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

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 ................................ 31
  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

ttheory FJDefs
imports Main
begin

1.1 Syntax

We use a named representation for terms: variables, method names, and
class names, are all represented as nat. 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 → 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.

\[
\text{inductive-set}
\begin{align*}
\text{isubexprs} :: & \ (\text{exp } \star \text{ exp}) \ \text{set} \\
\text{and} \ \text{isubexprs'} :: & \ [\text{exp}, \text{exp}] \Rightarrow \text{bool} \ (\cdot \in \text{isubexprs}'(\cdot) [80,80] 80)
\end{align*}
\]

where
\[
\begin{align*}
e' \in \text{isubexprs}(e) & \equiv (e',e) \in \text{isubexprs} \\
\text{se-field} : e \in \text{isubexprs}(\text{FieldProj } e \ f) \\
\text{se-invoke} : e \in \text{isubexprs}(\text{MethodInvk } e \ m \ es) \\
\text{se-invokearg} : [\ [ei \in \text{set } es \ ] \implies ei \in \text{isubexprs}(\text{MethodInvk } e \ m \ es) \\
\text{se-new} : [\ [ei \in \text{set } es \ ] \implies ei \in \text{isubexprs}(\text{New } C \ es) \\
\text{se-cast} : e \in \text{isubexprs}(\text{Cast } C \ e)
\end{align*}
\]

abbreviation
\[
\text{subexprs} :: [\text{exp}, \text{exp}] \Rightarrow \text{bool} \ (\cdot \in \text{subexprs}'(\cdot) [80,80] 80) \ \text{where}
\begin{align*}
e' \in \text{subexprs}(e) & \equiv (e',e) \in \text{subexprs}^\ast
\end{align*}
\]

1.3 Values

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

\[
\text{inductive}
\begin{align*}
\text{vals} :: & \ [(\text{exp } \text{list})] \Rightarrow \text{bool} \ (\text{vals}'(\cdot) [80] 80) \\
\text{and} \ \text{val} :: & \ [\text{exp}] \Rightarrow \text{bool} \ (\text{val}'(\cdot) [80] 80)
\end{align*}
\]

where
\[
\begin{align*}
\text{vals-nil} : & \ \text{vals}([]) \\
| \ \text{vals-cons} : & \ [\ \text{val}(vh); \ \text{vals}(vt) \ ] \implies \text{vals}((vh \ # \ vt)) \\
| \ \text{val} : & \ [\ \text{vals}(vs) \ ] \implies \text{val}(\text{New } C \ vs)
\end{align*}
\]

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.

\[
\text{primrec}
\begin{align*}
\text{substs} :: & \ (\text{varName } \rightarrow \text{exp}) \Rightarrow \text{exp} \Rightarrow \text{exp} \\
\text{and} \ \text{subst-list1} :: & \ (\text{varName } \rightarrow \text{exp}) \Rightarrow \text{exp } \text{list} \Rightarrow \text{exp } \text{list} \\
\text{and} \ \text{subst-list2} :: & \ (\text{varName } \rightarrow \text{exp}) \Rightarrow \text{exp } \text{list} \Rightarrow \text{exp } \text{list} \ \text{where}
\end{align*}
\]

\[
\begin{align*}
\text{substs } \sigma (\text{Var } x) = & \ (\text{case } (\sigma(x)) \text{ of } \text{None } \Rightarrow (\text{Var } x) \ | \ \text{Some } p \Rightarrow p) \\
\text{substs } \sigma (\text{FieldProj } e \ f) = & \ \text{FieldProj } (\text{substs } \sigma e) \ f \\
\text{substs } \sigma (\text{MethodInvk } e \ m \ es) = & \ \text{MethodInvk } (\text{substs } \sigma e) \ m \ (\text{subst-list1 } \sigma es) \\
\text{substs } \sigma (\text{New } C \ es) = & \ \text{New } C \ (\text{subst-list2 } \sigma es)
\end{align*}
\]


\[ \text{substs} \sigma (\text{Cast} \ C \ e) = \text{Cast} \ C (\text{substs} \ \sigma \ e) \]
\[ \text{subst-list1} \ \sigma \ [\ ] = [\ ] \]
\[ \text{subst-list1} \ \sigma \ (h \ # \ t) = (\text{substs} \ \sigma \ h) \ # \ (\text{subst-list1} \ \sigma \ t) \]
\[ \text{subst-list2} \ \sigma \ [\ ] = [\ ] \]
\[ \text{subst-list2} \ \sigma \ (h \ # \ t) = (\text{substs} \ \sigma \ h) \ # \ (\text{subst-list2} \ \sigma \ t) \]

**abbreviation**
\[ \text{substs-syn} :: \exp \ list \Rightarrow \varName \ list \Rightarrow \exp \Rightarrow \exp \]
\[ (\lfloor \cdot / \cdot \rfloor \ (80,80,80) 80) \where \]
\[ (ds/xs)e \equiv \text{substs} \ (\text{map-upds} \ \text{Map}.\emptyset \ x s \ ds) \ e \]

**abbreviation**
\[ \text{subst-list-syn} :: \exp \ list \Rightarrow \varName \ list \Rightarrow \exp \ list \Rightarrow \exp \ list \]
\[ (\lfloor \cdot / \cdot \rfloor \ (80,80,80) 80) \where \]
\[ [ds/xs]es \equiv \text{map} \ (\text{substs} \ (\text{map-upds} \ \text{Map}.\emptyset \ x s \ ds)) \ es \]

### 1.5 Lookup

The function \( \text{lookup} \ f \ l \) function returns an option containing the first element of \( l \) satisfying \( f \), or \text{None} if no such element exists.

**primrec** \( \text{lookup} :: \ 'a \ list \Rightarrow (\ 'a \Rightarrow \text{bool} ) \Rightarrow \ 'a \ option \)
\[ \where \]
\[ \text{lookup} [\ ] \ P = \text{None} \]
\[ \text{lookup} (h \# t \ ) \ P = (\text{if} \ P \ h \ \text{then} \ \text{Some} \ h \ \text{else} \ \text{lookup} \ t \ P ) \]

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

### 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**
\[ \text{varDefs-names} :: \varDef \ list \Rightarrow \varName \ list \ \where \]
\[ \text{varDefs-names} = \text{map} \ \text{vdName} \]

**definition**
\[ \text{varDefs-types} :: \varDef \ list \Rightarrow \className \ list \ \where \]
\[ \text{varDefs-types} = \text{map} \ \text{vdType} \]

### 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**

\[
\text{subtyping} :: [\text{classTable}, \text{className}, \text{className}] \Rightarrow \text{bool} \quad (- \vdash - <: - [80,80,80] 80)
\]

**where**

\[
s\text{-refl} : \quad CT \vdash C <: C \\
| s\text{-trans} : \quad \begin{array}{l}
CT \vdash C <: D; CT \vdash D <: E \\
\implies CT \vdash C <: E
\end{array} \\
| s\text{-super} : \quad \begin{array}{l}
CT(C) = \text{Some}(CDef); cSuper CDef = D \\
\implies CT \vdash C <: D
\end{array}
\]

**abbreviation**

\[
\text{neg-subtyping} :: [\text{classTable}, \text{className}, \text{className}] \Rightarrow \text{bool} \quad (- \vdash - \neg <: - [80,80,80] 80)
\]

**where**

\[
CT \vdash S \neg <: T \equiv \neg CT \vdash S <: T
\]

**inductive**

\[
\text{subtypings} :: [\text{classTable}, \text{className list}, \text{className list}] \Rightarrow \text{bool} \quad (- \vdash + -: - [80,80,80] 80)
\]

**where**

\[
\begin{array}{l}
ss\text{-nil} : \quad CT \vdash + [] <: [] \\
| ss\text{-cons} : \quad CT \vdash C_0 <: D_0; CT \vdash + C_s <: D_s \\
\implies CT \vdash + (C_0 \# C_s) <: (D_0 \# D_s)
\end{array}
\]

### 1.8 fields Relation

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

**inductive**

\[
\text{fields} :: [\text{classTable}, \text{className}, \text{varDef list}] \Rightarrow \text{bool} \quad (\text{fields}(\cdot, \cdot, \cdot) = - [80,80,80] 80)
\]

**where**

\[
\begin{array}{l}
f\text{-obj}: \\
f\text{fields}(CT, \text{Object}) = [] \\
f\text{-class}: \\
\begin{array}{l}
CT(C) = \text{Some}(CDef); cSuper CDef = D; cFields CDef = Cf; fields(CT,D) = Dg; DgCf = Dg \circ Cf \\
\end{array} \\
\implies \text{fields}(CT,C) = DgCf
\end{array}
\]

### 1.9 mtype Relation

The **mtype** relation, written $\text{mtype}(CT, m, C) = Cs \to C_0$ 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'(\text{\texttt{-,-,-}})) = \cdot \rightarrow [80,80,80,80] \rightarrow 80

\textbf{where}

\textit{mt-class:}

\begin{itemize}
\item $CT(C) = \text{Some}(CDef);$
\item lookup (cMethods CDef) $(\lambda md.(mName md = m)) = \text{Some}(mDef);$\n\item varDefs-types $(mParams mDef) = Bs;$
\item mReturn $mDef = B$
\end{itemize}

⇒ mtype(CT,m,C) = Bs → B

\textit{mt-super:}

\begin{itemize}
\item $CT(C) = \text{Some}(CDef);$
\item lookup (cMethods CDef) $(\lambda md.(mName md = m)) = \text{None};$
\item cSuper $CDef = D;$
\item mtype(CT,m,D) = Bs → B
\end{itemize}

⇒ mtype(CT,m,C) = Bs → B

\subsection*{1.10 mbody Relation}

The \textit{mtype} relation, written \textit{mbody}(CT,m,C) = xs.e₀ relates a class \textit{C},
methond name \textit{m}, and the names of the parameters \textit{xs} and the body of the
method \textit{e₀}. It either returns the parameter names and body of the declaration of \textit{m}
in \textit{C}, if any such declaration exists, and otherwise the parameter names and body of \textit{m}
from \textit{C}'s superclass.

\textbf{inductive}

\textit{mbody} :: [classTable, methodName, className, varName list, exp] ⇒ bool (mbody'(\text{\texttt{-,-,-}}))
= \cdot \cdot \cdot [80,80,80,80] \rightarrow 80

\textbf{where}

\textit{mb-class:}

\begin{itemize}
\item $CT(C) = \text{Some}(CDef);$
\item lookup (cMethods CDef) $(\lambda md.(mName md = m)) = \text{Some}(mDef);$\n\item varDefs-names $(mParams mDef) = xs;$
\item mBody $mDef = e$
\end{itemize}

⇒ mbody(CT,m,C) = xs . e

\textit{mb-super:}

\begin{itemize}
\item $CT(C) = \text{Some}(CDef);$
\item lookup (cMethods CDef) $(\lambda md.(mName md = m)) = \text{None};$
\item cSuper $CDef = D;$
\item mbody(CT,m,D) = xs . e
\end{itemize}

⇒ mbody(CT,m,C) = xs . e

\subsection*{1.11 Typing Relation}

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

\textbf{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;Γ ⊢ [] : []

 | ts-cons :
 | [ CT;Γ ⊢ e0 : C0; CT;Γ ⊢ es : Cs ]
 | ⇒ CT;Γ ⊢ (e0 # es) : (C0 # Cs)

 | t-var :
 | [ Γ(x) = Some C ] ⇒ CT;Γ ⊢ (Var x) : C

 | t-field :
 | [ CT;Γ ⊢ e0 : C0;
 | fields(CT,C0) = Cf;
 | lookup Cf (λfd.(vdName fd = fi)) = Some(fDef);
 | vdType fDef = Ci ]
 | ⇒ CT;Γ ⊢ FieldProj e0 fi : Ci

 | t-invk :
 | [ CT;Γ ⊢ e0 : C0;
 | mtype(CT,m,C0) = Ds → C;
 | CT;Γ ⊢ es : Cs;
 | CT ⊢ Cs < Ds;
 | length es = length Ds ]
 | ⇒ CT;Γ ⊢ MethodInvk e0 m es : C

 | t-new :
 | [ fields(CT,C) = Df;
 | length es = length Df;
 | varDefs-types Df = Ds;
 | CT;Γ ⊢ es : Cs;
 | CT ⊢ Cs < Ds ]
 | ⇒ CT;Γ ⊢ New C es : C

 | t-ucast :
 | [ CT;Γ ⊢ e0 : D;
 | CT ⊢ D <: C ]
 | ⇒ CT;Γ ⊢ Cast C e0 : C

 | t-dcast :
 | [ CT;Γ ⊢ e0 : D;
 | CT ⊢ C <: D; C ≠ D ]
 | ⇒ CT;Γ ⊢ Cast C e0 : C

 | t-scast :
 | [ CT;Γ ⊢ e0 : D;
\[ \text{proof} - \]
\[ \text{fix es Cs} \]
\[ \text{let } \sharp \text{IH} = \text{CT}; \text{Γ} \vdash \text{es : Cs} \rightarrow (\forall i < \text{length es}. \ P \text{ CT} \text{ Γ} (\text{es}!i) (\text{Cs}!i)) \]
\[ \text{have } \sharp \text{IH} \land (\sharp T \rightarrow ?P) \]
\[ \text{proof (induct rule: typings-typing.induct)} \]
\[ \text{case (t-snil CT Γ) show } \#\text{case by auto} \]
\[ \text{next} \]
\[ \text{case (t-cons CT Γ e0 C0 es Cs)} \]
\[ \text{show } \#\text{case proof} \]
\[ \text{fix i} \]
\[ \text{show } i < \text{length } (\text{e0}#\text{es}) \rightarrow P \text{ CT} \text{ Γ } ((\text{e0}#\text{es})!i) ((\text{C0}#\text{Cs})!i) \text{ using ts-cons by (cases i, auto)} \]
\[ \text{qed} \]
\[ \text{next} \]
\[ \text{case t-var then show } \#\text{case using assms by auto} \]
\[ \text{next} \]
\[ \text{case t-field then show } \#\text{case using assms by auto} \]
\[ \text{next} \]
\[ \text{case t-invk then show } \#\text{case using assms by auto} \]
\[ \text{next} \]
\[ \text{case t-new then show } \#\text{case using assms by auto} \]
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 \(\text{md}\), declared in a class \(C\), is well-typed, written \(\text{CT} \vdash \text{md} \text{OK IN } C\) 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 :: \([\text{classList}, \text{methodDef}, \text{className}] \Rightarrow \text{bool}\) (\(-\vdash - \text{OK IN -}[80,80,80]\) 80)
where
m-typing:
\[
\begin{align*}
\text{CT}(C) &= \text{Some}(C\text{Def}); \\
\text{Name } C\text{Def} &= C; \\
\text{Name } C\text{Def} &= D; \\
\text{Name } m\text{Def} &= m; \\
\text{lookup } (\text{cMethods } C\text{Def}) (\lambda m. \text{Name } m\text{Def} = m) &= \text{Some}(m\text{Def}); \\
\text{mReturn } m\text{Def} &= C0; \\
\text{mParams } m\text{Def} &= Cxs; \\
\text{mBody } m\text{Def} &= e0; \\
\text{varDefs-types } Cxs &= Cs; \\
\text{varDefs-names } Cxs &= xs; \\
\Gamma = (\text{map-upds Map.empty xs Cs})(\text{this }\mapsto C); \\
\text{CT};\Gamma \vdash e0 : E0; \\
\text{CT} \vdash E0 <: C0; \\
\forall Ds D0. (\text{mtype } (\text{CT}, m, D) = Ds \rightarrow D0) \rightarrow (Cs = Ds \land C0 = D0) \\
\Rightarrow \text{CT} \vdash m\text{Def} \text{OK IN } C
\end{align*}
\]

inductive method-typings :: \([\text{classList}, \text{methodDef list}, \text{className}] \Rightarrow \text{bool}\) (\(-\vdash - \text{OK IN -}[80,80,80]\) 80)
where
ms-nil :
\(\text{CT} \vdash + [] \text{OK IN } C\)
| ms-cons :
\[
\begin{align*}
\text{CT} \vdash m \text{OK IN } C; \\
\text{CT} \vdash + m\text{Def} \text{OK IN } C
\end{align*}
\]
\[\Rightarrow \text{CT} \vdash + (m \# m\text{Def}) \text{OK IN } C\]
1.13 Class Typing Relation

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

\[
\text{inductive class-typing} :: [\text{classTable}, \text{classDef}] \Rightarrow \mathbf{bool} \\
\text{where} \\
\text{t-class:} \ \ \ [ \begin{array}{l}
\text{cName CDef} = C; \cr
\text{cSuper CDef} = D; \cr
\text{cConstructor CDef} = KDef; \cr
\text{cMethods CDef} = M; \cr
\text{kName} KDef = C; \cr
\text{kParams} KDef = (Dg@Cf); \cr
\text{kSuper} KDef = \text{varDefs-names Dg}; \cr
\text{kInits} KDef = \text{varDefs-names Cf}; \cr
\text{fields}(CT,D) = Dg; \cr
CT \vdash M \text{ OK IN C } \end{array}] \Rightarrow CT \vdash \text{CDef} \text{ OK}
\]

1.14 Class Table Typing Relation

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

\[
\text{inductive ct-typing} :: \text{classTable} \Rightarrow \mathbf{bool} \\
\text{where} \\
\text{ct-all-ok:} \ \ \ [ \begin{array}{l}
\text{Object} \not\in \text{dom}(CT); \cr
\forall C \text{ CDef}. \ CT(C) = \text{Some}(CDef) \Rightarrow (CT \vdash \text{CDef} \text{ OK}) \land (\text{cName CDef} = C) \end{array}] \Rightarrow CT \text{ OK}
\]

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.

\[
\text{inductive reduction} :: [\text{classTable}, \text{exp}, \text{exp}] \Rightarrow \mathbf{bool} \\
\text{where} \\
\text{r-field:} \ \ \ [ \begin{array}{l}
\text{fields}(CT,C) = Cf; \cr
\text{lookup2} Cf es (\lambda fd.(vdName fd = fi)) = \text{Some} (ei) \end{array}] \Rightarrow CT \vdash \text{FieldProj} (\text{New C es} \ fi \rightarrow ei)
\]

| r-invK: 
| \begin{array}{l}
\text{mbody}(CT,m,C) = xs \cdot e0; 
\end{array} |
\[\text{subs } ((\text{map-upds } \text{Map}. \text{empty } \text{xs } \text{ds})(\text{this } \mapsto (\text{New } \text{C } \text{es}))) \text{ e0 } = \text{e0'}\]\[\implies \text{CT} \vdash \text{MethodInvk} (\text{New } \text{C } \text{es}) \text{ m } \text{ds} \to \text{e0'}\]

| r-cast:
| \[\begin{array}{l}
\text{CT} \vdash \text{C }\triangleleft: \text{D} \\
\implies \text{CT} \vdash \text{Cast } \text{D } (\text{New } \text{C } \text{es}) \to \text{New } \text{C } \text{es}
\end{array}\]

| rec-field:
| \[\begin{array}{l}
\text{CT} \vdash \text{e0 }\to \text{e0'} \\
\implies \text{CT} \vdash \text{FieldProj } \text{e0 } \text{f} \to \text{FieldProj } \text{e0'} \text{f}
\end{array}\]

| rec-invkrecv:
| \[\begin{array}{l}
\text{CT} \vdash \text{e0 }\to \text{e0'} \\
\implies \text{CT} \vdash \text{MethodInvk } \text{e0 } \text{m es }\to \text{MethodInvk } \text{e0'} \text{m es}
\end{array}\]

| rec-invkarg:
| \[\begin{array}{l}
\text{CT} \vdash \text{ei }\to \text{ei'} \\
\implies \text{CT} \vdash \text{MethodInvk } \text{e0 } \text{m } \text{(el@ei#er)} \to \text{MethodInvk } \text{e0 } \text{m } \text{(el@ei'#er)}
\end{array}\]

| rec-newarg:
| \[\begin{array}{l}
\text{CT} \vdash \text{ei }\to \text{ei'} \\
\implies \text{CT} \vdash \text{New } \text{C } \text{(el@ei#er)} \to \text{New } \text{C } \text{(el@ei'#er)}
\end{array}\]

| r-cast:
| \[\begin{array}{l}
\text{CT} \vdash \text{e0 }\to \text{e0'} \\
\implies \text{CT} \vdash \text{Cast } \text{C } \text{e0 }\to \text{Cast } \text{C } \text{e0'}
\end{array}\]

inductive
\[\text{reductions } :: [\text{classTable}, \text{exp}, \text{exp}] \Rightarrow \text{bool} (\cdot \vdash - \to\ast - [80,80,80] 80)\]
where
| rs-refl: \[\text{CT} \vdash \text{e }\to\ast \text{e}\]
| rs-trans: \[\begin{array}{l}
\text{CT} \vdash \text{e }\to \text{e'}; \text{CT} \vdash \text{e'} \to\ast \text{e''} \\
\implies \text{CT} \vdash \text{e }\to\ast \text{e''}
\end{array}\]

end

2 FJAux: Auxiliary Lemmas

theory FJAux imports FJDefs
begin

2.1 Non-FJ Lemmas

2.1.1 Lists

lemma mem-ith:
\[\begin{array}{l}
\text{assumes } \text{ei }\in \text{set } \text{es} \\
\text{shows } \exists \text{ el er. } \text{es }= \text{el@ei#er} \\
\text{using } \text{assms}
\end{array}\]

proof (induct \text{es})
case Nil thus \(\text{?case by auto}\)

next
case (Cons esh est)
{ assume esh = ei
  with Cons have \(\text{?case by blast}\)
}
moreover {  
  assume esh \(\neq\) ei
  with Cons have ei \(\in\) set est by auto
  with Cons obtain el er where esh \# est = (esh\#el) \(\oplus\) (ei\#er) by auto
  hence \(\text{?case by blast}\)  
}
ultimately show \(\text{?case by blast}\)
qed

lemma \(\text{ith-mem}: \bigwedge i. \left[ i < \text{length es} \right] \implies es!i \in \text{set es}\)
proof (induct es)
case Nil thus \(\text{?case by auto}\)
next
case (Cons h t) thus \(\text{?case by (cases i, auto)}\)
qed

2.1.2 Maps

lemma \(\text{map-shuffle:}\)
assumes \(\text{length xs} = \text{length ys}\)
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 \(\text{map-upds-index:}\)
assumes \(\text{length xs} = \text{length As}\)
and \([xs] \mapsto [As] x = \text{Some Ai}\)
shows \(\exists i. (\text{As!i} = Ai)\)
\(\land (i < \text{length As})\)
\(\land (\forall (Bs::'c list).((\text{length Bs} = \text{length As}) \implies ((xs[x]Bs) x = \text{Some (Bs !i)})))\)
(is \(\exists i. ?P_i x As\)
is \(\exists i. (?P_1 i As) \land (?P_2 i As) \land (\forall Bs::('c list).(?P_3 i xs As Bs)))\)
using assms
proof (induct xs As rule: list-induct2)
assume \([[]] \mapsto [[]] x = \text{Some Ai}\)
moreover have \(\neg [[]] \mapsto [[]] x = \text{Some Ai}\) by auto
ultimately show \(\exists i. ?P_i [] []\) by contradiction
next
fix xa xs y ys
assume \(\text{length-xs-ys}: \text{length xs} = \text{length ys}\)
and IH: \([xs] \mapsto [ys] x = \text{Some Ai} \implies \exists i. ?P_i x y\)
and map-eq-Some: \([xa \# xs] \mapsto [y \# ys] x = \text{Some Ai}\)
then have map-decomp: \([xa \# xs] \mapsto [y \# ys] = [xa \mapsto y] ++ [xs[x]ys]\) by fastforce
show \(\exists i. ?P_i (xa \# xs) (y \# ys)\)

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] Bs by fastforce
  { fix Bs :: 'c list
    assume length-Bs: length Bs = length ([![ys] :: ![ys])
    then obtain ![n] where length ([![ys] :: ![ys]) = Suc ![n] by auto
    with length-Bs obtain ![b] ![bs] where Bs-def: Bs = ![b] ![bs] ![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] ![xs] Bs by simp
  }
  with R1 R2 have ![P (i + 1)] (![xa] ![xs] ![ys]) ![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 ([![ys] :: ![ys])
    then obtain ![n] where length ([![ys] :: ![ys]) = Suc ![n] by auto
    with length-Bs obtain ![b] ![bs] where Bs-def: Bs = ![b] ![bs] ![bs] by (auto simp add:length-Suc-conv)
    with length-Bs have length ![ys] = length ![bs] by simp
    hence dom ([![ys] → ![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-unfold)
    with Bs-def have ![P3 0] (![xa] ![xs] ![ys]) Bs by simp
  }
  with ai-def have ![P 0] (![xa] ![xs] ![ys]) ![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 (subst ![σ] ![l])
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 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 (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 \( \exists i < (\text{length } l1). e = l1!i \land ((\text{length } l1 = \text{length } l2) =⇒ \text{lookup2} l1 l2 f = \text{Some} (l2!i)) \)
using assms
proof (induct l1)
case Nil thus ?case by auto
next
case (Cons h1 t1)
  \{ assume asm: f h1 
  hence 0 < length (h1 # t1) \land 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 tl f = Some e 
    using Cons by (auto simp add:lookup.simps) 
 then obtain i where 
  \(i<\text{length} \; \text{tl}\) 
  and \(e = tl ! i\) 
  and \(ih:\text{length} \; \text{tl} = \text{length} \; (tl \; \text{tl}) \rightarrow \text{lookup2} \; \text{tl} \; (tl \; \text{tl}) \; f = \text{Some} \; ((tl \; \text{tl}) \setminus i)\) 
    using Cons by blast 
  hence Suc i < \text{length} \; (h1 \# tl) \land e = (h1 \# tl)!(Suc i) 
    using Cons by auto 
 moreover { 
  assume length (h1 \# tl) = length l2 
  hence lens:length l2 = Suc (length tl) by auto 
 then obtain h2 where l2-def:l2 = h2\# tl by (auto simp add:length-Suc-conv) 
  hence lookup2 tl tl f = Some (tl ! i) 
    using ih l2-def lens by auto 
  hence lookup2 (h1 \# tl) 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: 
\[ l2. \left( \text{lookup2} \; tl \; l2 \; f = \text{Some} \; e; \right. 
\text{length} \; tl = \text{length} \; l2 \] \rightarrow \exists i < (\text{length} \; l2). 
\(e = (l2!i) \land \text{lookup} \; tl \; f = \text{Some} \; ((tl!i))\) 
proof (induct l1) 
case Nil thus ?case by auto 
next 
case (Cons h1 tl) 
  hence length l2 = Suc (length tl) by auto 
 then obtain h2 where l2-def:l2 = h2\# tl 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\# tl) \land e = (h2\# tl) ! 0 \land \text{lookup} \; (h1 \# tl) \; f = \text{Some} \; ((h1 \# tl) ! 0) 
    using asm by (auto simp add:lookup2.simps) 
  hence ?case using l2-def by auto 
} 
moreover { 
  assume asm:¬ (f h1) 
  hence \(\exists i<\text{length} \; tl. \; e = tl ! i \land \text{lookup} \; tl \; f = \text{Some} \; (tl ! i)\) using Cons l2-def by auto 
 then obtain i where \(i<\text{length} \; tl \land e = tl ! i \land \text{lookup} \; tl \; f = \text{Some} \; (tl ! i)\) 
    by auto 
  hence (Suc i) < \text{length}(h2\# tl) \land e = ((h2\# tl) ! (Suc i)) \land \text{lookup} \; (h1\# tl) 
\] \rightarrow \text{Some} \; ((h1\# tl) ! (Suc i)) 
    using asm by (force simp add:lookup2.simps) 
}
hence 
\texttt{?case using l2-def by auto}

\}

ultimately show \texttt{?case by auto}

qed

\textbf{lemma} \texttt{lookup-append}:
\begin{align*}
\text{assumes } & \texttt{lookup } l \ f = \texttt{Some } r \\
\text{shows } & \texttt{lookup } (\texttt{l@l'}) f = \texttt{Some } r \\
\text{using } & \texttt{assms by (induct } l) \texttt{ auto}
\end{align*}

\textbf{lemma} \texttt{method-typings-lookup}:
\begin{align*}
\text{assumes } & \texttt{lookup-eq-Some: lookup } M \ f = \texttt{Some mDef} \\
\text{and } & \texttt{M-ok: CT } \vdash M \texttt{ OK IN } C \\
\text{shows } & \texttt{CT } \vdash \texttt{mDef OK IN } C \\
\text{using } & \texttt{lookup-eq-Some } M\texttt{-ok}
\end{align*}

\textbf{proof} \texttt{(induct } M) 
\begin{align*}
\text{case } & \texttt{Nil thus } \texttt{?case by fastforce} \\
\text{next} & \\
\text{case } \texttt{(Cons } h \ t) \texttt{ thus } \texttt{?case by (cases } f \ h, \texttt{ auto elim:method-typings.cases simp add:lookup.simps)}
\end{align*}

\begin{align*}
\text{qed}\end{align*}

\textbf{2.2.3 Functional}

These lemmas prove that several relations are actually functions

\textbf{lemma} \texttt{mtype-functional}:
\begin{align*}
\text{assumes } & \texttt{mtype } (CT, m, C) = Cs \rightarrow C0 \\
\text{and } & \texttt{mtype } (CT, m, C) = Ds \rightarrow D0 \\
\text{shows } & Ds = Cs \land D0 = C0 \\
\text{using } & \texttt{assms by induct (auto elim:mtype.cases)}
\end{align*}

\textbf{lemma} \texttt{mbody-functional}:
\begin{align*}
\text{assumes } & \texttt{mb1: mbody } (CT, m, C) = xs \cdot e0 \\
\text{and } & \texttt{mb2: mbody } (CT, m, C) = ys \cdot d0 \\
\text{shows } & xs = ys \land e0 = d0 \\
\text{using } & \texttt{assms by induct (auto elim:mbody.cases)}
\end{align*}

\textbf{lemma} \texttt{fields-functional}:
\begin{align*}
\text{assumes } & \texttt{fields } (CT, C) = Cf \\
\text{and } & \texttt{CT OK} \\
\text{shows } & \bigwedge \texttt{Cf'. } [\texttt{fields } (CT, C) = CF] \implies CF = CF' \\
\text{using } & \texttt{assms}
\end{align*}

\textbf{proof} \texttt{induct}
\begin{align*}
\text{case } & \texttt{(f-obj } CT) \\
\text{hence } & \texttt{CT(Object) = None by (auto elim:ct-typing.cases)} \\
\text{thus } & \texttt{?case using f-obj by (auto elim:fields.cases)} \texttt{\textend{align*}

\textbf{next} 
\begin{align*}
\text{case } & \texttt{(f-class } CT C CDef D Cf Dg Dg Cf Dg Cf') \\
\text{hence } & \texttt{f-class-inv:}
\end{align*}
(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) = DyCf’ by fastforce
then obtain Dy’ where
  fields(CT,D) = Dy’
and DyCf’ = Dy’ ⊕ Cf
using f-class-inv c-not-obj by (auto elim:fields.cases)
hence Dy’ = 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;Γ ⊢ +es:Cs shows length es = length Cs
using assms by (induct es Cs) (auto elim:typings.cases)

lemma typings-index:
assumes CT;Γ ⊢ +es:Cs
shows ∀i. [i < length es] → CT;Γ ⊢ (es!i) : (Cs!i)
proof −
  have length es = length Cs using assms by (auto simp: typings-lengths)
  thus ∀i. [i < length es] → CT;Γ ⊢ (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

lemma subtypings-index:
assumes CT ⊢ +Cs <: Ds
shows ∀i. [i < length Cs] → CT ⊢ (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 ⊢ +Cs <: Ds
and CT ⊢ C <: D
shows CT ⊢ +(Cs@[C]) <: (Ds@[D])
using assms

18
by (induct rule:subtypings.induct) (auto simp add:subtypings.intros elim:subtypings.cases)

lemma typings-append:
  assumes CT;Γ ⊢+ 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 $h_1: CT \vdash hCs <: hDs$ and $t_1: CT \vdash tCs <: tDs$ and $Ds = hDs \# tDs$
by (auto elim: subtypings.cases)
then obtain $hEs tEs$
where $h_2: CT \vdash hDs <: hEs$ and $t_2: CT \vdash tDs <: tEs$ and $Es = hEs \# tEs$
using Cons by (auto elim: subtypings.cases)
moreover from subtyping.s-trans[OF $h_1 \ h_2$] have $CT \vdash hCs <: hEs$ by fastforce
moreover with $t_1 \ t_2$ have $CT \vdash tCs <: tEs$ using Cons by simp-all
ultimately show ?case by (auto simp add: subtypings.intros)
qed

lemma ith-typing-sub:
$\forall Cs. [\ CT; \Gamma \vdash + (es@((h\#t)): Cs; \hfill]
  CT; \Gamma \vdash h': C'i;
  CT \vdash C'i <: (Cs!(length es))) ]
\Longrightarrow \exists Cs'. (CT; \Gamma \vdash + (es@((h'\#t)): Cs' \wedge CT \vdash C's' <: Cs)
proof (induct es)
case Nil
then obtain $hCs tCs$
where $ts: CT; \Gamma \vdash \vdash t : tCs$
and $Cs-def: Cs = hCs \# tCs$ by (auto elim: typings.cases)
from $Cs-def Nil$ have $CT \vdash C'i <: hCs$ by auto
with $Cs-def have CT \vdash + (C'i\#tCs) <: Cs$ by (auto simp add: subtypings.ss-cons
subtypings-refl)
moreover from $ts Nil$ have $CT; \Gamma \vdash + (C'i\#tCs) : C'i\#tCs$ 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 $Cs-def: Cs = hCs \# tCs$
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:
$\forall Cs. [\ CT; \Gamma \vdash es: Cs; \ es \in set es] \Longrightarrow \exists C'i. CT; \Gamma \vdash C'i: C'i$
proof (induct es)
case Nil thus ?case by auto
next

20
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 \( CT \vdash Da <: D \) 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 \( e1 \in \text{isubexprs}(e0) \)
  shows \( \forall C. \ [ CT; \text{Map.empty} \vdash e0 : C \] \implies \exists D. CT; \text{Map.empty} \vdash e1 : D \)
  using assms
  by (induct rule:isubexprs.induct) (auto elim:typing.cases simp add:mem-typings)

lemma subexpr-typing:
  assumes \( e1 \in \text{subexprs}(e0) \)
  shows \( \forall C. \ [ CT; \text{Map.empty} \vdash e0 : C \] \implies \exists D. CT; \text{Map.empty} \vdash e1 : D \)
  using assms
  by (induct rule:rtrancl.induct) (auto, force simp add:isubexpr-typing)

lemma isubexpr-reduct:
  assumes \( d1 \in \text{isubexprs}(e1) \)
  shows \( \forall d2. \ [ CT \vdash d1 \rightarrow d2 \] \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 \( d1 \in \text{subexprs}(e1) \)
  shows \( \forall d2. \ [ CT \vdash d1 \rightarrow d2 \] \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:


```plaintext
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 assum
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 assum
proof (induct rule: subtyping.induct)
case (s-refl C CT) show ?case by fact
next
case (s-trans C CT D E)
   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 CDef = M
   by (auto elim: class-typing.cases)
let ?lookup-m = lookup M (λmd. (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 \( CT \vdash mDef \ OK \ IN \ C \) using \( M \ m \) by (auto simp add: method-typings-lookup)
then obtain \( CDef \ m' \ D' \ Cs' \ C0' \) where \( CT \vdash C \)
\( mDef \ OK \ IN \ C \) by (auto simp add: method-typings-lookup)
\( \forall Ds \ D0. \ (mtype(CT,m',D') = Ds \rightarrow D0) \rightarrow Cs' = Ds \land C0' = D0 \)
by (auto elim: method-typing cases)
with \( s-super \) \( mDef-name \) have \( CDef = CDef' \)
\( \land D = D' \)
\( \land m = m' \)
\( \land \forall Ds \ D0. \ (mtype(CT,m,D) = Ds \rightarrow D0) \rightarrow Cs' = Ds \land C0' = D0 \)
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

lemma sub-fields:
assumes \( CT \vdash C < \cdot \cdot \cdot D \)
shows \( \forall Dg. \ fields(CT,D) = Dg \rightarrow \exists Cf. \ fields(CT,C) = (Dg@Cf) \)
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 DDef 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 A1-2:
assumes \( CT \ OK \)
and \( \Gamma = \Gamma I \implies \Gamma 2 \)
and \( \Gamma 2 = [xs \mapsto] Bs \)
and \( \text{length } xs = \text{length } ds \)
and \( \text{length } Bs = \text{length } ds \)
and \( \exists As. \, CT;\Gamma \vdash \vdash ds : As \land CT \vdash As < Bs \)
shows \( CT;\Gamma \vdash \vdash es:Ds \implies \exists Cs. \, (CT;\Gamma 1 \vdash (([ds/xs]e):Cs \land CT \vdash Cs < \vdots Ds) \text{ is } \? \text{TYPINGS } \implies \? P1) \)
and \( CT;\Gamma \vdash e:D \implies \exists C. \, (CT;\Gamma 1 \vdash ((ds/xs)e):C \land CT \vdash C <: D) \text{ is } \? \text{TYPING } \implies \? P2) \)
proof –
let \( \? \text{COMMON-ASMS } = (CT \text{ OK}) \land (\Gamma = \Gamma I \implies \Gamma 2) \land (\Gamma 2 = [xs \mapsto] Bs) \)
\( \land (\text{length } Bs = \text{length } ds) \land (\exists As. \, CT;\Gamma 1 \vdash ds : As \land CT \vdash As < Bs) \)
have RESULT: (\( \? \text{TYPINGS } \implies \? \text{COMMON-ASMS } \implies \? P1) \)
\( \land (\? \text{TYPING } \implies \? \text{COMMON-ASMS } \implies \? P2) \)
proof (induct rule: typings-typing.induct)
case \( \? t\text{-nil } CT \Gamma \)
\( \text{show } \? \text{case} \)
proof (rule impI)
have \( (CT;\Gamma 1 \vdash (\{ds/xs\}[]):[]) \land (CT \vdash [] <: []) \)
by (auto simp add: typings-typing.intros subtypings-typing.intros)
then show \( \exists Cs.(CT;\Gamma 1 \vdash (\{ds/xs\}[]):Cs) \land (CT \vdash Cs <: []) \) by auto
qed
next
\( \text{case}(ts\text{-cons } CT \Gamma e0 C0 es Cs') \)
\( \text{show } \? \text{case} \)
proof (rule impI)
assume asms: \( (CT \text{ OK}) \land (\Gamma = \Gamma I \implies \Gamma 2) \land (\Gamma 2 = [xs \mapsto] Bs) \land (\text{length } Bs = \text{length } ds) \land (\exists As. \, CT;\Gamma 1 \vdash ds : As \land CT \vdash As < Bs) \)
with ts-cons have \( e0\text{-typ } CT;\Gamma \vdash e0 : C0 \) by fastforce
with ts-cons asms have
\( \exists C.(CT;\Gamma 1 \vdash (ds/xs) e0 : C) \land (CT \vdash C <: C0) \)
and \( \exists Cs.(CT;\Gamma 1 \vdash [ds/xs]es : Cs) \land (CT \vdash Cs <: Cs') \)
by auto
then obtain \( C Cs \) where
\( (CT;\Gamma 1 \vdash (ds/xs) e0 : C) \land (CT \vdash C <: C0) \)
and \( (CT;\Gamma 1 \vdash [ds/xs]es : Cs) \land (CT \vdash Cs <: Cs') \) 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-typing.intros)
then show \( \exists Cs. \, CT;\Gamma 1 \vdash \text{map (substs } [xs \mapsto] ds) \, (e0 \# es) : Cs \land CT \vdash Cs <: (C0 \# Cs') \)
by auto
qed
next
\( \text{case } (t\text{-var } \Gamma x C' CT) \)
\( \text{show } \? \text{case} \)
proof (rule impI)
assume asms: \( (CT \text{ OK}) \land (\Gamma = \Gamma I \implies \Gamma 2) \land (\Gamma 2 = [xs \mapsto] Bs) \land (\text{length } Bs = \text{length } ds) \land (\exists As. \, CT;\Gamma 1 \vdash ds : As \land CT \vdash As < Bs) \)
hence
lengths: \( \text{length } ds = \text{length } Bs \)
and \( \text{G-def: } \Gamma = \Gamma 1 ++ \Gamma 2 \)
and \( \text{G2-def: } \Gamma 2 = [xs[\rightarrow]Bs] \) by auto
from lengths G2-def have same-doms: \( \text{dom}([xs[\rightarrow]ds]) = \text{dom}((\Gamma 2)) \) by auto
from asms show \( \exists \ C. \ CT;\Gamma 1 \vdash \text{substs } [xs[\rightarrow]ds] (\text{Var } x) : C \land CT \vdash C <: C' \)
proof (cases \( \Gamma 2 \ x \))
case None
with \( \text{G-def t-var have G1-x: } \Gamma 1 \ x = \text{Some } C' \) by (simp add: map-add-Some-iff)
from None same-doms have \( x \notin \text{dom}([xs[\rightarrow]ds]) \) by (auto simp only: domIff)

hence \( [xs[\rightarrow]ds]|x = \text{None} \) by (auto simp only: map-add-Some-iff)

hence \( (ds/|xs)(\text{Var } x) = (\text{Var } x) \) by auto
with \( \text{G1-x have} \)
\( CT;\Gamma 1 \vdash (ds/|xs)(\text{Var } x) : C' \land CT \vdash C' <: C' \)
by (auto simp add: typings-typing.intros subtyping.intros)
thus \( \text{?thesis by auto} \)

next
case (Some \( Bi \))
with \( \text{G-def t-var have c'-eq-bi: } C' = Bi \) by (auto simp add: map-add-SomeD)
from lengths \( \text{xs = length } ds \) asms have \( \text{length } xs = \text{length } Bs \) by simp
with Some G2-def have \( \exists i. (Bs!i = Bi) \land (i < \text{length } Bs) \land \)
\((\forall i.((\text{length } l = \text{length } Bs)) \rightarrow ([xs[\rightarrow]l]|x = \text{Some } (\text{!!i})))\)
by (auto simp add: map-upds-index)
then obtain \( i \) where \( \text{bs-i-proj: } (Bs!i = Bi) \)
and \( i\text{-len: } i < \text{length } Bs \)
and \( P: \forall [l:\text{exp list}] . ((\text{length } l = \text{length } Bs) \rightarrow ([xs[\rightarrow]l]|x = \text{Some } (\text{!!i})))\)
by fastforce
from lengths \( P \) have subst-x: \( ([xs[\rightarrow]ds]|x = \text{Some } (ds!l)) \) by auto
from asms obtain \( As \) where as-ex: \( CT;\Gamma 1 \vdash + ds : As \land CT \vdash + As <: Bs \) by fastforce

hence \( \text{length } As = \text{length } Bs \) by (auto simp add: subtypings-length)

hence \( \text{proj-i: } CT;\Gamma 1 \vdash ds!l : As!l \land CT \vdash As!l <: Bs!l \)
using i-len lengths as-ex by (auto simp add: typings-proj)

hence \( CT;\Gamma 1 \vdash (ds/|xs)(\text{Var } x) : As!l \land CT \vdash As!l <: C' \)
using c'-eq-bi bs-i-proj subst-x by auto

thus \( \text{?thesis ..} \)
qed

ded

next
case \( t\text{-field } CT \binom{\text{c0}}{C0} Cf \binom{f\text{Def } Ci}{\text{f} } \)
show \( ?\text{case} \)
proof (rule impI)
assume asms: \( (CT \text{ OK}) \land (\Gamma = \Gamma 1 ++ \Gamma 2) \land \)
\((\Gamma 2 = [xs[\rightarrow]Bs]) \land (\text{length } Bs = \text{length } ds) \land (\exists As. CT;\Gamma 1 \vdash + ds : As \land CT \vdash + As <: Bs)\)
from t-field have flds: \( \text{fields}(CT,C0) = Cf \) by fastforce
from t-field asms obtain C where e0-typ: CT;ΓI ⊢ (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;ΓI1 ⊢ (ds/xs)(FieldProj e0 fi) : Ci
by(simp add:typings-typingintros)
moreover have CT ⊢ Ci <: Ci by (simp add:subtypingintros)
ultimately show ∃ C. CT;ΓI ⊢ (ds/xs)(FieldProj e0 fi) : C ∧ CT ⊢ C <: C
by auto
next
case(t-invk CT Γ e0 C0 m Ds Cs)
show ?case
proof (rule impI)
assume asms: (CT OK) ∧ (Γ = ΓI ++ Γ2) ∧ (Γ2 = [xs [→] Bs])
∧ (length Bs = length ds) ∧ (∃ As. CT;ΓI ⊢+ ds : As ∧ CT ⊢+ As <: B$s)
hence ct-ok: CT OK ..
from t-invk have mtyp: mtype(CT,m,C0) = Ds → C
and subs: CT ⊢+ Cs <: Ds
and lens: length es = length Ds
by auto
from t-invk asms obtain C′ where
e0-typ: CT;ΓI ⊢ (ds/xs)e0 : C′ and sub′: CT ⊢ C′ <: C0 by auto
from t-invk asms obtain Cs′ where
es-typ: CT;ΓI ⊢+ [ds/xs]es : Cs′ and subs′: CT ⊢+ 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-I-1[OF sub′ ct-ok mtyp]
es-typ
subtypings-trans[OF sub′s subs]
lens
subst-e
have CT;ΓI ⊢ (ds/xs)(MethodInvk e0 m es) : C by(auto simp add:typings-typingintros)
moreover have CT ⊢ C <: C by(simp add:subtypingintros)
ultimately show ∃ C′. CT;ΓI ⊢ (ds/xs)(MethodInvk e0 m es) : C′ ∧ CT ⊢ C′ <: C by auto
qed
next
case(t-new CT C Df Es Ds Γ Cs)
show ?case
proof (rule impI)
  assume asms: \((CT \ OK) \land (\Gamma = \Gamma_1 \vdash \vdash \Gamma_2) \land (\Gamma_2 = [xs \mapsto Bs]) \land (\text{length } Bs = \text{length } ds) \land (\exists As. CT; \Gamma_1 \vdash + ds : As \land CT \vdash + As <: Bs)\)
  hence ct-ok: \(CT \ OK\).
from t-new have
  \(subs: CT \vdash + Cs <: Ds\)
  and \(flds: fields(CT,C) = Df\)
  and \(len: \text{length } es = \text{length } Df\)
  and \(vlts: \text{varDefs-types } Df = Ds\)
  by auto
from t-new asms obtain Cs' where
  es-typ: \(CT; \Gamma_1 \vdash (\text{ds/xs})es : Cs'\) and \(subs': CT \vdash + Cs <: Cs\) by auto
have subst-e: \((\text{ds/xs})(\text{New } C es) = \text{New } C (\text{ds/xs})es\)
by (auto simp add: subst-list2-eq-map-substs)
from es-typ subtypings-trans[OF subs' subs] flds subst-e len vlts
have \(CT; \Gamma_1 \vdash (\text{ds/xs})(\text{New } C es) : C\) by (auto simp add: typings-typing.intros)
moreover have \(CT \vdash C <: C\) by (simp add: subtyping.intros)
ultimately show \(\exists C'. CT; \Gamma_1 \vdash (\text{ds/xs})(\text{New } C es) : C' \land CT \vdash C' <: C\)
by auto
qed

next
  case (t-ucast CT Γ e0 D C)
  show \(?case\)
proof (rule impI)
  assume asms: \((CT \ OK) \land (\Gamma = \Gamma_1 \vdash \vdash \Gamma_2) \land (\Gamma_2 = [xs \mapsto Bs]) \land (\text{length } Bs = \text{length } ds) \land (\exists As. CT; \Gamma_1 \vdash + ds : As \land CT \vdash + As <: Bs)\)
  from t-ucast asms obtain C' where e0-typ: \(CT; \Gamma_1 \vdash (\text{ds/xs})e0 : C'\)
  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 \(CT; \Gamma_1 \vdash (\text{ds/xs})(\text{Cast } C e0) : C\) by (auto simp add: typings-typing.intros)
moreover have \(CT \vdash C <: C\) by (rule s-refl)
ultimately show \(\exists C'. CT; \Gamma_1 \vdash (\text{ds/xs})(\text{Cast } C e0) : C' \land CT \vdash C' <: C\)
by auto
qed

next
  case (t-dcast CT Γ e0 D C)
  show \(?case\)
proof (rule impI)
  assume asms: \((CT \ OK) \land (\Gamma = \Gamma_1 \vdash \vdash \Gamma_2) \land (\Gamma_2 = [xs \mapsto Bs]) \land (\text{length } Bs = \text{length } ds) \land (\exists As. CT; \Gamma_1 \vdash + ds : As \land CT \vdash + As <: Bs)\)
  from t-dcast asms obtain C' where e0-typ: \(CT; \Gamma_1 \vdash (\text{ds/xs})e0 : C'\) by auto
  have \((CT \vdash C' <: C) \lor (C \neq C' \land CT \vdash C <: C') \lor (CT \vdash C \leftarrow; C' \land CT \vdash C' \leftarrow; C)\) by blast
moreover
  \{
  assume CT \vdash C' <: C
  with \(\text{e0-typ}\) have \(CT; \Gamma_1 \vdash (\text{ds/xs})(\text{Cast } C e0) : C\) by (auto simp add:
typings-typing.intros)
}

moreover
{ assume (C ≠ C’ ∧ CT ⊢ C <: C’)
  with e0-typ have CT;ΓI ⊢ (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;ΓI ⊢ (ds/xs) (Cast C e0) : C by (auto simp add:
typings-typing.intros)
}

ultimately have CT;ΓI ⊢ (ds/xs) (Cast C e0) : C by auto
moreover have CT ⊢ C <: C’ by (rule s-refl)
ultimately show ∃ C’. CT;ΓI ⊢ (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 = ΓI ++ Γ2) ∧ (Γ2 = [xs ↦→ Bs]) ∧ (length
Bs = length ds) ∧ (∃ As. CT;ΓI ⊢+ ds : As ∧ CT ⊢+ As ¬<: Bs)
from t-scast asms obtain C’ where e0-typ:CT;ΓI ⊢ (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;ΓI ⊢ (ds/xs) (Cast C e0) : C using e0-typ by (auto
simp add: typings-typing.intros)
thus ∃ C’. CT;ΓI ⊢ (ds/xs)(Cast C e0) : C’ ∧ CT ⊢ C’ <: C by (auto simp
add: s-refl)
qed

thus ?TYPINGS ⇒ ?P1 and ?TYPING ⇒ ?P2 using asms 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;ΓI++Γ2 ⊢+ es : Cs) (is ?P1 ⇒ ?P2)
and $CT; \Gamma_1 \vdash e : C \Rightarrow CT; \Gamma_1 + + \Gamma_2 \vdash e : C$ (is $?Q_1 \Rightarrow ?Q_2$)

proof

- have $(?P_1 \Rightarrow ?P_2) \land (?Q_1 \Rightarrow ?Q_2)$
  by (induct rule: typings-typing, induct, auto simp add: map-add-find-right typings-typing.intros)

  thus $?P_1 \Rightarrow ?P_2$ and $?Q_1 \Rightarrow ?Q_2$ by auto

qed

3.5 Method Body Typing Lemma

lemma A-1-4:

assumes ct-ok: $CT$ OK
and mb:mbody($CT,m,C$) = $xs \cdot e$
and mt:mtype($CT,m,C$) = $Ds \Rightarrow D$
shows $\exists D_0 C_0. (CT \vdash C < : D_0) \land$
  $(CT \vdash C_0 < : D) \land$
  $(CT; [xs \mapsto Ds] \{this \mapsto D_0\} \vdash e : C_0)$
using mb ct-ok mt proof (induct rule: mbody.induct)
case (mb-class $CT$ $C$ $CDef$ $m$ $mDef$ $zs$ $e$)

hence
  $m-param: varDefs-typings (mParams mDef) = Ds$
and $m-ret:mmReturn mDef = D$
and $CT \vdash CDef OK$
and $cName CDef = C$
by (auto elim:mtype.cases ct-typing.cases)

hence $CT \vdash (\text{cMethods CDef})$ 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 $\exists E_0. ((CT; [xs \mapsto Ds, this\mapsto C] \vdash e : E_0) \land (CT \vdash E_0 < : D))$
using mb-class $m-param$ $m-ret$ by (auto elim:method-typing.cases)

then obtain $E_0$

where $CT; [xs \mapsto Ds, this\mapsto C] \vdash e : E_0$
and $CT \vdash E_0 < : D$
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; $mtype(CT,m,Da) = Ds \Rightarrow D]$
  $\Rightarrow \exists D_0 C_0. (CT \vdash Da < : D_0) \land (CT \vdash C_0 < : C)$
  $\land (CT; [xs \mapsto Ds, this \mapsto D_0] \vdash e : C_0)$ 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($CT,m,Da$) = $Ds \Rightarrow D$ by (auto elim: mtype.cases)

from IH[OF ct mt] obtain $D_0 C_0$

where $s_1$: $CT \vdash Da < : D_0$
and $CT \vdash C_0 < : D$
and $CT; [xs \mapsto Ds, this \mapsto D_0] \vdash e : C_0$ by auto

thus $?case using s-trans[OF c-sub-da}$ $s_1]$ by blast

qed
3.6 Subject Reduction Theorem

theorem Thm-2-4-1:
  assumes $CT \vdash e \rightarrow e'$
  and $CT \text{ OK}$
  shows $\forall C. \ [ CT;\Gamma \vdash e : C ] \\
  \implies \exists C'. (CT;\Gamma \vdash e' : C' \land CT \vdash C' <: C)$
using assms
proof (induct rule: reduction_induct)
case (r-field $CT \ C_a \ C_f \ es \ fi \ e'$)
hence $CT;\Gamma \vdash \text{FieldProj (New Ca es)} \ fi : C$
  and ct-ok: $CT \text{ OK}$
  and fls: fields$(CT, Ca) = Cf$
  and lkup2: lookup2 $Cf \ es$ $(\lambda fd. \ vdName fd = fi) = \text{Some } e'$
then obtain $Ca' C_f' fDef$
  where new-typ: $CT;\Gamma \vdash \text{New Ca es} : Ca'$
  and flds': fields$(CT, Ca') = C_f'$
  and lkup: lookup $C_f'$ $(\lambda fd. \ vdName fd = fi) = \text{Some } fDef$
  and C-def: $\vdType fDef = C$ by (auto elim: typing_cases)
hence $Ca-Ca'$: $Ca = Ca'$
with fls' have $C_f-C_f': C_f = C_f'$
by (auto simp add: fields-functional[OF fls ct-ok])
from new-typ obtain $Cs \ Ds \ C_f''$
  where fields$(CT, Ca') = C_f''$
  and es-typs: $CT;\Gamma \vdash+ es:Cs$
  and Ds-def: varDefs-types $C_f'' = Ds$
  and length-$Cf-es$: $\text{length } C_f'' = \text{length es}$
  and subs: $CT \vdash+ Cs <: Ds$
by (auto elim: typing_cases)
with $Ca-Ca'$ have $C_f-C_f'': C_f = C_f''$
by (auto simp add: fields-functional[OF fls ct-ok])
from length-$Cf-es$ $C_f-C_f''$ lookup2-index[OF lkup2] obtain $i$ where
  i-bound: $i < \text{length es}$
  and $e' = e[i]
  and lookup $Cf$ $(\lambda fd. \ vdName fd = fi) = \text{Some } (Cf[i])$
by auto
moreover with $C$-def $Ds$-def lkup lkup2 have $Ds!i = C$
  using $Ca-Ca'$ $C_f$-$C_f''$ i-bound length-$Cf-es$ fls'
  by (auto simp add: nth-map varDefs-types-def fields-functional[OF fls 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 $CT \ m \ C_a \ xs \ es \ es' e$
from r-invk have $mb: \text{mbody}(CT,m, Ca) = xs . e$ by fastforce
from r-invk obtain $Ca' Ds \ Cs$
  where $CT;\Gamma \vdash \text{New Ca es} : Ca'$
  and $mtype(CT,m,Ca') = Cs \rightarrow C$
  and $Ds$-subs: $CT \vdash+ Ds <: Cs$
and \( \Gamma \): length \( ds = length C \) by (auto elim:typing.cases)

hence new-typ: \( CT,\Gamma \vdash \text{New Ca} es : Ca \)

and mt: mtype\((CT,m,\text{Ca}) = Cs \rightarrow C \) by (auto elim:typing.cases)

from ds-typs new-typ have CT,\Gamma \vdash (ds \circ (@\text{New Ca} es)) : (Dss\circ(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\rightarrow]Cs,this\rightarrow Da\vdash e : E by auto

moreover with Dss have CT \vdash (Ds\circ(Ca)) <: (Cs\circ(Da))

by (auto simp add: subtyping-append)

ultimately have ex: \( \exists As. \ CT;\Gamma \vdash (ds \circ([New \ Ca \ es]) : As \land CT \vdash As <: (Cs\circ(Da))

by auto

from e0-typ1 have e0-typ2: CT;((\Gamma ++ [xs\rightarrow]Cs,this\rightarrow Da)) \vdash e : E

by (simp only:A-1-3)

from e0-typ2 mtype-mbody-length[OF mt mb]

have e0-typ3: \( CT;((\Gamma ++ ([xs@this])[\rightarrow](Cs\circ(Da))) \vdash e : E \)

by (force simp only:map-shuffle)

let \( ?\Gamma 1 = \Gamma \) and \( ?\Gamma 2 = ([xs@this])[\rightarrow](Cs\circ(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 \( \Gamma \) mtype-mbody-length[OF mt mb]

obtain E' where e'-typ: \( CT;\Gamma \vdash \text{substs} ([xs@this])[\rightarrow]([ds\circ([New \ Ca \ es])] e : E'

and E'-sub-E: CT \vdash E' <: E by force

moreover from e'-typ \( \Gamma \) mtype-mbody-length[OF mt mb]

have CT;\Gamma \vdash \text{substs} [xs\rightarrow]ds,this\rightarrow (\text{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 \text{New Ca} es : Ca' by (auto elim:typing.cases)

thus ?case using r-cast by (auto elim:typing.cases)

next

case (r-field CT e0 e0' f)

then obtain C0 Cf fd where CT;\Gamma \vdash e0 : C0

and Cf-def: fields(CT,\text{C},C0) = Cf

and fd-def: lookup Cf (\lambda fd. (vdName fd = f)) = Some fd

and vldType fd = C

by (auto elim:typing.cases)

moreover with r-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 \( \text{fields}(CT, C') = (\text{Cf} \circ \text{Cf'}) \) by rule (rule \( \vdash CT \vdash C' : C0 )

moreover with \( \text{fd-def} \) have lookup \( (\text{Cf} \circ \text{Cf'}) (\lambda \text{fd. (vdName fd = f)}) = \text{Some fd} \)

by (simp add:lookup-append)

ultimately have \( CT, \Gamma \vdash \text{FieldProj e0'} f : C \) by (auto simp add: typings-typing.t-field)

thus \( ?\text{case} \) by (auto simp add: subtyping.s-refl)

next

case \((\text{rc-invk-rev} CT \ e0 \ e0' \ m \ es \ C)\)

then obtain \( C0 \ Ds \ Cs \)

where \( \text{ct-ok}; CT \ OK \)

and \( CT, \Gamma \vdash e0 : C0 \)

and \( \text{mt: mtype}(CT, m, C0) = Ds \rightarrow C \)

and \( CT, \Gamma \vdash es : Cs \)

and \( \text{length } es = \text{length } Ds \)

and \( CT \vdash Cs <: Ds \)

by (auto elim:typing.cases)

moreover with \( \text{rc-invk-rev} \) obtain \( C0' \)

where \( CT, \Gamma \vdash e0' : C0' \)

and \( CT \vdash C0' <: C0 \) by auto

moreover with \( \text{A-1-1}[\text{OF - ct-ok } \text{mt}] \) have \( \text{mtype}(CT, m, C0') = Ds \rightarrow C \) by simp

ultimately have \( CT, \Gamma \vdash \text{MethodInvk e0' m es} : C \) by (auto simp add: typings-typing.t-invk)

thus \( ?\text{case} \) by (auto simp add: subtyping.s-refl)

next

case \((\text{rc-invk-arg} CT \ ei \ ei' \ e0 \ m \ er \ C)\)

then obtain \( Cs \ Ds \ C0 \)

where \( \text{tytps: } CT, \Gamma \vdash+ (e0@\text{(ei#er)}) : Cs \)

and \( e0\text{-typ: } CT, \Gamma \vdash e0 : C0 \)

and \( \text{mt: mtype}(CT, m, C0) = Ds \rightarrow C \)

and \( \text{Cs-sub-Ds: } CT \vdash+ Cs <: Ds \)

and \( \text{len: length } (e0@\text{(ei#er)}) = \text{length } Ds \)

by (auto elim:typing.cases)

hence \( CT, \Gamma \vdash ei; (\text{Cs!}(\text{length } el)) \) by (simp add:ith-padding)

with \( \text{rc-invk-arg} \) obtain \( Ci' \)

where \( \text{ei-typ: } CT, \Gamma \vdash ei'; Ci' \)

and \( Ci\text{-sub: } CT \vdash Ci' <: (\text{Cs!}(\text{length } el)) \)

by auto

from \( \text{ith-padding-sub}[\text{OF tytps } \text{ei-typ } Ci\text{-sub}] \) obtain \( Cs' \)

where \( \text{es'-typs: } CT, \Gamma \vdash+ (e0@\text{(ei'#er)}) : Cs' \)

and \( Cs'\text{-sub-Cs: } CT \vdash+ Cs' <: Cs \) by auto

from \( \text{len have length } (e0@\text{(ei'#er)}) = \text{length } Ds \) by simp

with \( \text{es'-typs subtypings-trans}[\text{OF Cs'-sub-Cs Cs-sub-Ds}] \) \( e0\text{-typ } \text{mt} \) have

\( CT, \Gamma \vdash \text{MethodInvk e0 m (e0@\text{(ei'#er)}) : C} \)

by (auto simp add: typings-typing.t-invk)

thus \( ?\text{case} \) by (auto simp add: subtyping.s-refl)

next

case \((\text{rc-new-arg} CT \ ei \ ei' \ Ca \ er \ C)\)

then obtain \( Cs \ Df \ Ds \)

where \( \text{tytps: } CT, \Gamma \vdash+ (e0@\text{(ei#er)}) : Cs \)
and flds: fields\((CT,C) = Df\)
and len: \(\text{length} \ (el@ei\#er)) = \text{length} \ Df\)
and Ds-def: \(\text{varDefs-types} \ Df = Ds\)
and Cs-sub-Ds: \(CT \vdash Cs <: Ds\)
and C-def: \(Ca = C\)

by (auto elim:typing.cases)
hence \(CT;\Gamma \vdash ei:\ (Cs[\text{length} \ el])\) by (simp add:ith-typing)

with rc-new-arg obtain \(Ci'\)
where ei-typ: \(CT;\Gamma \vdash ei':Ci'\)
and Ci-sub: \(CT \vdash Ci' <: (\text{Cs[\text{length} \ el])}\)
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 \(\text{length} \ (el@ei'\#er)) = \text{length} \ Df\) by simp
with es'-typs substypings-trans[of Cs'-sub-Cs Cs-sub-Ds] flds Ds-def C-def have
\(CT;\Gamma \vdash \text{New} \ Ca \ (el@ei'\#er)) : C\)
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 \((CT \vdash D' <: C) \lor\)
\((C \neq D' \land CT \vdash C <: D') \lor\)
\((CT \vdash C \sim:<; D' \land CT \vdash D' \sim:<; C)\) by blast

moreover {
  assume CT \vdash D' <: C
  with e0'-typ have \(CT;\Gamma \vdash \text{Cast} \ C e0' : C\) by (auto simp add:typings-typing.t-ucast)
}

moreover {
  assume \((C \neq D' \land CT \vdash C <: D')\)
  with e0'-typ have \(CT;\Gamma \vdash \text{Cast} \ C e0' : C\) by (auto simp add:typings-typing.t-dcast)
}

moreover {
  assume \((CT \vdash C \sim:<; D' \land CT \vdash D' \sim:<; C)\)
  with e0'-typ have \(CT;\Gamma \vdash \text{Cast} \ C e0' : C\) by (auto simp add:typings-typing.t-scast)
}

ultimately have \(CT;\Gamma \vdash \text{Cast} \ C e0' : C\) 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 $\forall C. \ [\ CT;\Gamma \vdash \ e : C \] \implies \exists C'. \ (CT;\Gamma \vdash \ e' : C' \land 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: $\forall D. \ [\ CT;\Gamma \vdash \ e' \ : \ D; \ CT \ OK] \implies \exists E. \ CT;\Gamma \vdash \ e'' \ : \ E \land CT \vdash E \ < D$
  by auto

  from Thm-2-4-1[OF e-step ct-ok e-typ]
  obtain D where \_ __ \_ \_ \_ \ by auto

  with IH[OF e'-typ ct-ok] obtain E where CT;\Gamma \vdash \ e'' : E and E-sub-D: $CT \vdash E \ < D$ by auto

  moreover from s-trans[OF E-sub-D D-sub-C]
  have $CT \vdash E \ < D$ 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 $CT;\text{Map.\emptyset} \vdash \ e : C$

and FieldProj (New C0 es) $\bar{f} \in \text{subexprs}(e)$

shows $\exists \text{fDef} \ \cdot \ \text{fields}(CT, C0) = Cf \land \text{lookup C} \ (\lambda \text{fd} \cdot (\text{vdName fd} = \bar{f})) = \text{Some fDef}$

proof --

obtain Ci where $CT;\text{Map.\emptyset} \vdash (\text{FieldProj (New C0 es) \bar{f}}) : Ci$
using assms by (force simp add: subexpr-typing)

then obtain Cf fDef C0'
where $CT;\text{Map.\emptyset} \vdash (\text{New C0 es}) : C0'$
and fields(CT, C0') = Cf
and lookup Cf (\lambda fd. (vdName fd = \bar{f})) = 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 $CT;\text{Map.\emptyset} \vdash \ e : C$

and MethodInvk (New C0 es) $m \text{ ds} \in \text{subexprs}(e)$

shows $\exists \text{xs e0. mbody}(CT,m,C0) = \text{xs . e0} \land \text{length xs} = \text{length ds}$

proof --
obtain $D$ where $\text{CT};\text{Map.empty} \vdash \text{MethodInvk}(\text{New C0 es}) m \text{ ds} : D$

using assms by (force simp add:subexpr-typing)

then obtain $C0'$ $Cs$

where $\text{CT};\text{Map.empty} \vdash (\text{New C0 es}) : C0'$

and $\text{mt}:\text{mtype}(\text{CT},m,C0') = Cs \rightarrow D$

and $\text{length ds} = \text{length Cs}$

by (auto elim:typing.cases)

with $\text{mtype-nobody}(\text{OF} \text{ mt})$ show $\text{thesis}$ by (force elim:typing.cases)

qed

lemma closed-subterm-split:

assumes $\text{CT};\Gamma \vdash e : C$ and $\Gamma = \text{Map.empty}$

shows

$(\exists C0 \text{ es } fi. \ (\text{FieldProj} \ (\text{New C0 es} f)) \in \text{subexprs}(e))$

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

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

$\lor \text{val}(e) \ (\text{is} \ ?F e \lor ?M e \lor ?C e \lor ?V e \text{ is} \ ?\text{IH} e)$

using assms

proof (induct $C T \Gamma e C$ rule:typing-induct)

  case 1 thus $?case$ using assms by auto

next

  case (2 $C \ C T \Gamma x$) 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 $?\text{IH} e0$ by auto

  moreover

  { assume $?F e0$
    then obtain $C0 \text{ es } fi'$ where $s2: \text{FieldProj} \ (\text{New C0 es} f) \in \text{subexprs}(e0)$ by auto

    from rtrancl-trans[OF $s2$ $s1$] have $?case$ by auto
  }

  moreover {
    assume $?M e0$
    then obtain $C0 \text{ es m ds where } s2: \text{MethodInvk} \ (\text{New C0 es}) m \text{ ds} \in \text{subexprs}(e0)$ by auto

    from rtrancl-trans[OF $s2$ $s1$] have $?case$ by auto
  }

  moreover {
    assume $?C e0$
    then obtain $C0 D \text{ es where } s2: \text{Cast} D \ (\text{New C0 es}) \in \text{subexprs}(e0)$ by auto

    from rtrancl-trans[OF $s2$ $s1$] have $?case$ by auto
  }

  moreover {
    assume $?V e0$
    then obtain $C0 \text{ es where } e0 = (\text{New C0 es}) \text{ and } \text{vals(es)}$ by (force elim:val.cases)

    hence $?case$ by (force intro:isubexprs.intros)
  }

ultimately show $?case$ by blast

next

  case (4 $C0 CT Cs Ds \Gamma e0 \text{ es m}$)

  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: 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:subexprs.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 \( \exists i < \text{length } t \) \( (\exists F (t!i) \lor \exists M (t!i) \lor \exists C (t!i)) \)  
  then obtain \( i \)  
    where \( i < \text{length } t \)  
    and \( \exists F (t!i) \lor \exists M (t!i) \lor \exists C (t!i) \) by force  
  hence \( (\text{Suc } i < \text{length } (\text{Suc } i)) \land (\exists F ((\text{Suc } i)!(\text{Suc } i)) \lor \exists M ((\text{Suc } i)!(\text{Suc } i)) \lor \exists C ((\text{Suc } i)!(\text{Suc } i)) \) by auto  
  hence \( \exists i < \text{length } (\text{Suc } i) \). \( (\exists F ((\text{Suc } i)!) \lor \exists M ((\text{Suc } i)!) \lor \exists C ((\text{Suc } i)!) \)  
.  
  hence \( \exists Q ((\text{Suc } i)!) \) by auto  
} moreover {  
  assume \( v-t: \text{vals}(t) \)  
  from \( OIH[\text{OF - h-typ } G-\text{def}] \) have \( \exists IH \ h \) by auto  
  moreover {  
    assume \( \exists F \ h \lor \exists M \ h \lor \exists C \ h \)  
    hence \( \exists F ((\text{Suc } i)!) \lor \exists M ((\text{Suc } i)!) \lor \exists C ((\text{Suc } i)!) \) by auto  
    hence \( \exists Q ((\text{Suc } i)!) \) by force  
  } moreover {  
    assume \( \exists V \ h \)  
    with \( v-t \) have \( \text{vals}((\text{Suc } i)!) \) by \( \text{force intro:vals-val.intro}\)  
    hence \( \exists Q ((\text{Suc } i)!) \) by auto  
  } ultimately have \( \exists Q ((\text{Suc } i)!) \) by blast  
} ultimately show \( \exists Q ((\text{Suc } i)!) \) by blast  
qed  
moreover {  
  assume \( \exists i < \text{length } \text{es} \). \( \exists F (\text{es}!i) \lor \exists M (\text{es}!i) \lor \exists C (\text{es}!i) \)  
  then obtain \( i \) where \( i < \text{length } \text{es} \) and \( r: \exists F (\text{es}!i) \lor \exists M (\text{es}!i) \lor \exists C (\text{es}!i) \) by force  
    from \( \text{ith-mem}[\text{OF } i\text{-len}] \) have \( s1: \text{es}!i \in \text{subexprs}(\text{New } C \text{ es}) \) by \( \text{auto intro:subexprs.se-newarg}\)  
    {  
      assume \( \exists F (\text{es}!i) \)  
      then obtain \( C0 \ \text{es}'!fi \) where \( s2: \text{FieldProj } (\text{New } C0 \text{ es}') \ fi \in \text{subexprs}(\text{es}!i) \) by auto  
    } moreover {  
      assume \( \exists M (\text{es}!i) \)  
      then obtain \( C0 \ \text{es}'!m'!ds \) where \( s2: \text{MethodInvk } (\text{New } C0 \text{ es}') \ m'! ds \in \text{subexprs}(\text{es}!i) \) by force  
    } moreover {  
      assume \( \exists C (\text{es}!i) \)  
      then obtain \( C0 \ D \ es'!w \) where \( s2: \text{Cast } D \ (\text{New } C0 \text{ es}') \ es'!w \in \text{subexprs}(\text{es}!i) \) by auto  
    } ultimately have \( \exists F (\text{New } C \text{ es}) \lor \exists M (\text{New } C \text{ es}) \lor \exists C (\text{New } C \text{ es}) \) using \( r \) by blast  

hence \(?case\ \text{by } auto\)
} moreover {
assume vals(es)
hence \(?case\ \text{by } (auto\ \text{intro:vals-val.intro})\)
} ultimately show \(?case\ \text{by } blast\)

next
case \((6 \ C\ CT\ D\ \Gamma\ e0)\)
have \(s1: e0 \in \text{subexprs(Cast C e0)}\) by (auto simp add:isubexprs.intros)
from \(6\) have \(?IH e0\ \text{by } auto\)
moreover {
assume \(?F e0\)
then obtain \(C0\ es\ \text{fi where } s2: \text{FieldProj (New C0 es)} \ \text{fi} \in \text{subexprs(e0)}\) by auto
from rtrancl-trans[OF \(s2\ s1\)] have \(?case\ \text{by } auto\)
} moreover {
assume \(?M e0\)
then obtain \(C0\ es\ m\ ds\ \text{where } s2: \text{MethodInvk (New C0 es)} \ m\ ds \in \text{subexprs(e0)}\) by auto
from rtrancl-trans[OF \(s2\ s1\)] have \(?case\ \text{by } auto\)
} moreover {
assume \(?C e0\)
then obtain \(C0\ D'\ es\ \text{where } s2: \text{Cast D'} (\text{New C0 es}) \ (\text{New C0 es'}) \ \text{by } (force\ elim:val.cases)\)
hence \(?case\ \text{by } (force\ \text{intro:isubexprs.intros})\)
} ultimately show \(?case\ \text{by } blast\)

next
case \((7 \ C\ CT\ D\ \Gamma\ e0)\)
have \(s1: e0 \in \text{subexprs(Cast C e0)}\) by (auto simp add:isubexprs.intros)
from \(7\) have \(?IH e0\ \text{by } auto\)
moreover {
assume \(?F e0\)
then obtain \(C0\ es\ \text{fi where } s2: \text{FieldProj (New C0 es)} \ \text{fi} \in \text{subexprs(e0)}\) by auto
from rtrancl-trans[OF \(s2\ s1\)] have \(?case\ \text{by } auto\)
} moreover {
assume \(?M e0\)
then obtain \(C0\ es\ m\ ds\ \text{where } s2: \text{MethodInvk (New C0 es)} \ m\ ds \in \text{subexprs(e0)}\) by auto
from rtrancl-trans[OF \(s2\ s1\)] have \(?case\ \text{by } auto\)
} moreover {
assume \(?C e0\)
then obtain \(C0\ D'\ es\ \text{where } s2: \text{Cast D'} (\text{New C0 es}) \in \text{subexprs(e0)}\) by auto
\textbf{3.9 Type Soundness Theorem}

\textbf{Theorem} Thm-2.4-3:
\begin{itemize}
\item \textbf{assumes} e-typ: CT; Map.\emptyset \vdash e : C
\item e-typ: CT; Map.\emptyset \vdash e \rightarrow e1
\item \textbf{and} multisteps: CT \vdash e \rightarrow* e1
\item \textbf{and} no-step: \neg(\exists e2. CT \vdash e1 \rightarrow e2)
\item \textbf{shows} (val(e1) \land (\exists D. CT; Map.\emptyset \vdash e1 : D \land CT \vdash D <: C))
\item \lor (\exists D C es. (Cast D (New C es) \in subexprs(e1) \land CT \vdash C \leftarrow C :< D))
\end{itemize}

\textbf{proof} –
\begin{itemize}
\item \textbf{from} \text{assms} Cor-2.4-1-multi[OF multisteps e-typ e-typ] \textbf{obtain} C1
\item \textbf{where} e1-typ: CT; Map.\emptyset \vdash e1 : C1
\item \textbf{and} C1-sub-C: CT \vdash C1 <: C \textbf{by} auto
\end{itemize}
from e1-typ have \((\exists C0\ es\ fi.\ \text{(FieldProj (New C0 es) fi)) \in \text{subexprs}(e1))\)
\(\lor (\exists C0\ es\ m\ ds.\ \text{(MethodInuk (New C0 es) m ds)) \in \text{subexprs}(e1))\)
\(\lor (\exists C0\ D\ es.\ \text{(Cast D (New C0 es))} \in \text{subexprs}(e1))\)
\(\lor \text{val}(e1)\) is \(?F e1 \lor ?M e1 \lor ?C e1 \lor ?V e1\) by (simp add: closed-subterm-split)
moreover
\{ assume \(?F e1\)
then obtain C0 es fi where fp: FieldProj (New C0 es) fi \in \text{subexprs}(e1) by auto
then obtain Ci where CT:Map.empty \vdash \text{FieldProj (New C0 es) fi : Ci using e1-typ by(force simp add:subexpr-typing)}
then obtain C0’ where new-typ: CT:Map.empty \vdash \text{New C0 es : C0’ by (force elim: typing.cases)}
hence C0 = C0’ by (auto elim:typ-ing.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 (\(\lambda fd.\ \text{vdName fd = fi}\)) = 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) \(\implies\) lookup2 Cf es (\(\lambda fd.\ \text{vdName fd = fi}\)) = Some
\((e s ! i)\) by auto
ultimately have lookup2 Cf es (\(\lambda fd.\ \text{vdName fd = fi}\)) = Some (es!i) by auto
with f2 have CT \vdash \text{FieldProj (New C0 es) fi \to (es!i) by(auto intro:reduction.intros)}
with fp have \(\exists e2.\ CT \vdash e1 \to e2\) by(simp add:subexpr-reduct)
with no-step have ?thesis by auto
\} moreover \{ assume \(?M e1\)
then obtain C0 es m ds where mi:MethodInuk (New C0 es) m ds \in \text{subexprs}(e1) by auto
then obtain D where CT:Map.empty \vdash MethodInuk (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 \vdash \text{New C0 es : C0’}
and mtype(CT,m,C0’) = Es \to 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 \vdash (MethodInuk (New C0 es) m ds) \to (\text{subs}[xs[\to]ds,\text{this} \to (New C0 es)]e0) by(auto simp add:reduction.intros)
with mi have \(\exists e2.\ CT \vdash e1 \to e2\) 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) \in \text{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 \text{rc-invk-arg}': CT \vdash ei \rightarrow ei' \implies append el (ei \# er) e' \implies append el (ei' \# er) e'' \implies MethodInvk e m e' \implies MethodInvk e m e''
\text{unfolding append-def by simp (rule reduction.intros(6))}

lemma \text{rc-new-arg}': CT \vdash ei \rightarrow ei' \implies append el (ei \# er) e \implies append el (ei' \# er) e'
==\implies CT \vdash New C e \rightarrow New C e'
\text{unfolding append-def by simp (rule reduction.intros(7))}

lemmas [\text{code-pred-intro}] = reduction.intros(1–5)
\text{rc-invk-arg' rc-new-arg' reduction.intros(8)}

code-pred (modes: i => i => i => bool, i => i => o => bool as reduce)
\text{reduction}
\text{proof –}
\text{\hspace{1em} case append}
\text{\hspace{1em} from this show thesis}
\text{\hspace{1em} unfolding append-def by (cases xa) fastforce+}
\text{next}
\text{\hspace{1em} case reduction}
\text{\hspace{1em} from reduction.prems show thesis}
\text{\hspace{1em} proof (cases rule: reduction.cases)}
\text{\hspace{1em} case r-field}
\text{\hspace{1em} with reduction(1) show thesis by fastforce}
\text{next}
\text{\hspace{1em} case r-invk}
\text{\hspace{1em} with reduction(2) show thesis by fastforce}
\text{next}
\text{\hspace{1em} case r-cast}
\text{\hspace{1em} with reduction(3) show thesis by fastforce}
\text{next}
\text{\hspace{1em} case rc-field}
\text{\hspace{1em} with reduction(4) show thesis by fastforce}
\text{next}
\text{\hspace{1em} case rc-invk-rev}
\text{\hspace{1em} with reduction(5) show thesis by fastforce}
\text{next}
\text{\hspace{1em} case rc-invk-arg}
\text{\hspace{1em} with reduction(6) show thesis}
\text{\hspace{1em} unfolding append-def by fastforce}
\text{next}
\text{\hspace{1em} case rc-new-arg}
\text{\hspace{1em} with reduction(7) show thesis}
\text{\hspace{1em} unfolding append-def by fastforce}
\text{next}
\text{\hspace{1em} case rc-cast}
\text{\hspace{1em} with reduction(8) show thesis by fastforce}
\text{qed}
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 (FieldProj (New cPair [New cPair [New cPair [New A [], New B []], New A []]]) ffst) fsnd) fsnd →∗ x}

**end**

**theory** Featherweight-Java

**imports** FJSound Execute

**begin**

**end**

**References**
