bool \ (- \vdash + - <: - \ [80,80,80] \ 80)$

where

$ss-nil : CT \vdash + [] <: []$
 $| ss-cons : \llbracket CT \vdash C_0 <: D_0; CT \vdash + Cs <: Ds \rrbracket \Longrightarrow CT \vdash + (C_0 \# Cs) <: (D_0 \# Ds)$

1.8 fields Relation

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

inductive

$fields :: [classTable, className, varDef list] \Rightarrow bool \ (fields'(-, -) = - \ [80,80,80] \ 80)$

where

$f-obj:$
 $fields(CT, Object) = []$
 $| f-class:$
 $\llbracket CT(C) = Some(CDef); cSuper CDef = D; cFields CDef = Cf; fields(CT, D) = Dg; DgCf = Dg @ Cf \rrbracket$
 $\Longrightarrow fields(CT, C) = DgCf$

1.9 mtype Relation

The `mtype` relation, written $mtype(CT, m, C) = Cs \rightarrow C_0$ relates a class C , method name m , and the arrow type $Cs \rightarrow C_0$. 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

$mtype :: [classTable, methodName, className, className list, className] \Rightarrow bool$
 $(mtype'(-,-,-) = - \rightarrow - [80,80,80,80] 80)$

where

mt-class:

$\llbracket CT(C) = Some(CDef);$
 $lookup (cMethods CDef) (\lambda md.(methodName md = m)) = Some(mDef);$
 $varDefs-types (mParams mDef) = Bs;$
 $mReturn mDef = B \rrbracket$
 $\implies mtype(CT,m,C) = Bs \rightarrow B$

| *mt-super:*

$\llbracket CT(C) = Some (CDef);$
 $lookup (cMethods CDef) (\lambda md.(methodName md = m)) = None;$
 $cSuper CDef = D;$
 $mtype(CT,m,D) = Bs \rightarrow B \rrbracket$
 $\implies mtype(CT,m,C) = Bs \rightarrow B$

1.10 mbody Relation

The $mtype$ relation, written $mbody(CT, m, C) = xs.e_0$ 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

$mbody :: [classTable, methodName, className, varName list, exp] \Rightarrow bool$ ($mbody'(-,-,-)$
 $= - . - [80,80,80,80] 80)$

where

mb-class:

$\llbracket CT(C) = Some(CDef);$
 $lookup (cMethods CDef) (\lambda md.(methodName md = m)) = Some(mDef);$
 $varDefs-names (mParams mDef) = xs;$
 $mBody mDef = e \rrbracket$
 $\implies mbody(CT,m,C) = xs . e$

| *mb-super:*

$\llbracket CT(C) = Some(CDef);$
 $lookup (cMethods CDef) (\lambda md.(methodName md = m)) = None;$
 $cSuper CDef = D;$
 $mbody(CT,m,D) = xs . e \rrbracket$
 $\implies mbody(CT,m,C) = xs . e$

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

$typings :: [classTable, varCtx, exp list, className list] \Rightarrow bool$ (-;- $\vdash +$ - : - [80,80,80,80] 80)

and $typing :: [classTable, varCtx, exp, className] \Rightarrow bool$ (-;- \vdash - : - [80,80,80,80] 80)

where

$ts-nil : CT; \Gamma \vdash + [] : []$

| $ts-cons$:
 $\llbracket CT; \Gamma \vdash e0 : C0; CT; \Gamma \vdash + es : Cs \rrbracket$
 $\implies CT; \Gamma \vdash + (e0 \# es) : (C0 \# Cs)$

| $t-var$:
 $\llbracket \Gamma(x) = Some\ C \rrbracket \implies CT; \Gamma \vdash (Var\ x) : C$

| $t-field$:
 $\llbracket CT; \Gamma \vdash e0 : C0;$
 $fields(CT, C0) = Cf;$
 $lookup\ Cf\ (\lambda fd. (vdName\ fd = fi)) = Some(fDef);$
 $vdType\ fDef = Ci \rrbracket$
 $\implies CT; \Gamma \vdash FieldProj\ e0\ fi : Ci$

| $t-invk$:
 $\llbracket CT; \Gamma \vdash e0 : C0;$
 $mtype(CT, m, C0) = Ds \rightarrow C;$
 $CT; \Gamma \vdash + es : Cs;$
 $CT \vdash + Cs <: Ds;$
 $length\ es = length\ Ds \rrbracket$
 $\implies CT; \Gamma \vdash MethodInvk\ e0\ m\ es : C$

| $t-new$:
 $\llbracket fields(CT, C) = Df;$
 $length\ es = length\ Df;$
 $varDefs-types\ Df = Ds;$
 $CT; \Gamma \vdash + es : Cs;$
 $CT \vdash + Cs <: Ds \rrbracket$
 $\implies CT; \Gamma \vdash New\ C\ es : C$

| $t-ucast$:
 $\llbracket CT; \Gamma \vdash e0 : D;$
 $CT \vdash D <: C \rrbracket$
 $\implies CT; \Gamma \vdash Cast\ C\ e0 : C$

| $t-dcast$:

$$\begin{aligned} & \llbracket CT; \Gamma \vdash e0 : D; \\ & \quad CT \vdash C <: D; C \neq D \rrbracket \\ & \implies CT; \Gamma \vdash \text{Cast } C \ e0 : C \end{aligned}$$

| *t-scst* :

$$\begin{aligned} & \llbracket CT; \Gamma \vdash e0 : D; \\ & \quad CT \vdash C \neg <: D; \\ & \quad CT \vdash D \neg <: C \rrbracket \\ & \implies CT; \Gamma \vdash \text{Cast } C \ e0 : C \end{aligned}$$

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 $\bigwedge C \ CT \ \Gamma \ x. \ \Gamma \ x = \text{Some } C \implies P \ CT \ \Gamma \ (Var \ x) \ C$
and $\bigwedge C0 \ CT \ Cf \ Ci \ \Gamma \ e0 \ fDef \ fi. \ \llbracket CT; \Gamma \vdash e0 : C0; P \ CT \ \Gamma \ e0 \ C0; \text{fields}(CT, C0) = Cf; \text{lookup } Cf \ (\lambda fd. \text{vdName } fd = fi) = \text{Some } fDef; \text{vdType } fDef = Ci \rrbracket \implies P \ CT \ \Gamma \ (\text{FieldProj } e0 \ fi) \ Ci$
and $\bigwedge C \ C0 \ CT \ Cs \ Ds \ \Gamma \ e0 \ es \ m. \ \llbracket CT; \Gamma \vdash e0 : C0; P \ CT \ \Gamma \ e0 \ C0; \text{mtype}(CT, m, C0) = Ds \rightarrow C; CT; \Gamma \vdash + \ es : Cs; \bigwedge i. \ \llbracket i < \text{length } es \rrbracket \implies P \ CT \ \Gamma \ (es!i) \ (Cs!i); CT \vdash + \ Cs <: Ds; \text{length } es = \text{length } Ds \rrbracket \implies P \ CT \ \Gamma \ (\text{MethodInvk } e0 \ m \ es) \ C$
and $\bigwedge C \ CT \ Cs \ Df \ Ds \ \Gamma \ es. \ \llbracket \text{fields}(CT, C) = Df; \text{length } es = \text{length } Df; \text{varDefs-types } Df = Ds; CT; \Gamma \vdash + \ es : Cs; \bigwedge i. \ \llbracket i < \text{length } es \rrbracket \implies P \ CT \ \Gamma \ (es!i) \ (Cs!i); CT \vdash + \ Cs <: Ds \rrbracket \implies P \ CT \ \Gamma \ (\text{New } C \ es) \ C$
and $\bigwedge C \ CT \ D \ \Gamma \ e0. \ \llbracket CT; \Gamma \vdash e0 : D; P \ CT \ \Gamma \ e0 \ D; CT \vdash D <: C \rrbracket \implies P \ CT \ \Gamma \ (\text{Cast } C \ e0) \ C$
and $\bigwedge C \ CT \ D \ \Gamma \ e0. \ \llbracket CT; \Gamma \vdash e0 : D; P \ CT \ \Gamma \ e0 \ D; CT \vdash C <: D; C \neq D \rrbracket \implies P \ CT \ \Gamma \ (\text{Cast } C \ e0) \ C$
and $\bigwedge C \ CT \ D \ \Gamma \ e0. \ \llbracket CT; \Gamma \vdash e0 : D; P \ CT \ \Gamma \ e0 \ D; CT \vdash C \neg <: D; CT \vdash D \neg <: C \rrbracket \implies P \ CT \ \Gamma \ (\text{Cast } C \ e0) \ C$
shows $P \ CT \ \Gamma \ e \ C$ (**is** $?P$)

proof –

fix $es \ Cs$
let $?IH = CT; \Gamma \vdash + \ es : Cs \longrightarrow (\forall i < \text{length } es. \ P \ CT \ \Gamma \ (es!i) \ (Cs!i))$
have $?IH \wedge (?T \longrightarrow ?P)$
proof (*induct rule: typings-typing.induct*)
case (*ts-nil* $CT \ \Gamma$) **show** $?case$ **by** *auto*
next
case (*ts-cons* $CT \ \Gamma \ e0 \ C0 \ es \ Cs$)
show $?case$ **proof**
fix i
show $i < \text{length } (e0 \# es) \longrightarrow P \ CT \ \Gamma \ ((e0 \# es)!i) \ ((C0 \# Cs)!i)$ **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-inv 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 $CT \vdash mdOK \text{ IN } C$ 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*] \Rightarrow *bool* (- \vdash - *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) ( $\lambda md. (mName \ md = \ m)$ ) = Some(mDef);
   mReturn mDef = C0; mParams mDef = Cxs; mBody mDef = e0;
   varDefs-types Cxs = Cs;
   varDefs-names Cxs = xs;
    $\Gamma = (\text{map-upds } \text{empty } \text{xs } \text{Cs})(\text{this } \mapsto \text{C})$ ;
   CT;  $\Gamma \vdash e0 : E0$ ;
   CT  $\vdash E0 <: C0$ ;
    $\forall Ds \ D0. (\text{mtype}(\text{CT}, \text{m}, \text{D}) = Ds \rightarrow D0) \longrightarrow (Cs = Ds \wedge C0 = D0)$  ]]
 $\Rightarrow CT \vdash mDef \text{ OK IN } C$ 

```

inductive

method-typings :: [*classTable*, *methodDef list*, *className*] \Rightarrow *bool* (- $\vdash +$ - *OK IN* - [80,80,80] 80)

where

ms-nil :

$CT \vdash + [] \text{ OK IN } C$

| *ms-cons* :

```

[[ CT  $\vdash m \text{ OK IN } C$ ;
   CT  $\vdash + ms \text{ OK IN } C$  ]]

```

$\Rightarrow CT \vdash+ (m \# ms) OK IN C$

1.13 Class Typing Relation

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

inductive

$class\text{-}typing :: [classTable, classDef] \Rightarrow bool (- \vdash - OK [80,80] 80)$

where

$t\text{-}class: \llbracket cName CDef = C;$
 $cSuper CDef = D;$
 $cConstructor CDef = KDef;$
 $cMethods CDef = M;$
 $kName KDef = C;$
 $kParams KDef = (Dg@Cf);$
 $kSuper KDef = varDefs\text{-}names Dg;$
 $kInits KDef = varDefs\text{-}names Cf;$
 $fields(CT,D) = Dg;$
 $CT \vdash+ M OK IN C \rrbracket$

$\Rightarrow CT \vdash CDef OK$

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 well-typed and has name C .

inductive

$ct\text{-}typing :: classTable \Rightarrow bool (- OK 80)$

where

$ct\text{-}all\text{-}ok:$

$\llbracket Object \notin dom(CT);$
 $\forall C CDef. CT(C) = Some(CDef) \longrightarrow (CT \vdash CDef OK) \wedge (cName CDef = C) \rrbracket$

$\Rightarrow CT OK$

1.15 Evaluation Relation

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

inductive

$reduction :: [classTable, exp, exp] \Rightarrow bool (- \vdash - \rightarrow - [80,80,80] 80)$

where

$r\text{-}field:$

$\llbracket fields(CT,C) = Cf;$
 $lookup2 Cf es (\lambda fd.(vdName fd = fi)) = Some(ei) \rrbracket$

$\Rightarrow CT \vdash FieldProj (New C es) fi \rightarrow ei$

```

| r-invok:
  [[ mbody(CT,m,C) = xs . e0;
     subst ((map-upds empty xs ds)(this ↦ (New C es))) e0 = e0' ]]
  ⇒ CT ⊢ MethodInvk (New C es) m ds → e0'

| r-cast:
  [[ CT ⊢ C <: D ]]
  ⇒ CT ⊢ Cast D (New C es) → New C es

| rc-field:
  [[ CT ⊢ e0 → e0' ]]
  ⇒ CT ⊢ FieldProj e0 f → FieldProj e0' f

| rc-invok-recv:
  [[ CT ⊢ e0 → e0' ]]
  ⇒ CT ⊢ MethodInvk e0 m es → MethodInvk e0' m es

| rc-invok-arg:
  [[ CT ⊢ ei → ei' ]]
  ⇒ CT ⊢ MethodInvk e0 m (el@ei#er) → MethodInvk e0 m (el@ei'#er)

| rc-new-arg:
  [[ CT ⊢ ei → ei' ]]
  ⇒ CT ⊢ New C (el@ei#er) → New C (el@ei'#er)

| rc-cast:
  [[ CT ⊢ e0 → e0' ]]
  ⇒ CT ⊢ Cast C e0 → Cast C e0'

inductive
  reductions :: [classTable, exp, exp] ⇒ bool (- ⊢ - →* - [80,80,80] 80)
where
  rs-refl: CT ⊢ e →* e
  | rs-trans: [[ CT ⊢ e → e'; CT ⊢ e' →* e'' ]] ⇒ CT ⊢ e →* 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 ∈ 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  $esh \neq ei$ 
    with Cons have  $ei \in set\ est$  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:  $\bigwedge i. \llbracket i < length\ es \rrbracket \implies es!i \in set\ es$ 
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  $[xs[\mapsto]ys, x\mapsto y] = [(xs@[x])[\mapsto](ys@[y])]$ 
  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  $[xs[\mapsto]As]x = Some\ Ai$ 
  shows  $\exists i. (As!i = Ai)$ 
     $\wedge (i < length\ As)$ 
     $\wedge (\forall (Bs::'c\ list). ((length\ Bs = length\ As)$ 
       $\longrightarrow ([xs[\mapsto]Bs] x = Some\ (Bs\ !i))))$ 
  (is  $\exists i. ?P\ i\ xs\ As$ 
    is  $\exists i. (?P1\ i\ As) \wedge (?P2\ i\ As) \wedge (\forall Bs::('c\ list). (?P3\ i\ xs\ As\ Bs))$ )
using assms proof(induct xs As rule:list-induct2)
assume  $\llbracket [\mapsto] \rrbracket x = Some\ Ai$ 
moreover have  $\neg \llbracket [\mapsto] \rrbracket x = Some\ Ai$  by auto
ultimately show  $\exists i. ?P\ i\ \llbracket \rrbracket$  by contradiction
next
fix  $xa\ xs\ y\ ys$ 
assume length-xs-ys:  $length\ xs = length\ ys$ 
and IH:  $[xs\ [\mapsto]\ ys] x = Some\ Ai \implies \exists i. ?P\ i\ xs\ ys$ 
and map-eq-Some:  $[xa\ \# xs\ [\mapsto]\ y\ \# ys] x = Some\ Ai$ 

```

```

from prems have map-decomp:  $[xa\#xs \mapsto y\#ys] = [xa\mapsto y] ++ [xs\mapsto]ys$  by
fastsimp
from length-xs-ys IH map-eq-Some show  $\exists i. ?P\ i\ (xa\#xs)\ (y\ \# \ ys)$ 
proof(cases  $[xs\mapsto]ys$ )x)
  case(Some Ai')
    hence  $([xa \mapsto y] ++ [xs\mapsto]ys)\ x = \text{Some } Ai'$  by(rule map-add-find-right)
    hence  $P: [xs\mapsto]ys\ x = \text{Some } Ai$  using prems by simp
    from IH[OF P] obtain i where
      R1:  $ys\ !\ i = Ai$ 
      and R2:  $i < \text{length } ys$ 
      and pre-r3:  $\forall (Bs::'c\ \text{list}). ?P3\ i\ xs\ ys\ Bs$  by fastsimp
    { fix Bs::'c list
      assume length-Bs:  $\text{length } Bs = \text{length } (y\ \# \ ys)$ 
      then obtain n where  $\text{length } (y\ \# \ ys) = \text{Suc } n$  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  $\text{length } ys = \text{length } bs$  by simp
      with pre-r3 have  $([xa\mapsto b] ++ [xs\mapsto]bs)\ x = \text{Some } (bs!\ i)$  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 have  $[xa\mapsto y]\ x = \text{Some } Ai$  using prems 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:  $\text{length } Bs = \text{length } (y\ \# \ ys)$ 
      then obtain n where  $\text{length } (y\ \# \ ys) = \text{Suc } n$  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  $\text{length } ys = \text{length } bs$  by simp
      hence  $\text{dom}([xs\mapsto]ys) = \text{dom}([xs\mapsto]bs)$  by auto
      with None have  $[xs\mapsto]bs\ x = \text{None}$  by (auto simp only:domIff)
      moreover from x-eq-xa have sing-map:  $[xa\mapsto b]\ x = \text{Some } b$  by (auto simp
only:map-upd-Some-unfold)
      ultimately have  $([xa\mapsto b] ++ [xs\mapsto]bs)\ x = \text{Some } b$  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* :

$$\forall \sigma. \text{subst-list1 } \sigma\ l = \text{map } (\text{substs } \sigma)\ l$$

by (*induct l, simp-all*)

lemma *subst-list2-eq-map-substs* :
 $\forall \sigma. \text{subst-list2 } \sigma \ l = \text{map } (\text{substs } \sigma) \ l$
by (*induct l, simp-all*)

2.2.2 Lookup

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

lemma *lookup-true*:
 $\text{lookup } l \ f = \text{Some } r \implies f \ r$
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*:
 $\llbracket \text{length } l > 0; f \ (l!0) \rrbracket \implies \text{lookup } l \ f = \text{Some } (l!0)$
by (*induct l*) *auto*

lemma *lookup-split*: $\text{lookup } l \ f = \text{None} \vee (\exists h. \text{lookup } l \ f = \text{Some } h)$
by (*induct l*) *simp-all*

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

```

assume asm: $\neg$  (f h1)
hence lookup t1 f = Some e
  using prems by (auto simp add:lookup.simps)
then obtain i where
  i < length t1
  and e = t1 ! i
  and ih:(length t1 = length (tl l2))  $\longrightarrow$  lookup2 t1 (tl l2) f = Some ((tl l2) !
i)
  using prems by blast
hence Suc i < length (h1 # t1)  $\wedge$  e = (h1 # t1)!(Suc i) using prems 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*:

```

 $\bigwedge l2. \llbracket \text{lookup2 } l1 \ l2 \ f = \text{Some } e; \text{length } l1 = \text{length } l2 \rrbracket \implies \exists i < (\text{length } l2). e = (l2!i) \wedge \text{lookup } l1 \ f = \text{Some } (l1!i)$ 

```

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** *prems* **by** (*auto simp add:lookup2.simps*)

hence *0* < *length (h2 # t2)* \wedge *e* = (*h2 # t2*) ! *0* \wedge *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*: \neg (*f h1*)

hence $\exists i < \text{length } t2. e = t2 ! i \wedge \text{lookup } t1 \ f = \text{Some } (t1 ! i)$ **using** *prems l2-def* **by** *auto*

then obtain *i* **where** *i* < *length t2* \wedge *e* = *t2 ! i* \wedge *lookup t1 f* = *Some (t1 ! i)* **by** *auto*

hence (*Suc i*) < *length(h2 # t2)* \wedge *e* = ((*h2 # t2*) ! (*Suc i*)) \wedge *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 fastsimp
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** *fastsimp+*
hence $c\text{-not-obj}: C \neq \text{Object}$ **by** (*force elim:ct-typing.cases*)
from $f\text{-class}$ **have** $flds: \text{fields}(CT, C) = DgCf'$ **by** *fastsimp*
then obtain Dg' **where**
 $\text{fields}(CT, D) = Dg'$
and $DgCf' = Dg' @ Cf$
using $f\text{-class-inv } c\text{-not-obj}$ **by** (*auto elim:fields.cases*)
hence $Dg' = Dg$ **using** $f\text{-class}$ **by** *auto*
thus $?case$ **using** *prems* **by** *force*
qed

2.2.4 Subtyping and Typing

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

lemma *typings-index*:
assumes $CT; \Gamma \vdash \vdash es:Cs$
shows $\bigwedge i. \llbracket i < \text{length } es \rrbracket \implies CT; \Gamma \vdash (es!i) : (Cs!i)$
proof –
have $\text{length } es = \text{length } Cs$ **using** *assms* **by** (*auto simp: typings-lengths*)
thus $\bigwedge i. \llbracket i < \text{length } es \rrbracket \implies CT; \Gamma \vdash (es!i) : (Cs!i)$
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 $CT \vdash \vdash Cs <: Ds$
shows $\bigwedge i. \llbracket i < \text{length } Cs \rrbracket \implies CT \vdash (Cs!i) <: (Ds!i)$
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 $CT \vdash \vdash Cs <: Ds$
and $CT \vdash C <: D$
shows $CT \vdash \vdash (Cs@[C]) <: (Ds@[D])$
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 $CT; \Gamma \vdash e : C$
shows $CT; \Gamma \vdash (es@[e]) : (Cs@[C])$
proof –
have $length\ es = length\ Cs$ **using** *assms* **by** (*simp-all add:typings-lengths*)
thus $CT; \Gamma \vdash (es@[e]) : (Cs@[C])$ **using** *prems*
proof (*induct es Cs rule:list-induct2*)
have $CT; \Gamma \vdash [] : []$ **by** (*simp add:typings-typing.ts-nil*)
moreover from *assms* **have** $CT; \Gamma \vdash e : C$ **by** *simp*
ultimately show $CT; \Gamma \vdash ([]@[e]) : ([]@[C])$ **by** (*auto simp add:typings-typing.ts-cons*)
next
fix $x\ xs\ y\ ys$
assume $length\ xs = length\ ys$
and $IH: [CT; \Gamma \vdash xs : ys; CT; \Gamma \vdash e : C] \implies CT; \Gamma \vdash (xs @ [e]) : (ys @ [C])$
and $x\text{-}xs\text{-}typs: CT; \Gamma \vdash (x \# xs) : (y \# ys)$
and $e\text{-}typ: CT; \Gamma \vdash e : C$
from $x\text{-}xs\text{-}typs$ **have** $x\text{-}typ: CT; \Gamma \vdash x : y$ **and** $CT; \Gamma \vdash xs : ys$ **by** (*auto elim:typings.cases*)
with $IH\ e\text{-}typ$ **have** $CT; \Gamma \vdash (xs@[e]) : (ys@[C])$ **by** *simp*
with $x\text{-}typ$ **have** $CT; \Gamma \vdash ((x\#xs)@[e]) : ((y\#ys)@[C])$ **by** (*auto simp add:typings-typing.ts-cons*)
thus $CT; \Gamma \vdash ((x \# xs) @ [e]) : ((y \# ys) @ [C])$ **by** (*auto simp add:typings-typing.ts-cons*)
qed
qed

lemma *ith-typing*: $\bigwedge Cs. [CT; \Gamma \vdash (es@(h\#t)) : Cs] \implies CT; \Gamma \vdash h : (Cs!(length\ es))$
proof (*induct es, auto elim:typings.cases*)
qed

lemma *ith-subtyping*: $\bigwedge Ds. [CT \vdash (Cs@(h\#t)) <: Ds] \implies CT \vdash h <: (Ds!(length\ Cs))$
proof (*induct Cs, auto elim:subtypings.cases*)
qed

lemma *subtypings-refl*: $CT \vdash Cs <: Cs$
by (*induct Cs, auto simp add: subtyping.s-refl subtypings.intros*)

lemma *subtypings-trans*: $\bigwedge Ds\ Es. [CT \vdash Cs <: Ds; CT \vdash Ds <: Es] \implies CT \vdash Cs <: Es$
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 $h1:CT \vdash hCs <: hDs$ **and** $t1:CT \vdash tCs <: tDs$ **and** $Ds = hDs \# tDs$
by (*auto elim:subtypings.cases*)
then obtain $hEs \ tEs$
where $h2:CT \vdash hDs <: hEs$ **and** $t2:CT \vdash tDs <: tEs$ **and** $Es = hEs \# tEs$
using *Cons* **by** (*auto elim:subtypings.cases*)
moreover from *subtyping.s-trans[OF h1 h2]* **have** $CT \vdash hCs <: hEs$ **by** *fastsimp*
moreover with $t1 \ t2$ **have** $CT \vdash tCs <: tEs$ **using** *Cons* **by** *simp-all*
ultimately show *?case* **by** (*auto simp add:subtypings.intros*)
qed

lemma *ith-typing-sub*:

$\bigwedge Cs. \llbracket CT; \Gamma \vdash (es @ (h \# t)) : Cs;$
 $CT; \Gamma \vdash h' : Ci';$
 $CT \vdash Ci' <: (Cs!(length \ es)) \rrbracket$
 $\implies \exists Cs'. (CT; \Gamma \vdash (es @ (h' \# t)) : Cs' \wedge CT \vdash Cs' <: Cs)$

proof(*induct es*)

case *Nil*

then obtain $hCs \ tCs$

where $ts: CT; \Gamma \vdash t : tCs$

and $Cs\text{-def}: Cs = hCs \# tCs$ **by**(*auto elim:typings.cases*)

from $Cs\text{-def} \ Nil$ **have** $CT \vdash Ci' <: hCs$ **by** *auto*

with $Cs\text{-def}$ **have** $CT \vdash (Ci' \# tCs) <: Cs$ **by**(*auto simp add:subtypings.ss-cons subtypings-refl*)

moreover from $ts \ Nil$ **have** $CT; \Gamma \vdash (h' \# t) : (Ci' \# tCs)$ **by**(*auto simp add:typings-typing.ts-cons*)

ultimately show *?case* **by** *auto*

next

case (*Cons eh et*)

then obtain $hCs \ tCs$

where $CT; \Gamma \vdash eh : hCs$

and $CT; \Gamma \vdash (et @ (h \# t)) : tCs$

and $Cs\text{-def}: Cs = hCs \# tCs$

by(*auto elim:typings.cases*)

moreover with *Cons* **obtain** tCs'

where $CT; \Gamma \vdash (et @ (h' \# t)) : tCs'$

and $CT \vdash tCs' <: tCs$

by *auto*

ultimately have

$CT; \Gamma \vdash (eh \# (et @ (h' \# t))) : (hCs \# tCs')$

and $CT \vdash (hCs \# tCs') <: Cs$

by(*auto simp add:typings-typing.ts-cons subtypings.ss-cons subtyping.s-refl*)

thus *?case* **by** *auto*

qed

lemma *mem-typings*:

$\bigwedge Cs. \llbracket CT; \Gamma \vdash es:Cs; ei \in set \ es \rrbracket \implies \exists Ci. CT; \Gamma \vdash ei:Ci$

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 $CT; \Gamma \vdash ds : As$
and $CT \vdash As <: Bs$
and $length\ ds = length\ As$
and $length\ ds = length\ Bs$
and $i < length\ ds$
shows $CT; \Gamma \vdash ds!i : As!i$ **and** $CT \vdash As!i <: Bs!i$
using *assms* **by** (*auto simp add:typings-index subtypings-index*)

lemma *subtypings-length*:
 $CT \vdash As <: Bs \implies length\ As = length\ Bs$
by(*induct rule:subtypings.induct*) *simp-all*

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

lemma *not-subtypes*:
assumes $CT \vdash A <: C$
shows $\bigwedge D. \llbracket CT \vdash D \neg<: C; CT \vdash C \neg<: D \rrbracket \implies CT \vdash A \neg<: D$
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*: $CT \vdash Da \neg<: D$ **proof**(*rule ccontr*)
assume $\neg CT \vdash Da \neg<: D$
hence *da-sub-d*: $CT \vdash Da <: D$ **by** *auto*
have *d-sub-e*: $CT \vdash D <: E$ **using** *prems* **by** *fastsimp*
thus *False* **using** *prems* **by** (*force simp add:subtyping.s-trans[OF da-sub-d d-sub-e]*)
qed
have *d-nsub-da*: $CT \vdash D \neg<: Da$ **using** *s-trans* **by** *auto*
from *da-nsub-d d-nsub-da s-trans* **show** $CT \vdash C \neg<: Da$ **by** *auto*
next
case (*s-super* $CT\ C\ CDef\ D\ Da$)
have $C \neq Da$ **proof**(*rule ccontr*)
assume $\neg C \neq Da$
hence $C = Da$ **by** *auto*
hence $CT \vdash Da <: D$ **using** *prems* **by**(*auto simp add: subtyping.s-super*)
thus *False* **using** *prems* **by** *auto*

```

qed
thus ?case using prems by (auto simp add: not-subtypes-aux)
qed

```

2.2.5 Sub-Expressions

```

lemma isubexpr-typing:
  assumes e1 ∈ isubexprs(e0)
  shows  $\bigwedge C. \llbracket CT; \text{empty} \vdash e0 : C \rrbracket \implies \exists D. CT; \text{empty} \vdash e1 : D$ 
using assms
by (induct rule:isubexprs.induct) (auto elim:typing.cases simp add:mem-typings)

```

```

lemma subexpr-typing:
  assumes e1 ∈ subexprs(e0)
  shows  $\bigwedge C. \llbracket CT; \text{empty} \vdash e0 : C \rrbracket \implies \exists D. CT; \text{empty} \vdash e1 : D$ 
using assms
by (induct rule:rtrancl.induct) (auto, force simp add:isubexpr-typing)

```

```

lemma isubexpr-reduct:
  assumes d1 ∈ isubexprs(e1)
  shows  $\bigwedge d2. \llbracket CT \vdash d1 \rightarrow d2 \rrbracket \implies \exists e2. CT \vdash e1 \rightarrow e2$ 
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  $\bigwedge d2. \llbracket CT \vdash d1 \rightarrow d2 \rrbracket \implies \exists e2. CT \vdash e1 \rightarrow e2$ 
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  $mtype(CT, m, C) = Cs \rightarrow C0$ 
shows  $\exists xs\ e. mbody(CT, m, C) = xs . e \wedge length\ xs = length\ Cs$ 
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  $mt: mtype(CT, m, C) = Cs \rightarrow C0$ 
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  $CT \vdash C <: D$  and CT OK
shows  $(mtype(CT, m, D) = Cs \rightarrow C0) \implies (mtype(CT, m, C) = Cs \rightarrow C0)$ 
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  $CT \vdash CDef\ OK$  and  $cName\ CDef = C$ 
  by (auto elim: ct-typing.cases)
  with s-super obtain M
  where  $CT \vdash\ +\ M\ OK\ IN\ C$  and  $cMethods\ CDef = M$ 
  by (auto elim: class-typing.cases)
  let  $?lookup-m = lookup\ M\ (\lambda md. (mName\ md = m))$ 
  show ?case
  proof (cases  $\exists mDef. ?lookup-m = Some\ mDef$ )
  case True
    then obtain mDef where  $?lookup-m = Some\ mDef$  by (rule exE)

```

```

    hence  $mDef.name: mName\ mDef = m$  by (rule lookup-true)
  have  $CT \vdash mDef\ OK\ IN\ C$  using prems by (auto simp add: method-typings-lookup)
  then obtain  $CDef'\ m'\ D'\ Cs'\ C0'$ 
    where  $CT\ C = Some\ CDef'$ 
      and  $cSuper\ CDef' = D'$ 
      and  $mName\ mDef = m'$ 
      and  $mReturn\ mDef = C0'$ 
      and  $varDefs-types\ (mParams\ mDef) = Cs'$ 
      and  $\forall Ds\ D0. (mtype(CT, m', D') = Ds \rightarrow D0) \longrightarrow Cs' = Ds \wedge C0' = D0$ 
    by (auto elim: method-typing.cases)
  with  $s\text{-super}\ mDef.name$  have
     $CDef' = CDef'$ 
    and  $D = D'$ 
    and  $m = m'$ 
    and  $\forall Ds\ D0. (mtype(CT, m, D) = Ds \rightarrow D0) \longrightarrow Cs' = Ds \wedge C0' = D0$ 
  using prems by auto
  thus  $?thesis$  using prems by (auto intro: mtype.intros)
next
case False
  hence  $?lookup\ m = None$  by (simp add: lookup-split)
  show  $?thesis$  using prems by (auto simp add: mtype.intros)
qed
qed

```

```

lemma sub-fields:
  assumes  $CT \vdash C <: D$ 
  shows  $\bigwedge Dg. fields(CT, D) = Dg \implies \exists Cf. fields(CT, C) = (Dg @ Cf)$ 
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  $s\text{-super}$  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  $\Gamma = \Gamma 1 ++ \Gamma 2$ 

```

and $\Gamma 2 = [xs \mapsto Bs]$
and $length\ xs = length\ ds$
and $length\ Bs = length\ ds$
and $\exists As. CT; \Gamma 1 \vdash ds : As \wedge CT \vdash As <: Bs$
shows $CT; \Gamma \vdash es : Ds \implies \exists Cs. (CT; \Gamma 1 \vdash ([ds/xs]es) : Cs \wedge CT \vdash Cs <: Ds)$ (**is** $?TYPINGS \implies ?P1$)
and $CT; \Gamma \vdash e : D \implies \exists C. (CT; \Gamma 1 \vdash ((ds/xs)e) : C \wedge CT \vdash C <: D)$ (**is** $?TYPING \implies ?P2$)
proof –
let $?COMMON-ASMS = (CT\ OK) \wedge (\Gamma = \Gamma 1 ++ \Gamma 2) \wedge (\Gamma 2 = [xs \mapsto Bs])$
 $\wedge (length\ Bs = length\ ds) \wedge (\exists As. CT; \Gamma 1 \vdash ds : As \wedge CT \vdash As <: Bs)$
have $RESULT: (?TYPINGS \longrightarrow ?COMMON-ASMS \longrightarrow ?P1)$
 $\wedge (?TYPING \longrightarrow ?COMMON-ASMS \longrightarrow ?P2)$
proof(*induct rule:typings-typing.induct*)
case (*ts-nil* $CT\ \Gamma$)
show $?case$
proof (*rule impI*)
have $(CT; \Gamma 1 \vdash ([ds/xs][]) : []) \wedge (CT \vdash [] <: [])$
by (*auto simp add: typings-typing.intros subtypings.intros*)
from this show $\exists Cs. (CT; \Gamma 1 \vdash ([ds/xs][]) : Cs) \wedge (CT \vdash Cs <: [])$ **by** *auto*
qed
next
case(*ts-cons* $CT\ \Gamma\ e0\ C0\ es\ Cs'$)
show $?case$
proof (*rule impI*)
assume $asms: (CT\ OK) \wedge (\Gamma = \Gamma 1 ++ \Gamma 2) \wedge (\Gamma 2 = [xs \mapsto Bs]) \wedge (length\ Bs = length\ ds) \wedge (\exists As. CT; \Gamma 1 \vdash ds : As \wedge CT \vdash As <: Bs)$
with *ts-cons* **have** $e0\text{-typ}: CT; \Gamma \vdash e0 : C0$ **by** *fastsimp*
with *ts-cons asms* **have**
 $\exists C. (CT; \Gamma 1 \vdash (ds/xs)\ e0 : C) \wedge (CT \vdash C <: C0)$
and $\exists Cs. (CT; \Gamma 1 \vdash [ds/xs]es : Cs) \wedge (CT \vdash Cs <: Cs')$
by *auto*
then obtain $C\ Cs$ **where**
 $(CT; \Gamma 1 \vdash (ds/xs)\ e0 : C) \wedge (CT \vdash C <: C0)$
and $(CT; \Gamma 1 \vdash [ds/xs]es : Cs) \wedge (CT \vdash Cs <: Cs')$ **by** *auto*
hence $CT; \Gamma 1 \vdash [ds/xs](e0\#es) : (C\#Cs)$
and $CT \vdash (C\#Cs) <: (C0\#Cs')$
by (*auto simp add: typings-typing.intros subtypings.intros*)
then show $\exists Cs. CT; \Gamma 1 \vdash map\ (subst\ [xs \mapsto ds])\ (e0\ \# \ es) : Cs \wedge CT \vdash Cs <: (C0\ \# \ Cs')$ **by** *auto*
qed
next
case (*t-var* $\Gamma\ x\ C'\ CT$)
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
 $lengths: length\ ds = length\ Bs$

and $G\text{-def} : \Gamma = \Gamma 1 ++ \Gamma 2$
and $G2\text{-def} : \Gamma 2 = [xs \mapsto] Bs$ **by** *auto*
from *lengths* $G2\text{-def}$ **have** *same-doms*: $\text{dom}([xs \mapsto] ds) = \text{dom}(\Gamma 2)$ **by** *auto*
from *asms* **show** $\exists C. CT; \Gamma 1 \vdash \text{subst} [xs \mapsto] ds (Var x) : C \wedge CT \vdash C$
 $<: C'$
proof (*cases* $\Gamma 2$ x)
case *None*
with $G\text{-def } t\text{-var}$ **have** $G1\text{-}x : \Gamma 1 \ x = \text{Some } C'$ **by** (*simp add: map-add-Some-iff*)
from *None same-doms* **have** $x \notin \text{dom}([xs \mapsto] ds)$ **by** (*auto simp only: domIff*)

hence $[xs \mapsto] ds \ x = \text{None}$ **by** (*auto simp only: map-add-Some-iff*)
hence $(ds/xs)(Var x) = (Var x)$ **by** *auto*
with $G1\text{-}x$ **have**
 $CT; \Gamma 1 \vdash (ds/xs)(Var x) : C'$ **and** $CT \vdash C' <: C'$
by (*auto simp add: typings-typing.intros subtyping.intros*)
thus *?thesis* **by** *auto*
next
case (*Some* Bi)
with $G\text{-def } t\text{-var}$ **have** $c'\text{-eq-bi} : C' = Bi$ **by** (*auto simp add: map-add-SomeD*)
from *prems* **have** $\text{length } xs = \text{length } Bs$ **by** *simp*
with *Some* $G2\text{-def}$ **have** $\exists i. (Bs!i = Bi) \wedge (i < \text{length } Bs) \wedge (\forall l. ((\text{length } l = \text{length } Bs) \longrightarrow ([xs \mapsto] l) \ x = \text{Some } (l!i)))$
by (*auto simp add: map-upds-index*)
then obtain i **where**
 $bs\text{-}i\text{-proj} : (Bs!i = Bi)$
and $i\text{-len} : i < \text{length } Bs$
and $P : (\bigwedge (l :: \text{exp list}). (\text{length } l = \text{length } Bs) \longrightarrow ([xs \mapsto] l) \ x = \text{Some } (l!i))$
by *fastsimp*
from *lengths* P **have** $\text{subst-}x : ([xs \mapsto] ds) \ x = \text{Some } (ds!i)$ **by** *auto*
from *prems* **obtain** As **where** $as\text{-ex} : CT; \Gamma 1 \vdash ds : As \wedge CT \vdash As <: Bs$ **by** *fastsimp*
hence $\text{length } As = \text{length } Bs$ **by** (*auto simp add: subtypings-length*)
hence $\text{proj-}i : CT; \Gamma 1 \vdash ds!i : As!i \wedge CT \vdash As!i <: Bs!i$ **using** $i\text{-len}$ *lengths* $as\text{-ex}$ **by** (*auto simp add: typings-proj*)
hence $CT; \Gamma 1 \vdash (ds/xs)(Var x) : As!i \wedge CT \vdash As!i <: C'$ **using** $c'\text{-eq-bi}$ $bs\text{-}i\text{-proj}$ $\text{subst-}x$ **by** *auto*
thus *?thesis* **..**
qed
qed
next
case (*t-field* $CT \ \Gamma \ e0 \ C0 \ Cf \ fDef \ Ci$)
show *?case*
proof (*rule impI*)
assume $asms : (CT \ OK) \wedge (\Gamma = \Gamma 1 ++ \Gamma 2) \wedge (\Gamma 2 = [xs \mapsto] Bs) \wedge (\text{length } Bs = \text{length } ds) \wedge (\exists As. CT; \Gamma 1 \vdash ds : As \wedge CT \vdash As <: Bs)$
from *t-field* **have** $flds : \text{fields}(CT, C0) = Cf$ **by** *fastsimp*
from *prems* **obtain** C **where** $e0\text{-typ} : CT; \Gamma 1 \vdash (ds/xs)e0 : C$ **and** $sub : CT \vdash C <: C0$ **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*) ($\lambda fd. vdName\ fd = fi$) = *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* $\rightarrow C$
and *subs*: *CT* $\vdash Cs <: Ds$
and *lens*: *length es* = *length Ds*
by auto
from *prems* **obtain** *C'* **where** *e0-typ*: *CT*; $\Gamma 1 \vdash (ds/xs)e0 : C'$ **and** *sub'*: *CT* $\vdash C' <: C0$ **by auto**
from *prems* **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:substs-subst-list1-subst-list2.simps 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-types  $Df$  =  $Ds$ 
by auto
from prems obtain  $Cs'$  where  $es$ -typ:  $CT; \Gamma 1 \vdash \vdash [ds/xs]es : Cs'$  and
subs':  $CT \vdash \vdash Cs' <: Cs$  by auto
have subst-e:  $(ds/xs)(New\ C\ es) = New\ C\ ([ds/xs]es)$ 
by(auto simp add:substs-subst-list1-subst-list2.simps 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  $CT\ \Gamma\ e0\ D\ C$ )
show ?case
proof(rule impI)
assume  $asms$ :  $(CT\ OK) \wedge (\Gamma = \Gamma 1 \ ++\ \Gamma 2) \wedge (\Gamma 2 = [xs\ [\mapsto]\ Bs]) \wedge (length\ Bs = length\ ds) \wedge (\exists As. CT; \Gamma 1 \vdash \vdash ds : As \wedge CT \vdash \vdash As <: Bs)$ 
from prems 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  $CT\ \Gamma\ e0\ D\ C$ )
show ?case
proof(rule impI)
assume  $asms$ :  $(CT\ OK) \wedge (\Gamma = \Gamma 1 \ ++\ \Gamma 2) \wedge (\Gamma 2 = [xs\ [\mapsto]\ Bs]) \wedge (length\ Bs = length\ ds) \wedge (\exists As. CT; \Gamma 1 \vdash \vdash ds : As \wedge CT \vdash \vdash As <: Bs)$ 
from prems 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  $(CT \vdash C \neg<: C' \wedge CT \vdash C' \neg<: C)$ 
      with  $e0\text{-typ}$  have  $CT; \Gamma 1 \vdash (ds/xs) (Cast C e0) : C$  by  $(auto simp add:
typings-typing.intros)$ 
    }
    ultimately have  $CT; \Gamma 1 \vdash (ds/xs) (Cast C e0) : C$  by  $auto$ 
    moreover have  $CT \vdash C <: C$  by  $(rule s\text{-refl})$ 
    ultimately show  $\exists C'. CT; \Gamma 1 \vdash (ds/xs)(Cast C e0) : C' \wedge CT \vdash C' <:$ 
  C by  $auto$ 
  qed
  next
  case  $(t\text{-scast } CT \ \Gamma \ e0 \ D \ C)$ 
  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  $prems$  obtain  $C'$  where  $e0\text{-typ}: CT; \Gamma 1 \vdash (ds/xs)e0 : C'$ 
      and  $sub1: CT \vdash C' <: D$ 
      and  $nsub1: CT \vdash C \neg<: D$ 
      and  $nsub2: CT \vdash D \neg<: C$  by  $auto$ 
    from  $not\text{-subtypes}[OF \ sub1 \ nsub1 \ nsub2]$  have  $CT \vdash C' \neg<: C$  by  $fastsimp$ 
    moreover have  $CT \vdash C \neg<: C'$  proof  $(rule ccontr)$ 
      assume  $\neg CT \vdash C \neg<: C'$ 
      hence  $CT \vdash C <: C'$  by  $auto$ 
      hence  $CT \vdash C <: D$  using  $sub1$  by  $(rule s\text{-trans})$ 
      with  $nsub1$  show  $False$  by  $auto$ 
    qed
    ultimately have  $CT; \Gamma 1 \vdash (ds/xs) (Cast C e0) : C$  using  $e0\text{-typ}$  by  $(auto
simp add: typings-typing.intros)$ 
    thus  $\exists C'. CT; \Gamma 1 \vdash (ds/xs)(Cast C e0) : C' \wedge CT \vdash C' <: C$  by  $(auto
simp add: s\text{-refl})$ 
  qed
  qed
  thus  $?TYPINGS \implies ?P1$  and  $?TYPING \implies ?P2$  using  $prems$  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 $(CT; \Gamma 2 \vdash + es : Cs) \implies (CT; \Gamma 1 ++ \Gamma 2 \vdash + es : Cs)$ (**is** $?P1 \implies ?P2$)

and $CT; \Gamma 2 \vdash e : C \implies CT; \Gamma 1 ++ \Gamma 2 \vdash e : C$ (**is** $?Q1 \implies ?Q2$)

proof –

have $(?P1 \implies ?P2) \wedge (?Q1 \implies ?Q2)$

by $(induct \ rule: typings\text{-typing.induct}, auto \ simp \ add: map\text{-add-find-right typings-typing.intros)$

thus $?P1 \implies ?P2$ and $?Q1 \implies ?Q2$ by $auto$

qed

3.5 Method Body Typing Lemma

lemma *A-1-4*:

assumes *ct-ok*: $CT \text{ OK}$
and $mb:mbody(CT, m, C) = xs . e$
and $mt:mtype(CT, m, C) = Ds \rightarrow D$
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
 $m\text{-param:varDefs-types } (mParams \ mDef) = Ds$
and $m\text{-ret:mReturn } mDef = D$
and $CT \vdash CDef \text{ OK}$
and $cName \ CDef = C$
by (*auto elim:mtype.cases ct-typing.cases*)
hence $CT \vdash+ (cMethods \ CDef) \text{ OK IN } C$ **by** (*auto elim:class-typing.cases*)
hence $CT \vdash mDef \text{ OK IN } C$ **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*: $CT \text{ OK}$
and *IH*: $\llbracket CT \text{ OK}; mtype(CT, m, Da) = Ds \rightarrow D \rrbracket$
 $\implies \exists D0 \ C0. (CT \vdash Da <: D0) \wedge (CT \vdash C0 <: D)$
 $\wedge (CT; [xs \mapsto] Ds, this \mapsto D0) \vdash e : C0$ **by** *fastsimp+*
from *mb-super* **have** *c-sub-da*: $CT \vdash C <: Da$ **by** (*auto simp add:s-super*)
from *mb-super* **have** $mt:mtype(CT, m, Da) = Ds \rightarrow D$ **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 $CT \vdash e \rightarrow e'$
and $CT \text{ OK}$
shows $\bigwedge C. \llbracket CT; \Gamma \vdash e : C \rrbracket$
 $\implies \exists C'. (CT; \Gamma \vdash e' : C' \wedge CT \vdash C' <: C)$

```

using assms
proof(induct rule: reduction.induct)
  case (r-field CT Ca Cf es fi e')
  hence  $CT; \Gamma \vdash \text{FieldProj } (\text{New } Ca \text{ es}) \text{ fi} : C$ 
    and ct-ok: CT OK
    and flds: fields(CT, Ca) = Cf
    and lkup2: lookup2 Cf es ( $\lambda fd. \text{vdName } fd = fi$ ) = Some e' by fastsimp+
  then obtain  $Ca' Cf' fDef$ 
    where new-typ: CT;  $\Gamma \vdash \text{New } Ca \text{ es} : Ca'$ 
    and flds': fields(CT, Ca') = Cf'
    and lkup: lookup Cf' ( $\lambda fd. \text{vdName } fd = fi$ ) = 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;  $\Gamma \vdash + \text{es} : Cs$ 
    and Ds-def: varDefs-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  $e' = \text{es}!i$ 
    and lookup Cf ( $\lambda fd. \text{vdName } fd = fi$ ) = 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
     $CT; \Gamma \vdash (\text{es}!i) : (Cs!i)$  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 fastsimp
  from r-invk obtain Ca' Ds Cs
    where  $CT; \Gamma \vdash \text{New } Ca \text{ es} : Ca'$ 
    and  $mtype(CT, m, Ca') = Cs \rightarrow C$ 
    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;  $\Gamma \vdash \text{New } Ca \text{ es} : Ca$ 
    and  $mt: mtype(CT, m, Ca) = Cs \rightarrow C$  by (auto elim: typing.cases)
  from ds-typs new-typ have CT;  $\Gamma \vdash + (ds @[\text{New } Ca \text{ es}]) : (Ds @[Ca])$  by (simp add: typings-append)
  moreover from  $A-1-4[OF - mb mt]$  r-invk obtain Da E
    where  $CT \vdash Ca <: Da$ 

```

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

```

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-invok-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-invok)
thus ?case by (auto simp add: subtyping.s-refl)
next
case (rc-invok-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-invok-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-invok)
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 $\bigwedge C. \llbracket CT; \Gamma \vdash e : C \rrbracket \implies \exists C'. (CT; \Gamma \vdash e' : C' \wedge CT \vdash C' <: C)$

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*: $\bigwedge D. \llbracket CT; \Gamma \vdash e' : D; CT OK \rrbracket \implies \exists E. CT; \Gamma \vdash e'' : E \wedge CT \vdash E$

```

<: 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;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;empty ⊢ (FieldProj (New C0 es) fi) : Ci
  using assms by (force simp add:subexpr-typing)
  then obtain Cf fDef C0'
  where CT;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;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;empty ⊢ MethodInvk (New C0 es) m ds : D
  using assms by (force simp add:subexpr-typing)
  then obtain C0' Cs
  where CT;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  $\Gamma = \text{empty}$ 
  shows
     $((\exists C0\ es\ fi. (\text{FieldProj } (\text{New } C0\ es)\ fi) \in \text{subexprs}(e))$ 
     $\vee (\exists C0\ es\ m\ ds. (\text{MethodInvk } (\text{New } C0\ es)\ m\ ds) \in \text{subexprs}(e))$ 
     $\vee (\exists C0\ D\ es. (\text{Cast } D\ (\text{New } C0\ es)) \in \text{subexprs}(e))$ 
     $\vee \text{val}(e))$  (is  $?F\ e \vee ?M\ e \vee ?C\ e \vee ?V\ e$  is  $?IH\ e)$ 
  using assms
  proof(induct CT  $\Gamma$  e C rule:typing-induct)
    case 1 thus  $?case$  using assms by auto
  next
    case  $(2\ C\ CT\ \Gamma\ x)$  thus  $?case$  by auto
  next
    case  $(3\ C0\ Ct\ Cf\ Ci\ \Gamma\ e0\ fDef\ fi)$ 
    have  $s1: e0 \in \text{subexprs}(\text{FieldProj } e0\ fi)$  by(auto simp add:isubexprs.intros)
    from 3 have  $?IH\ e0$  by auto
    moreover
    { assume  $?F\ e0$ 
      then obtain  $C0\ es\ fi'$  where  $s2: \text{FieldProj } (\text{New } C0\ es)\ fi' \in \text{subexprs}(e0)$  by
auto
      from rtrancl-trans[OF s2 s1] have  $?case$  by auto
    } moreover {
      assume  $?M\ e0$ 
      then obtain  $C0\ es\ m\ ds$  where  $s2: \text{MethodInvk } (\text{New } C0\ es)\ m\ ds \in \text{subexprs}(e0)$  by auto
      from rtrancl-trans[OF s2 s1] have  $?case$  by auto
    } moreover {
      assume  $?C\ e0$ 
      then obtain  $C0\ D\ es$  where  $s2: \text{Cast } D\ (\text{New } C0\ es) \in \text{subexprs}(e0)$  by auto
      from rtrancl-trans[OF s2 s1] have  $?case$  by auto
    } moreover {
      assume  $?V\ e0$ 
      then obtain  $C0\ es$  where  $e0 = (\text{New } C0\ es)$  and  $\text{vals}(es)$  by (force elim:val.cases)
      hence  $?case$  by(force intro:isubexprs.intros)
    }
    ultimately show  $?case$  by blast
  next
    case  $(4\ C\ C0\ CT\ Cs\ Ds\ \Gamma\ e0\ es\ m)$ 
    have  $s1: e0 \in \text{subexprs}(\text{MethodInvk } e0\ m\ es)$  by(auto simp add:isubexprs.intros)
    from 4 have  $?IH\ e0$  by auto
    moreover
    { assume  $?F\ e0$ 
      then obtain  $C0\ es\ fi$  where  $s2: \text{FieldProj } (\text{New } C0\ es)\ fi \in \text{subexprs}(e0)$  by
auto
      from rtrancl-trans[OF s2 s1] have  $?case$  by auto
    } moreover {
      assume  $?M\ e0$ 
      then obtain  $C0\ es'\ m'\ ds$  where  $s2: \text{MethodInvk } (\text{New } C0\ es')\ m'\ ds \in \text{subexprs}(e0)$  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 = 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 =
empty] ⇒ ?IH ((h#t)!i)
  and G-def: Γ = 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 = 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 ∈ subex-
prs(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 ∈ subex-
prs(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 $s1: e0 \in \text{subexprs}(\text{Cast } C \ e0)$ **by** $(\text{auto simp add: isubexprs.intros})$
from 8 **have** $?IH \ e0$ **by** auto
moreover
{ **assume** $?F \ e0$
then obtain $C0 \ es \ fi$ **where** $s2: \text{FieldProj} \ (\text{New } C0 \ es) \ fi \in \text{subexprs}(e0)$ **by**
 auto
from $\text{rtrancl-trans}[OF \ s2 \ s1]$ **have** $?case$ **by** auto
} **moreover** {
assume $?M \ e0$
then obtain $C0 \ es \ m \ ds$ **where** $s2: \text{MethodInvk} \ (\text{New } C0 \ es) \ m \ ds \in \text{subexprs}(e0)$ **by**
 auto
from $\text{rtrancl-trans}[OF \ s2 \ s1]$ **have** $?case$ **by** auto
} **moreover** {
assume $?C \ e0$
then obtain $C0 \ D' \ es$ **where** $s2: \text{Cast } D' \ (\text{New } C0 \ es) \in \text{subexprs}(e0)$ **by**
 auto
from $\text{rtrancl-trans}[OF \ s2 \ s1]$ **have** $?case$ **by** auto
} **moreover** {
assume $?V \ e0$
then obtain $C0 \ es'$ **where** $e0 = (\text{New } C0 \ es')$ **and** $\text{vals}(es')$ **by** $(\text{force elim: val.cases})$
hence $?case$ **by** $(\text{force intro: isubexprs.intros})$
}
ultimately show $?case$ **by** blast
qed

3.9 Type Soundness Theorem

theorem *Thm-2-4-3:*

assumes $e\text{-typ}: CT; \text{empty} \vdash e : C$

and $ct\text{-ok}: CT \text{ OK}$

and $\text{multisteps}: CT \vdash e \rightarrow^* e1$

and $\text{no-step}: \neg(\exists e2. CT \vdash e1 \rightarrow e2)$

shows $(\text{val}(e1) \wedge (\exists D. CT; \text{empty} \vdash e1 : D \wedge CT \vdash D <: C))$

$\vee (\exists D \ C \ es. (\text{Cast } D \ (\text{New } C \ es) \in \text{subexprs}(e1) \wedge CT \vdash C \neg <: D))$

proof –

from assms $\text{Cor-2-4-1-multi}[OF \ \text{multisteps} \ ct\text{-ok} \ e\text{-typ}]$ **obtain** $C1$

where $e1\text{-typ}: CT; \text{empty} \vdash e1 : C1$

and $C1\text{-sub-}C: CT \vdash C1 <: C$ **by** auto

from $e1\text{-typ}$ **have** $(\exists C0 \ es \ fi. (\text{FieldProj} \ (\text{New } C0 \ es) \ fi) \in \text{subexprs}(e1))$

$\vee (\exists C0 \ es \ m \ ds. (\text{MethodInvk} \ (\text{New } C0 \ es) \ m \ ds) \in \text{subexprs}(e1))$

$\vee (\exists C0 \ D \ es. (\text{Cast } D \ (\text{New } C0 \ es)) \in \text{subexprs}(e1))$

$\vee \text{val}(e1))$ **(is** $?F \ e1 \ \vee \ ?M \ e1 \ \vee \ ?C \ e1 \ \vee \ ?V \ e1)$ **by** $(\text{simp add:}$

$\text{closed-subterm-split})$

moreover

{ **assume** $?F \ e1$

then obtain $C0 \ es \ fi$ **where** $fp: \text{FieldProj} \ (\text{New } C0 \ es) \ fi \in \text{subexprs}(e1)$ **by**
 auto

then obtain Ci **where** $CT; \text{empty} \vdash \text{FieldProj} \ (\text{New } C0 \ es) \ fi : Ci$ **using** $e1\text{-typ}$

by(*force simp add:subexpr-typing*)
then obtain $C0'$ **where** *new-typ: CT;empty* \vdash *New C0 es* : $C0'$ **by** (*force elim: typing.cases*)
hence $C0 = C0'$ **by** (*auto elim:typing.cases*)
with *new-typ* **obtain** Df **where** $f1: \text{fields}(CT, C0) = Df$ **and** $\text{length } es = \text{length } Df$ **by**(*auto elim:typing.cases*)
from *Thm-2-4-2-1[OF e1-typ fp]* **obtain** $Cf fDef$
where $f2: \text{fields}(CT, C0) = Cf$
and $lkup: \text{lookup } Cf (\lambda fd. \text{vdName } fd = fi) = \text{Some}(fDef)$ **by** *force*
moreover from *fields-functional[OF f1 ct-ok f2]* $\text{length } es = \text{length } Cf$ **by** *auto*
moreover from *lookup-index[OF lkup]* **obtain** i **where**
 $i < \text{length } Cf$
and $fDef = Cf ! i$
and $(\text{length } Cf = \text{length } es) \longrightarrow \text{lookup2 } Cf \text{ es } (\lambda fd. \text{vdName } fd = fi) = \text{Some}(es ! i)$ **by** *auto*
ultimately have $\text{lookup2 } Cf \text{ es } (\lambda fd. \text{vdName } fd = fi) = \text{Some}(es ! i)$ **by** *auto*
with $f2$ **have** $CT \vdash \text{FieldProj}(\text{New } C0 \text{ es}) \text{ fi} \rightarrow (es ! i)$ **by**(*auto intro:reduction.intros*)
with fp **have** $\exists e2. CT \vdash e1 \rightarrow e2$ **by**(*simp add:subexpr-reduct*)
with *no-step* **have** *?thesis* **by** *auto*
} **moreover** {
assume *?M e1*
then obtain $C0 \text{ es } m \text{ ds}$ **where** $mi: \text{MethodInvk}(\text{New } C0 \text{ es}) \text{ m } ds \in \text{subexprs}(e1)$
by *auto*
then obtain D **where** $CT; \text{empty} \vdash \text{MethodInvk}(\text{New } C0 \text{ es}) \text{ m } ds : D$ **using** *e1-typ* **by**(*force simp add:subexpr-typing*)
then obtain $C0' \text{ Es } E$
where $m\text{-typ: } CT; \text{empty} \vdash \text{New } C0 \text{ es} : C0'$
and $m\text{type}(CT, m, C0') = \text{Es} \rightarrow E$
and $\text{length } ds = \text{length } \text{Es}$
by (*auto elim:typing.cases*)
from *Thm-2-4-2-2[OF e1-typ mi]* **obtain** $xs \text{ e0}$ **where** $mb: \text{mbody}(CT, m, C0) = xs . \text{e0}$ **and** $\text{length } xs = \text{length } ds$ **by** *auto*
hence $CT \vdash (\text{MethodInvk}(\text{New } C0 \text{ es}) \text{ m } ds) \rightarrow (\text{subst}[xs[\mapsto]ds, \text{this} \mapsto (\text{New } C0 \text{ es})] \text{e0})$ **by**(*auto simp add:reduction.intros*)
with mi **have** $\exists e2. CT \vdash e1 \rightarrow e2$ **by**(*simp add:subexpr-reduct*)
with *no-step* **have** *?thesis* **by** *auto*
} **moreover** {
assume *?C e1*
then obtain $C0 \text{ D } es$ **where** $c\text{-def: } \text{Cast } D (\text{New } C0 \text{ es}) \in \text{subexprs}(e1)$ **by** *auto*
then obtain D' **where** $CT; \text{empty} \vdash \text{Cast } D (\text{New } C0 \text{ es}) : D'$ **using** *e1-typ* **by** (*force simp add:subexpr-typing*)
then obtain $C0'$ **where** $\text{new-typ: } CT; \text{empty} \vdash \text{New } C0 \text{ es} : C0'$ **and** $D\text{-eq-}D'$: $D = D'$ **by** (*auto elim:typing.cases*)
hence $C0\text{-eq-}C0'$: $C0 = C0'$ **by**(*auto elim:typing.cases*)
hence *?thesis* **proof**(*cases* $CT \vdash C0 <: D$)
case *True*
hence $CT \vdash \text{Cast } D (\text{New } C0 \text{ es}) \rightarrow (\text{New } C0 \text{ es})$ **by**(*auto simp add:reduction.intros*)

```

  with c-def have  $\exists e2. CT \vdash e1 \rightarrow e2$  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 prems 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 *op @* 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-invok-arg':  $CT \vdash ei \rightarrow ei' \implies \text{append } el (ei \# er) e' \implies \text{append } el (ei' \# er) e'' \implies$ 
 $CT \vdash \text{MethodInvk } e m e' \rightarrow \text{MethodInvk } e m e''$ 
unfolding append-def by simp (rule reduction.intros(6))

```

```

lemma rc-new-arg':  $CT \vdash ei \rightarrow ei' \implies \text{append } el (ei \# er) e \implies \text{append } el (ei' \# er) e'$ 
 $\implies CT \vdash \text{New } C e \rightarrow \text{New } C e'$ 

```

```

unfolding append-def by simp (rule reduction.intros(7))

lemmas [code-pred-intro] = reduction.intros(1-5)[unfolded Predicate.eq-is-eq[symmetric]]
  rc-invok-arg' rc-new-arg' reduction.intros(8)

code-pred (modes: i => i => o => bool as reduce) reduction
proof –
  case append
  from this show thesis
    unfolding append-def by (cases a1) fastsimp+
next
  case reduction
  from this show thesis
  proof (cases rule: reduction.cases)
    case r-field
    with reduction(2) show thesis
    unfolding Predicate.eq-is-eq by fastsimp
  next
    case r-invok
    with reduction(3) show thesis
    unfolding Predicate.eq-is-eq by fastsimp
  next
    case r-cast
    with reduction(4) show thesis
    by fastsimp
  next
    case rc-field
    with reduction(5) show thesis
    by fastsimp
  next
    case rc-invok-recv
    with reduction(6) show thesis
    by fastsimp
  next
    case rc-invok-arg
    with reduction(7) show thesis
    unfolding append-def by fastsimp
  next
    case rc-new-arg
    with reduction(8) show thesis
    unfolding append-def by fastsimp
  next
    case rc-cast
    with reduction(9) show thesis
    by fastsimp
  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 $CT\ m\ D\ Cs\ C = (\neg (\forall Ds\ D0. mtype(CT,m,D) = Ds \rightarrow D0 \rightarrow Cs = Ds \wedge C = D0))$

lemma *method-typing-aux*:

$(\forall Ds\ D0. mtype(CT,m,D) = Ds \rightarrow D0 \rightarrow Cs = Ds \wedge C = D0) = (\neg \text{method-typing-aux } CT\ m\ D\ Cs\ C)$

unfolding *method-typing-aux-def* **by** *auto*

lemma [*code-pred-intro*]:

$mtype(CT,m,D) = Ds \rightarrow D0 \implies Cs \neq Ds \implies \text{method-typing-aux } CT\ m\ D\ Cs\ C$

unfolding *method-typing-aux-def* **by** *auto*

lemma [*code-pred-intro*]:

$mtype(CT,m,D) = Ds \rightarrow D0 \implies C \neq D0 \implies \text{method-typing-aux } CT\ m\ D\ Cs\ C$

unfolding *method-typing-aux-def* **by** *auto*

declare *method-typing.intros*[*unfolded method-typing-aux*,
unfolded Predicate.eq-is-eq[symmetric], *code-pred-intro*]

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

code-pred (*modes: i => i => bool*) *class-typing*

proof –

case *class-typing*

from *class-typing.cases*[*OF this(1)*, *of thesis*] *this(2)* **show** *thesis*

unfolding *Predicate.eq-is-eq append-def* **by** *fastsimp*

next

case *method-typing*

from *method-typing.cases*[*OF this(1)*, *of thesis*] *this(2)* **show** *thesis*

unfolding *Predicate.eq-is-eq append-def method-typing-aux-def* **by** *fastsimp*

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 $A :: \text{className}$
where $A == \text{Suc } 0$

abbreviation $B :: \text{className}$
where $B == 2$

abbreviation $cPair :: \text{className}$
where $cPair == 3$

definition $\text{classA-Def} :: \text{classDef}$
where
 $\text{classA-Def} = (\mid \text{cName} = A, \text{cSuper} = \text{Object}, \text{cFields} = [], \text{cConstructor} =$
 $(\mid \text{kName} = A, \text{kParams} = [], \text{kSuper} = [], \text{kInits} = []), \text{cMethods} = [])$

definition
 $\text{classB-Def} = (\mid \text{cName} = B, \text{cSuper} = \text{Object}, \text{cFields} = [], \text{cConstructor} =$
 $(\mid \text{kName} = B, \text{kParams} = [], \text{kSuper} = [], \text{kInits} = []), \text{cMethods} = [])$

abbreviation $\text{ffst} :: \text{varName}$
where
 $\text{ffst} == 4$

abbreviation $\text{fsnd} :: \text{varName}$
where
 $\text{fsnd} == 5$

abbreviation $\text{setfst} :: \text{methodName}$
where
 $\text{setfst} == 6$

abbreviation $\text{newfst} :: \text{varName}$
where
 $\text{newfst} == 7$

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

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

value [*code*] *exampleProg* ⊢ *classA-Def OK*
value [*code*] *exampleProg* ⊢ *classB-Def OK*
value [*code*] *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* []]) *fst* *fsnd* →* *x*}

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/>.