A Theory of Featherweight Java in Isabelle/HOL

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

Abstract

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

Contents

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

theory FJDefs
imports Main
begin

1.1 Syntax

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

1.1.1 Type definitions

type-synonym varName = nat
type-synonym methodName = nat
type-synonym className = nat
record varDef =
  vdName :: varName
  vdType :: className
type-synonym varCtx = varName → className

1.1.2 Constants

definition
  Object :: className where
  Object = 0
definition
  this :: varName where
  this == 0

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

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

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

1.1.6 Classes
record classDef =
  cName :: className
  cSuper :: className
  cFields :: varDef list
  cConstructor :: constructorDef
  cMethods :: methodDef list
1.1.7 Class Tables

type-synonym classTable = className → 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
\[
\text{isubexprs} :: (exp * exp) \Rightarrow \text{bool} \quad (- \in \text{isubexprs}'(-) [80,80] 80)
\]
where
\[
e' \in \text{isubexprs}(e) \equiv (e',e) \in \text{isubexprs}
\]
- \text{se-field} : e \in \text{isubexprs}(FieldProj\ e\ f)
- \text{se-invkrecv} : e \in \text{isubexprs}(MethodInvk\ e\ m\ es)
- \text{se-invkarg} : [\[ ei \in \text{set}\ es \]] \Rightarrow ei \in \text{isubexprs}(MethodInvk\ e\ m\ es)
- \text{se-cast} : e \in \text{isubexprs}(\text{Cast}\ e)

abbreviation
\[
\text{subexprs} :: \text{[exp,exp]} \Rightarrow \text{bool} \quad (- \in \text{subexprs}'(-) [80,80] 80)
\]
where
\[
e' \in \text{subexprs}(e) \equiv (e',e) \in \text{subexprs}'
\]

1.3 Values

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

inductive
\[
\text{vals} :: [\text{exp list}] \Rightarrow \text{bool} \quad (vals'(\cdot) [80] 80)
\]
and \( \text{val} :: [\text{exp}] \Rightarrow \text{bool} \quad (val'(\cdot) [80] 80) \)
where
\[
\text{vals-nil} : \text{vals}([],[])
\]
\[
| \text{vals-cons} : \text{[ val(\overline{vh}); val(\overline{vt}) ]} \Rightarrow \text{vals}((\overline{vh} \# \overline{vt}))
\]
\[
| \text{val} : \text{[ val(\overline{vs}) ]} \Rightarrow \text{val}(\text{New C}\ \overline{vs})
\]

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 \((ds/\overline{xs})e\) and \([ds/\overline{xs}]es\) respectively.

primrec \( \text{substs} :: (\overline{varName} \rightarrow \text{exp}) \Rightarrow \text{exp} \Rightarrow \text{exp} \)
and \( \text{subst-list1} :: (\overline{varName} \rightarrow \text{exp}) \Rightarrow \text{exp list} \Rightarrow \text{exp list} \)
and \( \text{subst-list2} :: (\overline{varName} \rightarrow \text{exp}) \Rightarrow \text{exp list} \Rightarrow \text{exp list} \) where
\[
\text{substs} \sigma (\overline{Var}\ x) = \text{case } (\sigma(x)) \text{ of None } \Rightarrow (\overline{Var}\ x) \mid \text{Some } p \Rightarrow p
\]
\[
| \text{substs} \sigma (\text{FieldProj} e f) = \text{FieldProj} (\text{substs} \sigma\ e) f
\]
\[
| \text{substs} \sigma (\text{MethodInvk} e\ m\ es) = \text{MethodInvk} (\text{substs} \sigma\ e) m (\text{subst-list1} \sigma\ es)
\]
\[
| \text{substs} \sigma (\text{New\ C}\ es) = \text{New C} (\text{subst-list2} \sigma\ es)
\]
\[ \text{substs } \sigma (\text{Cast } C e) = \] \text{Cast } C (\text{substs } \sigma e) \\
\[ \text{subst-list1 } \sigma [] = [] \] \\
\[ \text{subst-list1 } \sigma (h \# t) = (\text{substs } \sigma h) \# (\text{subst-list1 } \sigma t) \] \\
\[ \text{subst-list2 } \sigma [] = [] \] \\
\[ \text{subst-list2 } \sigma (h \# t) = (\text{substs } \sigma h) \# (\text{subst-list2 } \sigma t) \] 

\text{abbreviation} \\
\text{substs-syn} :: [\exp \list] \Rightarrow [\varName \list] \Rightarrow [\exp] \Rightarrow [\exp] \\
\text{abbreviation} \\
\text{subst-list-syn} :: [\exp \list] \Rightarrow [\varName \list] \Rightarrow [\exp \list] \Rightarrow [\exp \list] 

1.5 Lookup

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

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

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

1.6 Variable Definition Accessors

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

\texttt{definition} \\
\texttt{varDefs-names} :: \varDef \list \Rightarrow \varName \list \texttt{where} \\
\texttt{varDefs-names} = \texttt{map vdName}

\texttt{definition} \\
\texttt{varDefs-types} :: \varDef \list \Rightarrow \className \list \texttt{where} \\
\texttt{varDefs-types} = \texttt{map vdType}

1.7 Subtyping Relation

The subtyping relation, written \( CT \vdash C <: D \) is just the reflexive and transitive closure of the immediate subclass relation. (For the sake of simplicity,
we define subtyping directly instead of using the reflexive and transitive closure operator.) The subtyping relation is extended to lists of classes, written $CT \vdash +Cs <: Ds$.

**inductive**

```
subtyping :: [classTable, className, className] ⇒ bool
```

where

```
s-refl : CT ⊢ C <: C
| s-trans : [ CT ⊢ C <: D; CT ⊢ D <: E ] ⇒ CT ⊢ C <: E
| s-super : [ CT(C) = Some(CDef); cSuper CDef = D ] ⇒ CT ⊢ C <: D
```

**abbreviation**

```
neg-subtyping :: [classTable, className, className] ⇒ bool
```

where

```
CT ⊢ S ¬<: T ≡ ¬ CT ⊢ S <: T
```

**inductive**

```
subtypings :: [classTable, className list, className list] ⇒ bool
```

where

```
ss-nil : CT ⊢ + [] <: []
| ss-cons : [ CT ⊢ C0 <: D0; CT ⊢ + Cs <: Ds ] ⇒ CT ⊢ + (C0 # Cs) <: (D0 # Ds)
```

### 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**

```
fields :: [classTable, className, varDef list] ⇒ bool
```

where

```
f-obj:

fields(CT, Object) = []
```

```
f-class:

[ CT(C) = Some(CDef); cSuper CDef = D; cFields CDef = Cf; fields(CT,D) = Dg; DgCf = Dg @ Cf ]
⇒ fields(CT, C) = DgCf
```

### 1.9 mtype Relation

The **mtype** relation, written $\text{mtype}(CT, m, C) = Cs \rightarrow C_0$ 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] ⇒ bool
(mtype′(\(-,-,-\)) = \(-\to [80,80,80,80] 80\))
where
  mt-class:
  [ CT(C) = Some(CDef);
    lookup (cMethods CDef) (\(\lambda md.(mName md = m)\)) = Some(mDef);
    varDefs-types (mParams mDef) = Bs;
    mReturn mDef = B ]
  ⇒ mtype(CT,m,C) = Bs → B

  mt-super:
  [ CT(C) = Some(CDef);
    lookup (cMethods CDef) (\(\lambda md.(mName md = m)\)) = None;
    cSuper CDef = D;
    mtype(CT,m,D) = Bs → B ]
  ⇒ mtype(CT,m,C) = Bs → B

1.10 mbody Relation

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

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

  mb-super:
  [ CT(C) = Some(CDef);
    lookup (cMethods CDef) (\(\lambda md.(mName md = m)\)) = None;
    cSuper CDef = D;
    mbody(CT,m,D) = xs . e ]
  ⇒ mbody(CT,m,C) = xs . e

1.11 Typing Relation

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

inductive
typings :: [classTable, varCtx, exp list, className list] ⇒ bool (\:- ⊢ - : - [80, 80, 80, 80] 80)

and typing :: [classTable, varCtx, exp, className] ⇒ bool (\:- ⊢ - : - [80, 80, 80, 80] 80)

where

ts-nil : CT;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 (\lambda fd.(vdName fd = fi)) = Some(fDef); 
  vtType 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 \neq D ] ⇒ CT;Γ ⊢ Cast C e0 : C

t-scast : [ CT;Γ ⊢ e0 : D; ]
\[ CT \vdash C \neg\llcorner: D;\]
\[ CT \vdash D \neg\llcorner: C \]
\[ \implies CT; \Gamma \vdash \text{Cast } C \ e_0 : C \]

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

**Lemma** typing-induct:

- **Assumes:** \( CT; \Gamma \vdash e : C \) (is \( ?T \))
- **And:** \( \bigwedge C \ CT \ \Gamma \ \chi. \ \Gamma \ x = \text{Some } C \implies P \ CT \ \Gamma (\text{Var } x) \ C \)
- **And:** \( \bigwedge C \ 0 \ CT \ C \ f \ \Gamma \ e_0 \ f \ \text{Def } f. \ [CT; \Gamma \vdash \ e_0 : C_0; \ P \ CT \ \Gamma \ e_0 \ C_0; \ \text{fields}(CT, C_0) = C_f; \ \text{lookup } C_f (\lambda \text{fd. } \text{vdName } \text{fd } = f) = \text{Some } \text{fDef}; \ \text{vdType } \text{fDef } = C_i] \implies P \ CT \ \Gamma (\text{FieldProj } e_0 \ f) \ C_i \)
- **And:** \( \bigwedge C \ C_0 \ CT \ C_0 \ Ds \ \Gamma \ e_0 \ es \ m. \ [CT; \Gamma \vdash \ e_0 : C_0; \ P \ CT \ \Gamma \ e_0 \ C_0; \ \text{mtype}(CT, m, C_0) = Ds \to C; \ CT; \Gamma \vdash+ \ es : Cs; \ \bigwedge i. \ [i < \text{length es }] \implies P \ CT \ \Gamma (\text{es}!i) (Cs!i); \ CT \vdash+ Cs <: Ds; \ \text{length es} = \text{length Ds} \implies P \ CT \ \Gamma (\text{MethodInvk } e_0 \ m \ es) \ C \)
- **And:** \( \bigwedge C \ C_0 \ CT \ C_0 \ Df \ Ds \ \Gamma \ es. \ [\text{fields}(CT, C) = Df; \ \text{length es} = \text{length Df}; \ \text{varDefs-types } Df = Ds; \ CT; \Gamma \vdash+ \ es : Cs; \ \bigwedge i. \ [i < \text{length es }] \implies P \ CT \ \Gamma (\text{es}!i) (Cs!i); \ CT \vdash+ Cs <: Ds] \implies P \ CT \ \Gamma (\text{New } C \ es) \ C \)
- **And:** \( \bigwedge C \ C_0 \ CT \ C_0 \ D \ \Gamma \ e_0. \ [CT; \Gamma \vdash \ e_0 : D; \ P \ CT \ \Gamma \ e_0 \ D; \ CT \vdash D <: C] \implies P \ CT \ \Gamma (\text{Cast } C \ e_0) \ C \)
- **And:** \( \bigwedge C \ C_0 \ CT \ C_0 \ D \ \Gamma \ e_0. \ [\exists (\forall i < \text{length es}. 
\text{P } CT \ \Gamma (\text{es}!i) (Cs!i)) \ \implies \ P \ CT \ \Gamma (\text{Cast } C \ e_0) \ C \)
- **And:** \( \bigwedge C \ C_0 \ CT \ C_0 \ D \ \Gamma \ e_0. \ [CT; \Gamma \vdash \ e_0 : D; \ P \ CT \ \Gamma \ e_0 \ D; \ CT \vdash C <: D; \ C \neq D] \implies P \ CT \ \Gamma (\text{Cast } C \ e_0) \ C \)

**Shows:** \( P \ CT \ \Gamma \ e \ C \) (is \( ?P \))

**Proof**

- **Fix:** \( es \ Cs \)
- **Let:** \( \text{IH} = \text{CT}; \Gamma \vdash+ \ es : Cs \implies (\forall i < \text{length es}. \ P \ CT \ \Gamma (\text{es}!i) (Cs!i)) \)
- **Have:** \( \text{IH} \land (\text{f} \implies \ ?P) \)

**Proof (induct rule: typings-typing.induct)**

- **Case (ts-nil CT \ \Gamma):** show \( ?\text{case by auto} \)

**Next**

- **Case (ts-cons CT \ \Gamma \ e_0 \ C_0 \ es \ Cs):**
  - **Show:** \( ?\text{case proof} \)
    - **Fix:** \( i \)
      - **Show:** \( i < \text{length (e_0#es)} \implies P \ CT \ \Gamma ((e_0\#\text{es}!i) ((C_0\#\text{Cs}!i)) \text{ using ts-cons by cases } i, \ \text{auto}) \)
    - **Qed**

**Next**

- **Case t-var then show:** \( ?\text{case using assms by auto} \)

**Next**

- **Case t-field then show:** \( ?\text{case using assms by auto} \)

**Next**

- **Case t-invk then show:** \( ?\text{case using assms by auto} \)

**Next**

- **Case t-new then show:** \( ?\text{case using assms by auto} \)

**Next**
1.12 Method Typing Relation

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

\[
\text{inductive method-typing :: [classTable, methodDef, className] ⇒ bool} (\cdot \vdash \cdot \ OK \ IN \ ·\ [80,80,80] 80)
\]

\[
\text{where m-typing:}
\]

\[
\begin{align*}
\lbrack & CT(C) = \text{Some}(C\text{Def}); \\
& c\text{Name} C\text{Def} = C; \\
& c\text{Super} C\text{Def} = D; \\
& m\text{Name} m\text{Def} = m; \\
& \text{lookup}(c\text{Methods} C\text{Def}) (\lambda md.(m\text{Name} md = m)) = \text{Some}(m\text{Def}); \\
& m\text{Return} m\text{Def} = C0; m\text{Params} m\text{Def} = Cx\times; m\text{Body} m\text{Def} = e0; \\
& \text{varDefs-types} Cx\times = C\times; \\
& \text{varDefs-names} Cx\times = x\times; \\
& \Gamma = (\text{map-upds Map.empty} x\times C\times)(\text{this} \mapsto C); \\
& CT;\Gamma \vdash e0 : E0; \\
& CT \vdash E0 <: C0; \\
& \forall C\times D0. (m\text{type}(CT,m,D) = D\times \rightarrow D0) \rightarrow (C\times=D\times \land C0=D0) \] \\
\Longrightarrow CT \vdash m\text{Def} OK \ IN \ C
\end{align*}
\]

\[
\text{inductive method-typings :: [classTable, methodDef list, className] ⇒ bool} (\cdot \vdash+ \cdot \ OK \ IN \ ·\ [80,80,80] 80)
\]

\[
\text{where}
\]

\[
\begin{align*}
\lbrack & \text{ms-nil :} \\
& CT \vdash+ [] OK \ IN \ C
\end{align*}
\]

\[
\lbrack & \text{ms-cons :} \\
& \lbrack CT \vdash m \ OK \ IN \ C; \\
& \quad CT \vdash+ \text{ms} \ OK \ IN \ C \] \\
\Longrightarrow CT \vdash+ (m \neq \text{ms}) \ OK \ IN \ C
\end{align*}
\]
1.13 Class Typing Relation

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

inductive
\[
\text{class-typing} :: [\text{classTable}, \text{classDef}] \Rightarrow \text{bool} (- \vdash - \text{OK} [80,80] 80)
\]
where
t-class:
\[
\begin{align*}
\text{cName CDef} &= C; \\
\text{cSuper CDef} &= D; \\
\text{cConstructor CDef} &= KDef; \\
\text{cMethods CDef} &= M; \\
\text{kName KDef} &= C; \\
\text{kParams KDef} &= (Dg@Cf); \\
\text{kSuper KDef} &= \text{varDefs-names Dg}; \\
\text{kInits KDef} &= \text{varDefs-names Cf}; \\
\text{fields}(CT,D) &= Dg; \\
CT \vdash M \text{OK IN C}
\end{align*}
\]  
\[ \Rightarrow CT \vdash CDef \text{OK} \]

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 \( C \).

inductive
class-typing :: classTable \Rightarrow bool (- \text{OK} 80)
where
class-all-ok:
\[
\begin{align*}
\text{Object} & \notin \text{dom}(CT); \\
\forall C CDef. \ CT(C) = \text{Some}(CDef) & \rightarrow (CT \vdash CDef \text{OK}) \land (\text{cName CDef} = C)
\end{align*}
\]
\[ \Rightarrow CT \text{OK} \]

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.

inductive
\[
\text{reduction} :: [\text{classTable}, \text{exp}, \text{exp}] \Rightarrow \text{bool} (- \vdash - \rightarrow - [80,80,80] 80)
\]
where
\[
\begin{align*}
\text{r-field}:
\begin{align*}
\text{fields}(CT,C) &= Cf; \\
\text{lookup2 Cf es} (\lambda fd.(vdName fd = fi)) &= \text{Some}(ei)
\end{align*}
\]  
\[ \Rightarrow CT \vdash \text{FieldProj (New C es) fi} \rightarrow ei \]
| \begin{align*} \text{r-invk}:
\begin{align*}
\text{mbody}(CT,m,C) &= xs . e0;
\end{align*}
\end{align*}
\[ \text{subs} \ ((\text{map-upds \ Map.empty \ xs \ ds}) (\text{this} \mapsto (\text{New \ C} \ \text{es}))) \ e0 = e0' \]
\[ \implies CT \vdash \text{MethodInvk} (\text{New \ C} \ \text{es}) \ m \ ds \to e0' \]

| r-cast:  
\[ [ CT \vdash C <: D ] \]
\[ \implies CT \vdash \text{Cast} \ D (\text{New \ C} \ \text{es}) \to \text{New \ C} \ \text{es} \]

| r-field:  
\[ [ CT \vdash e0 \to e0' ] \]
\[ \implies CT \vdash \text{FieldProj} \ e0 \ f \to \text{FieldProj} \ e0' \ f \]

| r-invk-rece:  
\[ [ CT \vdash e0 \to e0' ] \]
\[ \implies CT \vdash \text{MethodInvk} \ e0 \ m \ es \to \text{MethodInvk} \ e0' \ m \ es \]

| r-invk-arg:  
\[ [ CT \vdash e1 \to e1' ] \]
\[ \implies CT \vdash \text{MethodInvk} \ e0 \ m \ (el@ei#er) \to \text{MethodInvk} \ e0 \ m \ (el@ei'#er) \]

| r-new-arg:  
\[ [ CT \vdash e1 \to e1' ] \]
\[ \implies CT \vdash \text{New} \ C (el@ei#er) \to \text{New} \ C (el@ei'#er) \]

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

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

end

2 FJAux: Auxiliary Lemmas

theory FJAux imports FJDefs begin

2.1 Non-FJ Lemmas

2.1.1 Lists

lemma mem-ith:  
\[ \text{assumes } ei \in \text{set} \ \text{es} \]
\[ \text{shows } \exists \ el. \ er. \ \text{es} = el@ei#er \]
\[ \text{using assms} \]
\[ \text{proof} (\text{induct} \ \text{es}) \]
case Nil thus \(\text{?case by auto}\)
next
case (Cons es h 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) \@ (ei\#er) by auto hence ?case by blast \}
ultimately show ?case by blast
qed

lemma ith-mem: \(\forall i. \left[ i < \text{length es} \right] \implies es!i \in \text{set es}\)
p
proof (induct es)
case Nil thus \(\text{?case by auto}\)
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 \([xs[\mapsto]ys,x\mapsto y] = [(xs[\mapsto]x)\mapsto(y[\mapsto]y)]]\)
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 \([xs[\mapsto]As]\mapsto x = \text{Some Ai}\)
shows \(\exists i. (As!i = Ai) \wedge (i < \text{length As}) \wedge (\forall (Bs::'c list).((\text{length Bs} = \text{length As}) \implies ((xs[\mapsto]Bs) x = \text{Some (Bs !i)})))\)
(is \(\exists i. \ ?P\ i\ x\ As\)
is \(\exists i. (\ ?P\ i\ As) \wedge (\ ?P\2\ i\ As) \wedge (\forall (Bs::\ 'c list).((\ ?P\3\ i\ xs\ As\ Bs)))\)
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 \(\exists i. \ ?P\ i\ [[\mapsto]]\ by\ contradiction\)
next
fix xa xs y ys
assume length-xs-ys: length xs = length ys
and IH: \([xs[\mapsto]ys]\ x = \text{Some Ai} \implies \exists i. \ ?P\ i\ xs\ ys\)
and map-eq- Some: \([xa \# xs[\mapsto]y \# ys]\ x = \text{Some Ai}\)
then have map-decomp: \([xa\#xs[\mapsto]y\#ys] = [xa\rightarrow y]++[xs[\mapsto]ys]\) by fastforce
show \(\exists i. \ ?P\ i\ (xa\#xs) (y \# ys)\)
proof (cases \([x \mapsto y] \cdot x\))

\begin{verbatim}
  case (Some Ai')
  hence \([^a \mapsto y] ++ [x \mapsto y] = Some Ai'\) by (rule map-add-find-right)
  hence P: \([x \mapsto y] \cdot x = Some Ai\) using map-eq-Some Some by simp

from IH \([OF \ P]\) obtain \(i\) where
  \(R1\) \(\vdots \) \(i = Ai\)
  and \(R2\): \(i < \text{length } y\)
  and \(pre-r3\): \(\forall (Bs::'c list). \ ?P3 i \ x \ y \ Bs\) by fastforce

\{ fix Bs::'c list
  assume length-Bs: \(\text{length } B = \text{length } (y \# y)\)
  then obtain \(n\) where \(\text{length } (y \# y) = Suc \ n\) by auto
  with length-Bs obtain \(b \ b\) where Bs-def: \(B = b \# b\) by (auto simp add: length-Suc-conv)
    with length-Bs have \(\text{length } y = \text{length } b\) by simp
    with pre-r3 have \([^a \mapsto b] ++ [x \mapsto b] = Some (b!i)\) by (auto simp only: map-add-find-right)
      with pre-r3 Bs-def length-Bs have \(?P3 (i+1) (xa\#xs) (y\#ys) Bs\) by simp }
  with \(R1\) \(R2\) have \(?P (i+1) (xa\#xs) (y\#ys)\) by auto
  thus \(?thesis\) ..
next
  case None
    with map-decomp map-eq-Some have \([^a \mapsto y] = Some Ai\) by (auto simp only: map-add-find-right)
  hence ai-def: \(y = Ai\) and x-eq-xa: \(x = xa\) by (auto simp only: map-upd-Some-unfold)
\end{verbatim}

\{ fix Bs::'c list
  assume length-Bs: \(\text{length } B = \text{length } (y \# y)\)
  then obtain \(n\) where \(\text{length } (y \# y) = Suc \ n\) by auto
  with length-Bs obtain \(b \ b\) where Bs-def: \(B = b \# b\) by (auto simp add: length-Suc-conv)
    with length-Bs have \(\text{length } y = \text{length } b\) by simp
    hence dom([x \mapsto y]) = dom([x \mapsto b]) by auto
    with None have \([x \mapsto y] = None\) by (auto simp only: domIff)
    moreover from x-eq-xa x-eq-xa have sing-map: \([^a \mapsto b] = Some b\) by (auto simp only: map-upd-Some-unfold)
    ultimately have \([^a \mapsto b] ++ [x \mapsto b] = Some b\) 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

2.2 FJ Lemmas

2.2.1 Substitution

lemma subst-list1-eq-map-substs :
\(\forall \sigma. \ \text{subst-list1 } \sigma \ l = \text{map (subs t } \sigma\) \(\ l\)
  by (induct \(l\), simp-all)

lemma subst-list2-eq-map-substs :
\(\forall \sigma. \ \text{subst-list2 } \sigma \ l = \text{map (subs t } \sigma\) \(\ l\)
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 by (cases f h) (auto simp add: lookup.simps)
qed

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

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

\[ i < \text{length } t1 \]

and $e = t1 ! i$

and

\[ \text{ih} : (\text{length } t1 = \text{length } (tl l2) \implies \text{lookup2 } t1 (tl l2) f = \text{Some } ((tl l2) ! i)) \]

using $\text{Cons}$ by blast

hence $Suc i < \text{length } (h1 \# t1) \land e = (h1 \# t1) ! (Suc i)$ using $\text{Cons}$ by auto

moreover 

\[ \begin{align*}
&\text{assume } \text{length } (h1 \# t1) = \text{length } l2 \\
&\text{hence } \text{lens} : \text{length } l2 = \text{Suc } (\text{length } t1) \text{ by auto} \\
&\text{then obtain } h2 t2 \text{ where } l2 \text{-def} : l2 = h2 \# t2 \text{ by } (\text{auto simp add; length-Suc-conv)} \\
&\text{hence } \text{lookup2 } t1 t2 f = \text{Some } (t2 ! i) \text{ using } \text{ih} \text{ l2-def } \text{lens} \text{ by auto} \\
&\text{hence } \text{lookup2 } (h1 \# t1) l2 f = \text{Some } ((l2 ! (Suc i))) \\
&\text{using } \text{asm } l2 \text{-def } \text{by} (\text{auto simp add; lookup2-simps})
\end{align*} \]

ultimately have \text{?case by auto}

\}

ultimately show \text{?case by auto}

qed

lemma lookup2-index:

\[ \forall l2. [ \begin{align*}
&\text{lookup2 } l1 l2 f = \text{Some } e; \\
&\text{length } l1 = \text{length } l2 \implies \exists i < (\text{length } l2). e = (l2 ! i) \land \text{lookup } l1 f = \text{Some } ((l1 ! i))
\end{align*} ] \]

proof (induct $l1$)

case Nil thus \text{?case by auto}

next

case (Cons $h1 t1$)

hence $\text{length } l2 = \text{Suc } (\text{length } t1)$ by auto

then obtain $h2 t2$ where $l2 \text{-def} : l2 = h2 \# t2$ by (auto simp add; length-Suc-conv)

\[ \begin{align*}
&\text{assume } \text{asm} : f \ h1 \\
&\text{hence } e = h2 \text{ using } \text{Cons } l2 \text{-def } \text{by } (\text{auto simp add; lookup2-simps}) \\
&\text{hence } 0 < \text{length } (h2 \# t2) \land e = (h2 \# t2) ! 0 \land \text{lookup } (h1 \# t1) f = \text{Some } ((h1 \# t1) ! 0) \\
&\text{using } \text{asm } \text{by } (\text{auto simp add; lookup-simps}) \\
&\text{hence } \text{?case using } l2 \text{-def } \text{by } \text{auto}
\end{align*} \]

moreover 

\[ \begin{align*}
&\text{assume } \text{asm} : \neg (f \ h1) \\
&\text{hence } \exists i < \text{length } t2. e = t2 ! i \land \text{lookup } t1 f = \text{Some } (t1 ! i) \text{ using } \text{Cons } l2 \text{-def}
\end{align*} \]

by auto

then obtain $i$ where $i < \text{length } t2 \land e = t2 ! i \land \text{lookup } t1 f = \text{Some } (t1 ! i)$ by auto

hence $(Suc i) < \text{length } (h2 \# t2) \land e = ((h2 \# t2) ! (Suc i)) \land \text{lookup } (h1 \# t1) f = \text{Some } ((h1 \# t1) ! (Suc i))$

using $\text{asm}$ by (force simp add; lookup-simps)

hence \text{?case using } l2 \text{-def } \text{by } \text{auto}

\}

ultimately show \text{?case by auto}

qed
lemma lookup-append:
  assumes lookup \( l f = \text{Some } r \)
  shows lookup (l@l') \( f = \text{Some } r \)
  using assms by (induct l) auto

lemma method-typings-lookup:
  assumes lookup-eq-Some: lookup \( M f = \text{Some } m \)
  and M-ok: \( \text{CT} \vdash M \text{OK IN } C \)
  shows \( \text{CT} \vdash m \text{Def OK IN } C \)
  using lookup-eq-Some M-ok
proof (induct M)
  case Nil thus ?case by fastforce
next
  case (Cons h t) thus ?case by (cases f h, auto elim: method-typings.cases simp add: lookup.simps)
qed

2.2.3 Functional

These lemmas prove that several relations are actually functions

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

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

lemma fields-functional:
  assumes fields: \( \text{fields}(\text{CT}, C) = Cf \)
  and CT OK
  shows \( \forall \text{Cf'} . \left[ \right. \text{fields}(\text{CT}, C) = \text{Cf} \left. \right] \implies \text{Cf} = \text{Cf'} \)
  using assms
proof
  case (f-obj CT)
  hence \( \text{CT}(\text{Object}) = \text{None} \) 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 Cf DgCf DgCf')
  hence f-class-inv:
    \( (\text{CT} C = \text{Some } C\text{Def}) \land (\text{cSuper } C\text{Def} = D) \land (\text{cFields } C\text{Def} = \text{Cf}) \)
    and \( \text{CT OK} \) by fastforce
  hence c-not-obj: \( C \neq \text{Object} \) by (force elim: ct-typing.cases)
  from f-class have fds: \( \text{fields}(\text{CT}, C) = DgCf' \) by fastforce

17
then obtain \( Dg' \) where
\[ \text{fields}(CT,D) = Dg' \]
and \( DgCf' = Dg' \otimes Cf \)
using \( f\text{-class-inv} \) \( \text{c-not-obj} \) by (auto elim:fields.cases)
hence \( Dg' = Dg \) using \( f\text{-class} \) by auto
thus \( \text{cases} \) using \( \langle DgCf = Dg \otimes Cf \rangle \) and \( \langle DgCf' = Dg' \otimes Cf \rangle \) by force
qed

2.2.4 Subtyping and Typing

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

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

**Lemma** \( \text{subtypings-index} \):
assumes \( CT \vdash Cs <: Ds \)
 showcases \( \forall i. [ i < \text{length} \ Cs ] \Longrightarrow CT \vdash (Cs!i) <: (Ds!i) \)
using assms
proof induct
case \( \text{ss-nil} \) thus \( ?\text{case} \) by auto
next
case \( \text{ss-cons} \ hCs \ CT \ tCs \ hDs \ tDs \ i \)
thus \( ?\text{case} \) by (cases \( i \), auto)
qed

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

**Lemma** \( \text{typings-append} \):
assumes \( CT;\Gamma \vdash es : Cs \)
and CT;Γ ⊢ e : C
shows CT;Γ ⊢+ (es@[@e]) : (Cs@[@C])
proof
  have length es = length Cs using assms by (simp-all add: typings-lengths)
  thus CT;Γ ⊢+ (es@[@e]) : (Cs@[@C]) using assms
proof (induct es Cs rule: list-Induct2)
  have CT;Γ ⊢+ [[]] by (simp add: typings-typing-ts-nil)
  moreover from assms have CT;Γ ⊢ e : C by simp
ultimately show CT;Γ ⊢+ ([@e]) : ([@C]) by (auto simp add: typings-typing-ts-cons)
next
  fix x xs y ys
  assume length xs = length ys
  and IH: [CT;Γ ⊢+ xs : ys; CT;Γ ⊢ e : C] =⇒ CT;Γ ⊢+ (xs @ [e]) : (ys @ [C])
  and x-xs-typs: CT;Γ ⊢+ (x ≠ xs) : (y ≠ ys)
  and e-typ: CT;Γ ⊢ e : C
  from x-xs-typs have x-typ: CT;Γ ⊢ x : y and CT;Γ ⊢+ xs : ys by (auto elim: typings.cases)
  with IH e-typ have CT;Γ ⊢+ (xs@[@e]) : (ys@[@C]) by simp
  with x-typ have CT;Γ ⊢+ ((x#xs)@[@e]) : ((y#ys)@[@C]) by (auto simp add: typings-typing-ts-cons)
  thus CT;Γ ⊢+ ((x ≠ xs) @ [e]) : ((y ≠ ys) @ [C]) by (auto simp add: typings-typing-ts-cons)
qed

lemma ith-typing: \( \forall Cs. \ [CT;Γ ⊢+ (es@[@(h#t)]) : Cs] =⇒ CT;Γ ⊢ h : (Cs!(length es)) \)
proof (induct es, auto elim: typings.cases)
qed

lemma ith-subtyping: \( \forall Ds. \ [CT ⊢+ (Cs@[@(h#t)]) <: Ds] =⇒ CT ⊢ h <: (Ds!(length Cs)) \)
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: \( \forall Ds Es. \ [CT ⊢+ Cs <: Ds; CT ⊢+ Ds <: Es] =⇒ CT ⊢+ Cs <: Es \)
proof (induct Cs)
case Nil thus ?case
  by (auto elim: subtypings.cases simp add: subtypings.ss-nil)
next
case (Cons hCs tCs)
than obtain hDs tDs
where h1: CT ⊢ hCs <: hDs and t1: CT ⊢+ tCs <: tDs and Ds = hDs#tDs
by (auto elim: subtypings.cases)
then obtain hEs tEs
where h2: CT ⊢ hDs <: hEs and t2: CT ⊢+ tDs <: tEs and Es = hEs#tEs
using Cons by (auto elim:subtypings.cases)
moreover from subtyping.s-trans[OF h1 h2] have CT ⊢ hCs <: hEs by fastforce
moreover with t1 t2 have CT ⊢+ tCs <: tEs using Cons by simp-all
ultimately show ?case by (auto simp add:subtypings.intros)
qed

lemma ith-typing-sub:
\[ \forall Cs. \exists CS, \exists CT, \exists \Gamma. \] CT;\Gamma ⊢+ (es@((h # t))): Cs;
CT;\Gamma ⊢ h': Ci';
\[ \Rightarrow \exists CS', (CT;\Gamma ⊢+ (es@((h' # t))): Cs' ∧ CT ⊢+ Cs' <: Cs) \]
proof (induct es)
case Nil
then obtain hCs tCs
  where ts: CT;\Gamma ⊢+ t : tCs
  and Cs-def: Cs = hCs # tCs by (auto elim:typings.cases)
from Cs-def Nil have CT ⊢+ Ci' <: hCs by auto
with Cs-def have CT ⊢+ (Ci' # tCs) <: Cs by (auto simp add:subtypings.ss-cons subtypings-refl)
moreover from ts Nil have CT;\Gamma ⊢+ (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 ts: CT;\Gamma ⊢ eh : hCs
  and Cs-def: Cs = hCs # tCs
  and Cs-def: Cs = hCs # tCs
  by (auto elim:typings.cases)
moreover with Cons obtain tCs'
  where CT;\Gamma ⊢+ (et@((h' # t))): tCs'
  and CT ⊢+ tCs' <: tCs
  by auto
ultimately have
CT;\Gamma ⊢+ (eh#(et@((h'#t)))): (hCs#tCs')
and CT ⊢+ (hCs#tCs') <: Cs
by (auto simp add:typings-typing.ts-cons subtypings.ss-cons subtyping.s-refl)
thus ?case by auto
qed

lemma mem-typings:
\[ \forall Cs. \exists CT, \exists \Gamma, \exists es:Cs; es ∈ set es \] \Rightarrow \exists Ci. CT;\Gamma ⊢ 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 \( \text{length } ds = \text{length } As \)
and \( \text{length } ds = \text{length } Bs \)
and \( i < \text{length } ds \)
shows \( CT; \Gamma \vdash ds!i : As!i \text{ and } CT \vdash As!i <: Bs!i \)
using assms by (auto simp add: typings-index subtypings-index)

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

lemma not-subtypes-aux:
assumes \( CT \vdash C <: Da \)
and \( C \neq Da \)
and \( CT C = \text{Some } C\text{Def} \)
and \( \text{cSuper } C\text{Def} = D \)
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 \( \forall D. [ CT \vdash D \neg<: C; \ CT \vdash C \neg<: D ] \implies CT \vdash A \neg<: D \)
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 \( CT \vdash Da \neg<: D \)
hence da-sub-d: \( CT \vdash Da <: D \) by auto
  have d-nsub-e: \( CT \vdash D <: E \) using s-trans by fastforce
  thus False using s-trans by (force simp add: subtyping.s-trans[OF da-sub-d
d-nsub-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 \( CT C \text{Def} D Da \))
have \( C \neq Da \) proof (rule ccontr)
  assume \( C \neq Da \)
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 \( e_1 \in \text{isubexprs}(e_0) \)

shows \( \forall C. [ C; \text{Map.empty} \vdash e_0 : C ] \implies \exists D. C; \text{Map.empty} \vdash e_1 : D \)

using assms

by (induct rule:isubexprs.induct) (auto elim:typing.cases simp add:mem-typings)

**Lemma subexpr-typing:**

assumes \( e_1 \in \text{subexprs}(e_0) \)

shows \( \forall C. [ CT; \text{Map.empty} \vdash e_0 : C ] \implies \exists D. CT; \text{Map.empty} \vdash e_1 : D \)

using assms

by (induct rule:isubexprs.induct) (auto elim:typing.cases simp add:mem-typings)

**Lemma isubexpr-reduct:**

assumes \( d_1 \in \text{isubexprs}(e_1) \)

shows \( \forall d_2. [ CT \vdash d_1 \rightarrow d_2 ] \implies \exists e_2. CT \vdash e_1 \rightarrow e_2 \)

using assms mem-ith

by (induct rule:rtrancl.induct) (auto force simp add:isubexpr-reduct)

**Lemma subexpr-reduct:**

assumes \( d_1 \in \text{subexprs}(e_1) \)

shows \( \forall d_2. [ CT \vdash d_1 \rightarrow d_2 ] \implies \exists e_2. CT \vdash e_1 \rightarrow e_2 \)

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 \( Cs :: \text{nat list} \)

assumes \( \text{mtype}(CT,m,C) = Cs \rightarrow C0 \)

shows \( \exists xs. e. \text{mbody}(CT,m,C) = xs \cdot e \wedge \text{length } xs = \text{length } Cs \)

using assms

proof (induct rule:mtype.induct)

- case (mt-class C0 Cs C CDef CT m mDef)

22
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)
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 M: CT \vdash+ M OK IN C and cMethods: 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 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: CT C = Some CDef'
    and cSuper CDef' = D'

and mName mDef = m'
and mReturn: mReturn mDef = C0'
and varDefs-types: varDefs-types (mParams mDef) = Cs'
and \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'
and D = D'
and m = m'
and \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 :: D
shows \forall 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-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 = \Gamma1 ++ \Gamma2
and \Gamma2 = [xs \mapsto Bs]
and length xs = length ds
and length Bs = length ds
and \exists As. CT;\Gamma1 \vdash ds : As \land CT \vdash As :: Bs
shows CT;\Gamma \vdash es:Ds \iff \exists Cs. (CT;\Gamma \vdash ([ds/xs]es):Cs \land CT \vdash Cs :: Bs)
Ds) is  ?TYPINGS  =⇒  ?P1
and  CT;Γ ⊢ e:D  =⇒  ∃C. (CT;Γ1 ⊢ ((ds/xs)e):C ∧ CT ⊢ C <: D) (is  ?TYPING  =⇒  ?P2)

proof
let  ?COMMON-ASMS = (CT OK) ∧ (Γ = Γ1 ++ Γ2) ∧ (Γ2 = [xs [→] Bs]) ∧ (length Bs = length ds) ∧ (∃As. CT;Γ1 ⊢+ ds : As ∧ CT ⊢+ As <: Bs)
have RESULT: (?TYPINGS =⇒ ?COMMON-ASMS =⇒ ?P1)
∧ (?TYPING =⇒ ?COMMON-ASMS =⇒ ?P2)
proof(rule: typings-typing.induct)
case (ts-nil CT Γ)
show ?case
proof (rule impI)
assume asms: (CT OK) ∧ (Γ = Γ1 ++ Γ2) ∧ (Γ2 = [xs [→] Bs]) ∧ (length Bs = length ds) ∧ (∃As. CT;Γ1 ⊢+ ds : As ∧ CT ⊢+ As <: Bs)
with ts-cons have e0-typ: CT;Γ ⊢ e0 : C0 by fastforce

with ts-cons asms have
  ∃C.(CT;Γ1 ⊢ (ds/xs) e0 : C) ∧ (CT ⊢ C <: C0)
and ∃Cs.(CT;Γ1 ⊢+ [ds/xs]es : Cs) ∧ (CT ⊢+ Cs <: Cs') by auto
then obtain C Cs where
  (CT;Γ1 ⊢ (ds/xs) e0 : C) ∧ (CT ⊢ C <: C0)
and (CT;Γ1 ⊢+ [ds/xs]es : Cs) ∧ (CT ⊢+ Cs <: Cs') by auto
hence CT;Γ1 ⊢+ [ds/xs][e0#es] : (C#Cs)
and CT ⊢+ (C#Cs) <: (C0#Cs') by (auto simp add: typings-typing.intros subtypings.intros)
then show ∃Cs. CT;Γ1 ⊢+ map (subs ts [→] ds) (e0 # es) : Cs ∧ CT ⊢+ Cs <: (C0 # Cs') by auto
qed

next
case (t-var Γ x C' CT)
show ?case
proof (rule impI)
assume asms: (CT OK) ∧ (Γ = Γ1 ++ Γ2) ∧ (Γ2 = [xs [→] Bs]) ∧ (length Bs = length ds) ∧ (∃As. CT;Γ1 ⊢+ ds : As ∧ CT ⊢+ As <: Bs)
hence
  lengths: length ds = length Bs
and G-def: Γ = Γ1 ++ Γ2
and G2-def : Γ2 = [xs[→]Bs] by auto
from lengths G2-def have same-doms: dom([xs[→]ds]) = dom(Γ2) by auto
from asms show ∃C. CT;Γ1 ⊢ subs [xs [→] ds] (Var x) : C ∧ CT ⊢ C <:
proof (cases Γ 2 x)
case None
with G-def t-var have G1-x: Γ 1 x = Some C' by (simp add: map-add-Some-iff)
from None same-doms have x ∈ dom([xs[↦→]ds]) by (auto simp only:domIff)

hence [xs[↦→]ds]x = None by (auto simp only:map-add-Some-iff)
hence (ds/xs)(Var x) = (Var x) by auto
with G1-x have
  CT;Γ 1 ⊢ (ds/xs)(Var x) : C' and CT ⊢ C' <: C'
  by (auto simp add: typings-typingintros subtyping-intros)
thus ?thesis by auto
next
case (Some Bi)
with G-def t-var have c'-eq-bi: C' = Bi by (auto simp add: map-add-SomeD)
from length xs = length ds) asms have length xs = length Bs by simp
with Some G2-def have ∃i.(Bs!i = Bi) ∧ (i < length Bs) ∧
  (∀ l.(length l = length Bs) → ([xs[↦→]l] x = Some ([ll]i))))
  by (auto simp add: map-upds-index)
then obtain i where bs-i-proj: (Bs!i = Bi)
and i-len: i < length Bs
and P: (∀ l:(length l = length Bs) → ([xs[↦→]l] x = Some ([ll]i))))
  by fastforce
from lengths P have subst-x: ([xs[↦→]ds]x = Some (ds!i)) by auto
from asms obtain As where as-ex:CT;Γ 1 ⊢+ ds : As ∧ CT ⊢+ As <: Bs
  by fastforce
hence length As = length Bs by (auto simp add: subtypings-lengths)
hence proj-i:CT;Γ 1 ⊢+ ds!i : As!i ∧ CT ⊢+ As!i <: Bs!i
  using i-len lengths as-ex by (auto simp add: typings-proj)
hence CT;Γ 1 ⊢+ (ds/xs)(Var x) : As!i ∧ CT ⊢+ As!i <: C'
  using c'-eq-bi bs-i-proj subst-x by auto
thus ?thesis ..
qed
qed
next
case(t-field CT Γ c!0 C0 Cf fi fDef Ci)
show ?case
proof (rule impl)
  assume asms: (CT OK) ∧ (Γ = Γ 1 ++ Γ 2) ∧
  (Γ 2 = [xs[↦→]Bs]) ∧ (length Bs = length ds) ∧ (∃ As. CT;Γ 1 ⊢+ ds : As ∧ CT ⊢+ As <: Bs)
  from t-field have flds: fields(CT,C0) = Cf by fastforce
  from t-field asms obtain C where e0-typ: CT;Γ 1 ⊢+ (ds/xs)e!0 : 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) (λ fd. vdName fd = fi) =

Some fDef 

by (simp add: lookup-append)

from e0-typ flds-C lookup-CfDg t-field have CT;Γ1 ⊢ (ds/xs)(FieldProj e0 fi)

: Ci

by (simp add: typings-typing.intros)

moreover have CT ⊢ Ci <: Ci by (simp add: subtyping.intros)

ultimately show ∃ C. CT;Γ1 ⊢ (ds/xs)(FieldProj e0 fi) : C ∧ CT ⊢ C <: Ci

by auto

qed

next

case(t-inv k CT e0 C0 m Ds C es Cs)

show ?case

proof (rule impI)

assume asms: (CT OK) ∧ (Γ = Γ1 ++ Γ2) ∧ (Γ2 = [xs ↦→ Bs]) ∧ (length Bs = length ds) ∧ (∃ As. CT;Γ1 ⊢ + ds : As ∧ CT ⊢ + As <: Bs)

hence ct-ok: CT OK ..

from t-inv k have mtyp: mtype(CT,m,C0) = Ds → C

and subs: CT ⊢+ Cs <: Ds

and lens: length es = length Ds

by auto

from t-inv k asms obtain C’ where

e0-typ: CT;Γ1 ⊢ (ds/xs)e0 : C’ and sub’: CT ⊢ C’ <: C0 by auto

from t-inv k asms obtain Cs’ where

es-typ: CT;Γ1 ⊢+ [ds/xs]es : Cs’ and subs’: CT ⊢+ Cs’ <: Cs by auto

have subst-e: (ds/xs)(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 subs’ subs]

lens

subst-e

have CT;Γ1 ⊢ (ds/xs)(MethodInvk e0 m es) : C by (auto simp add: typings-typing.intros)

moreover have CT ⊢ C <: C by (simp add: subtyping.intros)

ultimately show ∃ C’. CT;Γ1 ⊢ (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) ∧ (Γ = Γ1 ++ Γ2) ∧ (Γ2 = [xs ↦→ Bs]) ∧ (length Bs = length ds) ∧ (∃ As. CT;Γ1 ⊢+ ds : As ∧ CT ⊢+ As <: Bs)

hence ct-ok: CT OK ..

from t-new have

subs: CT ⊢+ Cs <: Ds
and flds: fields(CT,C) = Df
and len: length es = length Df
and vcts: varDefs-types Df = Ds
by auto
from t-new asms obtain Cs' where
  es-typ: CT;ΓI ⊢ (ds/Es) : Cs' and subs': CT ⊢ Cs' <: Cs by auto
have subst-e: (ds/Es)(New C es) = New C ([ds/Es]es)
  by(auto simp add: subst-list2-eq-map-substs)
from es-typ subtypings-trans[OF subs' subs] flds subst-e len vcts
have CT;ΓI ⊢ (ds/Es)(New C es) : C by(auto simp add: typings-typing.intros)
moreover have CT ⊢ C <: C by(simp add: subtyping.intros)
ultimately show ∃ C'. CT;ΓI ⊢ (ds/Es)(New C es) : C ∧ CT ⊢ C' <: C by
auto
qed
next
case(t-ucast CT Γ e0 D C)
  show ?case
  proof(rule impl)
    assume asms: (CT OK) ∧ (Γ = Γ 1 ++ Γ 2) ∧ (Γ 2 = [xs →] Bs) ∧ (length
    Bs = length ds) ∧ (∃ As. CT;ΓI ⊢ + ds : As ∧ CT ⊢ + As <: Bs)
    from t-ucast asms obtain C' where e0-typ: CT;ΓI ⊢ (ds/Es)e0 : C'
      and sub1:CT ⊢ C' <: D
      and sub2:CT ⊢ D <: C by auto
    from sub1 sub2 have CT ⊢ C' <: C by (rule s-trans)
    with e0-typ have CT;ΓI ⊢ (ds/Es)(Cast C e0) : C by(auto simp add: typings-typing.intros)
    moreover have CT ⊢ C <: C by (rule s-refl)
    ultimately show ∃ C'. CT;ΓI ⊢ (ds/Es)(Cast C e0) : C' ∧ CT ⊢ C' <: C by
    auto
    qed
next
case(t-dcast CT Γ e0 D C)
  show ?case
  proof(rule impl)
    assume asms: (CT OK) ∧ (Γ = Γ 1 ++ Γ 2) ∧ (Γ 2 = [xs →] Bs) ∧ (length
    Bs = length ds) ∧ (∃ As. CT;ΓI ⊢ + ds : As ∧ CT ⊢ + As <: Bs)
    from t-dcast asms obtain C' where e0-typ: CT;ΓI ⊢ (ds/Es)e0 : C'
      by auto
      (C' ≠ C ∧ CT ⊢ C <: C') ∨
      (CT ⊢ C =<: C' ∧ CT ⊢ C' =<: C) by blast
    moreover
      { assume CT ⊢ C' <: C
        with e0-typ have CT;ΓI ⊢ (ds/Es) (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/Es) (Cast C e0) : C by (auto simp add: typings-typing.intros)
      }
  qed
next
case(t-new CT Γ e0 D C)
  show ?case
  proof(rule impl)
    assume asms: (CT OK) ∧ (Γ = Γ 1 ++ Γ 2) ∧ (Γ 2 = [xs →] Bs) ∧ (length
    Bs = length ds) ∧ (∃ As. CT;ΓI ⊢ + ds : As ∧ CT ⊢ + As <: Bs)
    from t-new asms obtain Cs' where e0-typ: CT;ΓI ⊢ (ds/Es)e0 : C'
      and sub1:CT ⊢ C' <: D
      and sub2:CT ⊢ D <: C by auto
    from sub1 sub2 have CT ⊢ C' <: C by (rule s-trans)
    with e0-typ have CT;ΓI ⊢ (ds/Es)(Cast C e0) : C by(auto simp add: typings-typing.intros)
    moreover have CT ⊢ C <: C by (rule s-refl)
    ultimately show ∃ C'. CT;ΓI ⊢ (ds/Es)(Cast C e0) : C' ∧ CT ⊢ C' <: C by
    auto
    qed
next
case(t-ucast CT Γ e0 D C)
  show ?case
  proof(rule impl)
    assume asms: (CT OK) ∧ (Γ = Γ 1 ++ Γ 2) ∧ (Γ 2 = [xs →] Bs) ∧ (length
    Bs = length ds) ∧ (∃ As. CT;ΓI ⊢ + ds : As ∧ CT ⊢ + As <: Bs)
    from t-ucast asms obtain C' where e0-typ: CT;ΓI ⊢ (ds/Es)e0 : C'
      and sub1:CT ⊢ C' <: D
      and sub2:CT ⊢ D <: C by auto
    from sub1 sub2 have CT ⊢ C' <: C by (rule s-trans)
    with e0-typ have CT;ΓI ⊢ (ds/Es)(Cast C e0) : C by(auto simp add: typings-typing.intros)
    moreover have CT ⊢ C <: C by (rule s-refl)
    ultimately show ∃ C'. CT;ΓI ⊢ (ds/Es)(Cast C e0) : C' ∧ CT ⊢ C' <: C by
    auto
    qed
next
case(t-dcast CT Γ e0 D C)
  show ?case
  proof(rule impl)
    assume asms: (CT OK) ∧ (Γ = Γ 1 ++ Γ 2) ∧ (Γ 2 = [xs →] Bs) ∧ (length
    Bs = length ds) ∧ (∃ As. CT;ΓI ⊢ + ds : As ∧ CT ⊢ + As <: Bs)
    from t-dcast asms obtain C' where e0-typ: CT;ΓI ⊢ (ds/Es)e0 : C'
      by auto
      (C' ≠ C ∧ CT ⊢ C <: C') ∨
      (CT ⊢ C =<: C' ∧ CT ⊢ C' =<: C) by blast
    moreover
      { assume CT ⊢ C' <: C
        with e0-typ have CT;ΓI ⊢ (ds/Es) (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/Es) (Cast C e0) : C by (auto simp add: typings-typing.intros)
      }
  qed
next
moreover
\{ assume (CT ⊢ C <: C′ ∧ CT ⊢ C′ <: C) 
with e0-typ have CT;Γ I ⊢ (ds/xs) (Cast C e0) : C by (auto simp add: typings-typing,intros) \}
ultimately have CT;Γ I ⊢ (ds/xs) (Cast C e0) : C by auto
moreover have CT ⊢ C <: C by (rule s-refl)
ultimately show ∃ C′. CT;Γ I ⊢ (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 = Γ I ++ Γ 2) ∧ (Γ 2 = [xs ↦→ Bs]) ∧ (length Bs = length ds) ∧ (∃ As. CT;Γ I ⊢+ ds : As ∧ CT ⊢+ As <: Bs)
from t-scast asms obtain C′ where e0-typ: CT;Γ I ⊢ (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;Γ I ⊢ (ds/xs) (Cast C e0) : C using e0-typ by (auto simp add: typings-typing,intros)
thus ∃ C′. CT;Γ I ⊢ (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;Γ 2 ⊢+ es : Cs) ⇒ (CT;Γ I++Γ 2 ⊢+ es : Cs) (is ?P1 ⇒ ?P2)
and CT;Γ 2 ⊢ e : C ⇒ CT;Γ I++Γ 2 ⊢ e : C (is ?Q1 ⇒ ?Q2)
proof –
have (?P1 ⇒ ?P2) ∧ (?Q1 ⇒ ?Q2)
by (induct rule: typings-typing.induct, auto simp add: map-add-find-right typings-typing.intros)

thus ?P1 ⇒ ?P2 and ?Q1 ⇒ ?Q2 by auto
3.5 Method Body Typing Lemma

lemma A-1-4:
assumes ct-ok: CT OK
and mb:mbody(CT,m,C) = xs . e
and mt:mtype(CT,m,C) = Ds → D
shows ∃ D0 C0, (CT ⊢ C <: D0) ∧ (CT:|xs|→Ds|this|→D0) ⊢ e : C0)
using mb ct-ok mt proof (induct rule: mbody.induct)
case (mb-class CT C CDef m mDef xs e)
hence
  m-param:varDefs-types (mParams mDef) = Ds
and m-ret:mReturn mDef = D
and CT ⊢ CDef OK
and cName CDef = C
by (auto elim: mtype.cases ct-typing.cases)
hence CT ⊢ (cMethods CDef) OK IN C by (auto elim: class-typing.cases)
hence CT ⊢ mDef OK IN C using mb-class by (auto simp add: method-typings-lookup)
hence ∃ E0. ((CT:|xs|→Ds|this|→C| ⊢ e : E0) ∧ (CT ⊢ E0 <: D))
  using mb-class m-param m-ret by (auto elim: method-typing.cases)
then obtain E0
  where CT:|xs|→Ds|this|→C| ⊢ e : E0
and CT ⊢ E0 <: D
and CT ⊢ 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 → D]
  ⇒ ∃ D0 C0. (CT ⊢ Da <: D0) ∧ (CT ⊢ C0 <: D)
  ∧ (CT:|xs|→Ds|this|→D0| ⊢ e:C0) by fastforce+
from mb-super have c-sub-da: CT ⊢ C <: Da by (auto simp add: s-super)
from mb-super have mt:mtype(CT,m,Da) = Ds → D by (auto elim: mtype.cases)
from IH[OF ct mt] obtain D0 C0
  where s1: CT ⊢ Da <: D0
and CT ⊢ C0 <: D
and CT:|xs|→Ds|this|→D0| ⊢ 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 ⊢ e → e'
and CT OK
shows [C; [CT;Γ ⊢ e : C]]
  ⇒ ∃ C'. (CT;Γ ⊢ e' : C' ∧ CT ⊢ C' <: C)
using assms

proof (induct rule: reduction.induct)
  case (r-field CT Ca Cf es fi e')
  hence CT;Γ ⊢ FieldProj (New Ca es) fi : C
      and ct-ok: CT OK
      and flds': fields(CT, Ca') = Cf
      and lookup2: lookup2 Cf es (∑fd. vdName fd = fi) = Some e' by fastforce+
  then obtain Ca'' Cf'' fDef
    where new-typ: CT;Γ ⊢ New Ca es : Ca''
    and es-typs: CT;Γ ⊢ es: Cs
    and Ds-def: varDefs-types Cf'' = Ds
    and length-Cf-es: length Cf'' = length es
    and subs: CT ⊢ Cs <: Ds
    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;Γ ⊢ es: Cs
    and Ds-def: varDefs-types Cf'' = Ds
    and length-Cf-es: length Cf'' = length es
    and subs: CT ⊢ 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 lookup2] obtain i where
    i-bound: i < length es
    and e' = es!i
    and lookup Cf (∑fd. vdName fd = fi) = Some (Cf''!i) by auto
  moreover
  with C-def Ds-def lookup lookup2 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;Γ ⊢ (es!i): (Cs!i) and CT ⊢ (Cs!i) <: (Ds!i) using i-bound
  by (auto simp add:typing-index subtypings-index typing_lengths)
  ultimately show ?case by auto
next
  case (r-invkt CT m Ca xs e ds es e')
  from r-invkt have mb: mbbody(CT,m,Ca) = xs . e by fastforce
  from r-invkt obtain Ca' Ds Cs
    where CT;Γ ⊢ New Ca es : Ca'
    and mtype(CT,m,Ca') = Cs → C
    and ds-typs: CT;Γ ⊢ ds : Ds
    and Ds-subs: CT ⊢ Ds <: Cs
    and l': length ds = length Cs by (auto elim:typing.cases)
  hence new-typ: CT;Γ ⊢ New Ca es : Ca
  and mt: mtype(CT,m,Ca) = Cs → C by (auto elim:typing.cases)
  from ds-typs new-typ have CT;Γ ⊢ (ds @[[New Ca es]]) : (Ds @[[Ca]])
    by (simp add:typing-append)
  moreover from A-1-4[OF - mb mt] r-invkt obtain Da E
where $CT \vdash Ca <: Da$
and $E\text{-sub-}C$: $CT \vdash E <: C$
and $e\text{-typ}1$: $CT;[xs[\rightarrow]Cs, this \rightarrow Da] \vdash e : E$ by auto
moreover with $Ds\text{-subs}$ have $CT \vdash+ (Ds@[(Ca)]) <: (Cs@[Da])$
  by(auto simp add:subtyping-append)
ultimately have ex: $\exists As. CT;\Gamma \vdash+ (ds @[New Ca es]) : As \land CT \vdash+ As <: (Cs@[Da])$
  by auto
from $e\text{-typ}1$ have $e\text{-typ}2$: $CT;[\Gamma;[xs[\rightarrow]Cs, this \rightarrow Da]] \vdash e : E$
  by(simp only:A-1-3)
from $e\text{-typ}2$ mtype-mbody-length[OF mt mb]
have $e\text{-typ}3$: $CT;[\Gamma;[\{xs[\rightarrow]Cs, this \rightarrow Da\}]] \vdash e : E$
  by(force simp only:map-shuffle)
let $\forall 1 = \Gamma$ and $\forall 2 = [(xs[\rightarrow]Cs, this)[Da]]$
have g-def: $(\forall 1 \vdash+ \forall 2) = (\forall 1 \vdash+ \forall 2)$ and g2-def: $\forall 2 = \forall 2$ by auto
from A-1-2[OF g-def g2-def - ex] $e\text{-typ}3$ r-invk 1 mtype-mbody-length[OF mt mb]
obtain $E'$ where $e'\text{-typ} : CT;\Gamma \vdash \text{substs} [(xs[\rightarrow]Cs, this)[ds@[New Ca es]]] : E'$
  and $E'\text{-sub-}E$: $CT \vdash E' <: E$ by force
moreover from $e'\text{-typ} 1$ mtype-mbody-length[OF mt mb]
have $CT;\Gamma \vdash \text{substs} [xs[\rightarrow]ds, this \rightarrow (New Ca es)] : E'$
  by(auto simp only:map-shuffle)
moreover from $E'\text{-sub-}E\text{-sub-}C$ have $CT \vdash E' <: C$ by (rule subtyping.s-trans)
ultimately show $\forall case using$ r-invk by auto
next
  case (r-cast $CT Ca Da es$)
  then obtain $Ca'$
    where $C = D$
  and $CT;\Gamma \vdash New Ca es : Ca'$ by (auto elim: typing_cases)
  thus $\forall case using$ r-cast by (auto elim: typing_cases)
next
  case (rc-field $CT e0 e0' f$)
  then obtain $C0 Cf fd$ where $CT;\Gamma \vdash e0 : C0$
    and $Cf\text{-def}: fields(CT,C0) = Cf$
    and $fd\text{-def}:\text{lookup} Cf (\lambda fd. (vdName fd = f)) = Some fd$
    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-fields[OF OF - $Cf\text{-def}$] obtain $Cf'$
  where $\text{fields}(CT,C') = (Cf\oplus Cf')$ by rule (rule CT \vdash C' <: C0)
moreover with $fd\text{-def}$ have $\text{lookup} (Cf\oplus Cf') (\lambda fd. (vdName fd = f)) = Some fd$
  by(simp add:lookup-append)
ultimately have $CT;\Gamma \vdash FieldProj e0' f : C$ by(auto simp add:typings-typing.t-field)
  thus $\forall case by$ (auto simp add:subtyping.s-refl)
next
  case (rc-invk-rev$CT e0 e0' m es C$)
  then obtain $C0 Ds Cs$ 32
where $ct-ok : CT \rightarrow OK$
and $CT;\Gamma \vdash e0 : C0$
and $mt : mtype(CT, m, C0) = Ds \rightarrow C$
and $CT;\Gamma \vdash es : C$s
and length $es = length Ds$
and $CT \vdash Cs <: Ds$
by (auto elim:typing.cases)

moreover with $rc-invk-recv$ obtain $C0'$
where $CT;\Gamma \vdash e0' : C0'$
and $CT \vdash C0' <: C0$ by auto

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

ultimately have $CT;\Gamma \vdash MethodInvek e0' m es : C$ by (auto simp add: typings-typing.t-invk)
thus \{case by (auto simp add:subtyping.s-refl)

next

\{case (rc-invk-arg $CT \equiv ei' e0 m el er C$)
then obtain $Cs Ds C0$
where $typts : CT;\Gamma \vdash (el@(ei'#er)) : Cs$
and $el-typ : CT;\Gamma \vdash e0 : C0$
and $mt : mtype(CT, m, C0) = Ds \rightarrow C$
and $Cs\text{-sub-Ds} : CT \vdash Cs <: Ds$
and length $el@(ei'#er) = length Ds$
by (auto elim:typing.cases)

hence $CT;\Gamma \vdash ei:(Cs!(length el))$ by (simp add:ith-typing)

\{with $rc-invk-arg$ obtain $Ci'$
where $ei-typ : CT;\Gamma \vdash ei':Ci'$
and $Ci-sub : CT \vdash Ci' <: (Cs!(length el))$
by auto

from $ith\text{-}typing\text{-}sub[\text{of } typts ei-typ Ci-sub]$ obtain $Cs'$
where $es'\text{-typs} : CT;\Gamma \vdash (el@(ei'#er)) : Cs'$
and $Cs'\text{-sub-Cs} : CT \vdash Cs' <: Cs$ by auto

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

with $es'\text{-typs subtypings-trans[\text{of } Cs'\text{-sub-Cs Cs-sub-Ds}] el-typ}$ have
$CT;\Gamma \vdash MethodInvek e0 m (el@(ei'#er)) : C$
by (auto simp add: typings-typing.t-invk)

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

next

\{case (rc-new-arg $CT \equiv ei' Ca el er C$)
then obtain $Cs Df Ds$
where $typts : CT;\Gamma \vdash (el(@(ei'#er))) : Cs$
and $flds : fields(CT, C) = Df$
and length $el@(ei'#er) = length Df$
and $Ds\text{-def} : varDefs$-types $Df = Ds$
and $Cs\text{-sub-Ds} : CT \vdash Cs <: Ds$
and $C\text{-def} : Ca = C$
by (auto elim:typing.cases)

hence $CT;\Gamma \vdash ei:(Cs!(length el))$ by (simp add:ith-typing)

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

3.7 Multi-Step Subject Reduction Theorem

corollary \( \text{Cor-2-4-1-multi} : \)
assumes \( \text{CT} \vdash \text{e} \leftrightarrow \text{e}' \)
and \( \text{CT OK} \)
shows \( \forall \text{C} . \left[ \text{CT} ; \Gamma \vdash \text{e} : \text{C} \right] \implies \exists \text{C}' . \left( \text{CT} ; \Gamma \vdash \text{e}' : \text{C}' \land \text{CT} \vdash \text{C}' < : \text{C} \right) \)
using \( \text{assms} \)
proof \( \text{induct} \)
case \( \text{rs-refl} \text{ CT e C} \) thus \(?\text{case}\) by (auto simp add: subtyping.s-refl)
next
case \( \text{rs-trans} \text{ CT e e' e'' C} \)
hence \( \text{e-typ}: \text{CT};\Gamma \vdash \text{e} : \text{C} \)
and \( \text{e-step}: \text{CT} \vdash \text{e} \rightarrow \text{e'} \)
and \( \text{ct-ok}: \text{CT OK} \)
and \( \text{IH}: \forall \text{D}. \left[ \text{CT} ; \Gamma \vdash \text{e'} : \text{D} ; \text{CT OK} \right] \implies \exists \text{E}. \text{CT};\Gamma \vdash \text{e''} : \text{E} \land \text{CT} \vdash \text{E} < :
D
by auto
from Thm-2-4-1[OF e-step ct-ok e-typ] obtain D where e'-typ: CT;Γ ⊢ e' : D
and D-sub-C: CT ⊢ D <: C by auto
with IH[OF e'-typ ct-ok] obtain E where CT;Γ ⊢ e'": E and E-sub-D: CT ⊢ E
<: D by auto
moreover from s-trans[OF E-sub-D D-sub-C] have CT ⊢ E <: C by auto
ultimately show ?case by auto
qed

3.8 Progress

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

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

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

proof (induct CT Γ e C rule: typing-induct)
  case 1 thus ?case using assms by auto
next
case (2 C CT Γ x) thus ?case by auto
next
case (3 C0 Ct Cf Ci Γ e0 C0 es fDef fi)
  have s1: e0 ∈ subexprs(FieldProj e0 fi) by (auto simp add: subexprs.intros)
  from 3 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 (4 C C0 CT Cs Ds Γ e0 es m)
  have s1: e0 ∈ subexprs(MethodInvk e0 m es) by (auto simp add: subexprs.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 \( C \) D es where \( s2 : \) Cast D (New C0 es) \( \in \) subexprs(e0) by auto
} moreover {
  assume \( ?V \) e0
  then obtain \( C \)0 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 \( \Gamma \) es)
hence
length es = length Cs
\( \forall i. \; \lfloor i \rfloor < \text{length es} ; \; CT; \Gamma \vdash (\text{es}!i) : (\text{Cs}!i); \; \Gamma = \text{Map.empty} \Rightarrow ?IH (\text{es}!i) \)
and \( CT; \Gamma \vdash+ \text{es} : \text{Cs} \)
by (auto simp add: typings-lengths)
hence \( \exists i < \text{length es} . \; (?F (\text{es}!i) \lor ?M (\text{es}!i) \lor ?C (\text{es}!i)) \lor (\text{vals}(\text{es})) \) (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: \( CT; \Gamma \vdash+ (h\#t) : (Ch\#Ct) \)
  and \( \text{OIH}: \; \forall i. \; \lfloor i \rfloor < \text{length } (h\#t) ; \; CT; \Gamma \vdash ((h\#t)!i) : ((Ch\#Ct)!i); \; \Gamma = \text{Map.empty} \Rightarrow ?IH ((h\#t)!i) \)
  and \( \text{G-def}: \; \Gamma = \text{Map.empty} \)
  by auto
from \( \text{h-t-typs have} \)
  \( h\)-typ: \( CT; \Gamma \vdash (h\#t)!0 : (Ch\#Ct)!0 \)
  and \( t\)-typs: \( CT; \Gamma \vdash+ t : Ct \)
  by (auto elim:typings.cases)
\{ fix \( i \) assume \( i < \text{length } t \)
  hence \( s\)-i: Suc \( i < \text{length } (h\#t) \) by auto
from \( \text{OIH}[OF s\text{-i}] \) have \( \lfloor i \rfloor < \text{length } t ; \; CT; \Gamma \vdash+ (t!i) : (Ct!i); \; \Gamma = \text{Map.empty} \)
\( \Rightarrow ?IH (t!i) \) by auto \}
with \( t\)-typs have \( ?Q \) \( t \) using Cons by auto
moreover {
  assume \( \exists i < \text{length } t . \; (\forall F (t!i) \lor \exists M (t!i) \lor \exists C (t!i)) \)
  then obtain \( i \)
  where \( i < \text{length } t \)
  and \( ?F (t!i) \lor \exists M (t!i) \lor \exists C (t!i) \) by force
  hence \( (\text{Suc } i < \text{length } (h\#t)) \land (\forall F ((h\#t)!((\text{Suc } i))) \lor \exists M ((h\#t)!((\text{Suc } i))) \lor \exists C ((h\#t)!((\text{Suc } i))) \lor \exists ?Q ((h\#t)!((\text{Suc } i))) \) by auto
hence \( \exists i < \text{length } (h\#t) . \; (\forall F ((h\#t)!i) \lor \exists M ((h\#t)!i) \lor \exists C ((h\#t)!i)) \) ..
  hence \( ?Q (h\#t) \) by auto
} moreover {
  assume \( v\)-t: vals\( t \)
from \(\text{OIH}[\text{OF - h-typ G-def}]\) have \(?IH\ h\) by auto

moreover
\{ assume \(?F\ h\lor \?M\ h\lor \?C\ h\) 
  hence \(?F\ ((h\#t)!0)\lor \?M\ ((h\#t)!0)\lor \?C\ ((h\#t)!0)\) by auto 
  hence \(?Q\ (h\#t)\) by force 
\} moreover \{ 
  assume \(?V\ h\) 
  with \(v-t\) have \(\text{vals}(h\#t)\) by (force intro:vals-valintros) 
  hence \(?Q(h\#t)\) by auto 
\} ultimately have \(?Q(h\#t)\) by blast

\} ultimately show \(?Q(h\#t)\) by blast

qed

moreover \{ 
  assume \(\exists i<\text{length}\ es. \ ?F\ (es!i)\lor \?M\ (es!i)\lor \?C(es!i)\) 
  then obtain \(i\) where \(i\text{-len: }i<\text{length}\ es\) and \(r:\ ?F\ (es!i)\lor \?M\ (es!i)\lor \?C(es!i)\) by force 
  from \text{ith-mem}[\text{OF i-len}] have \(s1:es!i\in\text{subexprs}(\text{New}\ C\ es)\) by(auto intro:subexprs.se-newary) 
  \{ assume \(?F\ (es!i)\)
  then obtain \(C0\ es'\ fi\) where \(s2: \text{FieldProj}\ (\text{New}\ C0\ es')\ fi\in\text{subexprs}(es!i)\) by auto 
  from \text{rtrancl-trans}[\text{OF s2 s1}] have \(?F(\text{New}\ C\ es)\lor \?M(\text{New}\ C\ es)\lor \?C(\text{New}\ C\ es)\) by auto 
  \} moreover \{ 
  assume \(?M\ (es!i)\)
  then obtain \(C0\ es'\ m'\ ds\) where \(s2: \text{MethodInvk}\ (\text{New}\ C0\ es')\ m'\ ds\in\text{subexprs}(es!i)\) by force 
  from \text{rtrancl-trans}[\text{OF s2 s1}] have \(?F(\text{New}\ C\ es)\lor \?M(\text{New}\ C\ es)\lor \?C(\text{New}\ C\ es)\) by auto 
  \} moreover \{ 
  assume \(?C\ (es!i)\)
  then obtain \(C0\ D\ es'\) where \(s2: \text{Cast}\ D\ (\text{New}\ C0\ es')\in\text{subexprs}(es!i)\) by auto 
  from \text{rtrancl-trans}[\text{OF s2 s1}] have \(?F(\text{New}\ C\ es)\lor \?M(\text{New}\ C\ es)\lor \?C(\text{New}\ C\ es)\) by auto 
  \} ultimately have \(?F(\text{New}\ C\ es)\lor \?M(\text{New}\ C\ es)\lor \?C(\text{New}\ C\ es)\) using \(r\) by blast 
  hence \(?case\ by\ auto\)
  \} moreover \{ 
  assume \(\text{vals}(es)\)
  hence \(?case\ by(\text{auto intro:vals-valintros})\)
  \} ultimately show \(?case\ by\ blast\)

next
  case \((6\ C\ CT\ D\ \Gamma\ e0)\) 
  have \(s1: e0\in\text{subexprs}(\text{Cast}\ C\ e0)\) by(auto simp add:isubexprs.intros) 
  from \(6\) have \(?IH\ e0\) by auto 
  moreover 
  \{ assume \(?F\ e0\)
  then obtain \(C0\ es\ fi\) where \(s2: \text{FieldProj}\ (\text{New}\ C0\ es)\ fi\in\text{subexprs}(e0)\) by
auto
  from rtrancl-trans[OF s2 s1] have ?case by auto
} moreover {
  assume ?M e0
  then obtain C0 es m ds where s2: MethodInvk (New C0 es) m ds ∈ subexprs(e0) by auto
  from rtrancl-trans[OF s2 s1] have ?case by auto
} moreover {
  assume ?C e0
  then obtain C0 D′ es where s2: Cast D′ (New C0 es) ∈ subexprs(e0) by auto
  from rtrancl-trans[OF s2 s1] have ?case by auto
} moreover {
  assume ?V e0
  then obtain C0 es′ where e0 = (New C0 es′) and vals(es′) by (force elim:val.cases)
  hence ?case by(force intro:isubexprs.intros)
} ultimately show ?case by blast
next
  case (7 C CT D Γ e0)
  have s1: e0 ∈ subexprs(Cast C e0) by(auto simp add:isubexprs.intros)
  from 7 have ?IH e0 by auto
  moreover {
    assume ?F e0
    then obtain C0 es fi where s2: FieldProj (New C0 es) fi ∈ subexprs(e0) by auto
    from rtrancl-trans[OF s2 s1] have ?case by auto
  } moreover {
    assume ?M e0
    then obtain C0 es m ds where s2: MethodInvk (New C0 es) m ds ∈ subexprs(e0) by auto
    from rtrancl-trans[OF s2 s1] have ?case by auto
  } moreover {
    assume ?C e0
    then obtain C0 D′ es where s2: Cast D′ (New C0 es) ∈ subexprs(e0) by auto
    from rtrancl-trans[OF s2 s1] have ?case by auto
  } moreover {
    assume ?V e0
    then obtain C0 es′ where e0 = (New C0 es′) and vals(es′) by (force elim:val.cases)
    hence ?case by(force intro:isubexprs.intros)
  } ultimately show ?case by blast
next
  case (8 C CT D Γ e0)
  have s1: e0 ∈ subexprs(Cast C e0) by(auto simp add:isubexprs.intros)
  from 8 have ?IH e0 by auto
  moreover {
    assume ?F e0
    then obtain C0 es fi where s2: FieldProj (New C0 es) fi ∈ subexprs(e0) by auto
  }

39
from rtrancl-trans[OF s2 s1] have \( ? \text{case by auto} \)
} moreover {
  assume \( ?M e0 \)
then obtain \( C0 \) es m ds where \( s2 : \text{MethodInvk (New C0 es) m ds} \in \text{subexprs}(e0) \)
by auto
  from rtrancl-trans[OF s2 s1] have \( ? \text{case by auto} \)
} moreover {
  assume \( ?C e0 \)
then obtain \( C0 \) D' es where \( s2 : \text{Cast D' (New C0 es)} \in \text{subexprs}(e0) \) by auto
from rtrancl-trans[OF s2 s1] have \( ? \text{case by auto} \)
} moreover {
  assume \( ?V e0 \)
then obtain \( C0 \) es' where \( e0 = (\text{New C0 es'}) \) and \( \text{vals(es')} \) by (force elim:val.cases)
hence \( ? \text{case by (force intro:subexprs.intros)} \)
} ultimately show \( ? \text{case by blast} \)
qed

3.9 Type Soundness Theorem

theorem Thm-2-4-3:
assumes \( e\text{-typ} : \text{CT;Map.empty \vdash e : C} \)
and \( \text{ct-ok} : \text{CT OK} \)
and \( \text{multisteps} : \text{CT} \vdash e \rightarrow^{*} e1 \)
and \( \text{no-step} : \neg(\exists e2. \text{CT} \vdash e1 \rightarrow e2) \)
shows \( (\text{val}(e1) \land (\exists D. \text{CT};\text{Map.empty} \vdash e1 : D \land \text{CT} \vdash D <: C)) \)
\lor (\exists D C es. (\text{Cast D (New C0 es)} \in \text{subexprs}(e1) \land \text{CT} \vdash C \land D <: D))

proof –
from asms Cor-2-4-1-multi[OF multisteps ct-ok e-typ] obtain C1
  where \( e1\text{-typ} : \text{CT;Map.empty \vdash e1 : C1} \)
  and \( C1\text{-sub-C} : \text{CT} \vdash C1 <: C \) by auto
from \( e1\text{-typ} \) have \( (\exists \text{C0 es } fi. \text{FieldProj (New C0 es) f} \in \text{subexprs}(e1)) \)
\lor (\exists \text{C0 es m ds. (MethodInvk (New C0 es) m ds) \in subexprs(e1))}
\lor (\exists D C \text{ es. (Cast D (New C0 es)} \in \text{subexprs}(e1))
\lor \text{val}(e1)) \) \( \text{is } ?F e1 \lor ?M e1 \lor ?C e1 \lor ?V e1) \) by (simp add: closed-subterm-split)
moreover {
  assume \( ?F e1 \)
then obtain \( C0 \) es fi where \( fp : \text{FieldProj (New C0 es) f} \in \text{subexprs}(e1) \) by auto
  then obtain \( C1 \) where \( CT;\text{Map.empty} \vdash \text{FieldProj (New C0 es) f} : C1 \) using \( e1\text{-typ} \) by (force simp add:subexpr-typing)
  then obtain \( C0' \) where \( \text{new-typ} : \text{CT;Map.empty} \vdash \text{New C0 es} : C0' \) by (force elim:typing.cases)
hence \( C0 = C0' \) by (auto elim:typing.cases)
with \( \text{new-typ} \) obtain \( Df \) where \( f1 : \text{fields(CT,C0)} = Df \) and \( \text{lens: length es} = \text{length Df} \) by (auto elim:typing.cases)
from \( \text{Thm-2-4-2-1[OF e1-typ fp]} \) obtain \( Cf \) fDef
  where \( f2 : \text{fields(CT,C0)} = Cf \)
  and \( \text{lookup} \text{ CF } (\lambda fd. \text{valName} \text{ fd = fi} = \text{Some(fDef)} \) by force
moreover from \( \text{fields-functional[OF f1 ct-ok f2]} \) \text{lens} have \( \text{length es} = \text{length Cf} \)
by auto

moreover from lookup-index[OF lkup] obtain i where
  i<length Cf
  and fDef = Cf ! i
  and (length Cf = length es) --> lookup2 Cf es (λfd. vdName fd = fi) = Some (es!i) by auto

ultimately have lookup2 Cf es (λfd. vdName fd = fi) = Some (es!i) by auto

with f2 have CT ⊢ FieldProj(New C0 es) fi (es!i) by(auto intro: reduction.intros)

with fp have ∃ e2. CT ⊢ e1 → e2 by(simp add: subexpr-reduct)

with no-step have thesis by auto

} moreover {

assume ?M e1

then obtain C0 es m ds where mi:MethodInsk (New C0 es) m ds ∈ subexprs(e1)

by auto

then obtain D where CT; Map.empty ⊢ MethodInsk (New C0 es) m ds : D

using e1-typ by(force simp add: subexpr-typing)

then obtain C0′ Es E
  where m-typ: CT; Map.empty ⊢ New C0 es : C0′
  and m-type(CT, m, C0′) = Es → E
  and length ds = length Es

by (auto elim:typing.cases)

from Thm-2-4-2-2[OF e1-typ mi] obtain xs e0 where mb: mbody(CT, m, C0) = xs . e0 and length xs = length ds by auto

hence CT ⊢ (MethodInsk (New C0 es) m ds) → (subs[xs[→]ds, this→(New C0 es)]e0) by(auto simp add: reduction.intros)

with mi have ∃ e2. CT ⊢ e1 → e2 by(simp add: subexpr-reduct)

with no-step have thesis by auto

} moreover {

assume ?C e1

then obtain C0 D es where c-def: Cast D (New C0 es) ∈ subexprs(e1) by auto

then obtain D′ where CT; Map.empty ⊢ Cast D (New C0 es) : D′ using e1-typ

by (force simp add: subexpr-typing)

then obtain C0′ where new-typ: CT; Map.empty ⊢ New C0 es : C0′ and D-eq-D′:
D = D′ by (auto elim:typing.cases)

hence C0-eq-C0′: C0 = C0′ by(auto elim:typing.cases)

hence thesis proof(cases CT ⊢ C0 :<: D)
  case True
  hence CT ⊢ Cast D (New C0 es) → (New C0 es) by(auto simp add: reduction.intros)
  with c-def have ∃ e2. CT ⊢ e1 → e2 by (simp add: subexpr-reduct)
  with no-step show thesis by auto

next

  case False
  with c-def show thesis by auto

qed

} moreover {

assume ?V e1

hence thesis using assms by(auto simp add: Cor-2-4-1-multi)

} ultimately show thesis by blast

qed
theory Execute
imports FJSound
begin

4 Executing FeatherweightJava programs

We execute FeatherweightJava programs using the predicate compiler.

\textbf{code-pred} (modes: \(i \Rightarrow i \Rightarrow i \Rightarrow \text{bool}\),
\(i \Rightarrow i \Rightarrow o \Rightarrow \text{bool}\) as supertypes-of) subtyping.

\textbf{thm} subtyping.equation

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

\textbf{definition} append where append \(xs\ ys\ zs\ =\ (zs\ =\ xs\ \oplus\ ys)\)

\textbf{lemma} [code-pred-intro]: append \(\[]\ xs\ xs\)
unfolding append-def by simp

\textbf{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:

\textbf{lemma} rc-invk-arg': \(CT\vdash\ ei\rightarrow\ ei'\\implies\ append\ el\ (ei\ #\ er)\ e'\implies\ append\ el\ (ei')\ #\ er\ e''\implies\ CT\vdash\ MethodInvk\ e\ m\ e'\rightarrow\ MethodInvk\ e\ m\ e''\)
unfolding append-def by simp (rule reduction.intros(6))

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

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

\textbf{code-pred} (modes: \(i \Rightarrow i \Rightarrow i \Rightarrow \text{bool}\), \(i \Rightarrow i \Rightarrow o \Rightarrow \text{bool}\) as reduce)
\textbf{proof} –
\textbf{case} append
from this show thesis
unfolding append-def by (cases xa) fastforce+
next
case reduction
from reduction.prems show thesis
proof (cases rule: reduction.cases)
  case r-field
    with reduction(1) show thesis by fastforce
next
case r-invk
  with reduction(2) show thesis by fastforce
next
case r-cast
  with reduction(3) show thesis by fastforce
next
case re-field
  with reduction(4) show thesis by fastforce
next
case re-invk-rev
  with reduction(5) show thesis by fastforce
next
case re-invk-arg
  with reduction(6) show thesis
    unfolding append-def by fastforce
next
case rc-new-arg
  with reduction(7) show thesis
    unfolding append-def by fastforce
next
case rc-cast
  with reduction(8) show thesis by fastforce
qed

thm reduction.equation

code-pred reductions.

thm reductions.equation

We also make the class typing executable: this requires that we derive rules for method-typing.
definition method-typing-aux
where
  method-typing-aux CT m D Cs C = (¬ (∀ Ds D0. mtype(CT,m,D) = Ds → D0
        → Cs = Ds ∧ C = D0))

lemma method-typing-aux:
  (∀ Ds D0. mtype(CT,m,D) = Ds → D0 → Cs = Ds ∧ C = D0) = (¬
  method-typing-aux CT m D Cs C)
unfolding method-typing-aux-def by auto
lemma [code-pred-intro]:
\[ mtype(CT,m,D) = Ds \to D0 \implies Cs \neq Ds \implies \text{method-typing-aux } CT m D Cs \]

unfolding method-typing-aux-def by auto

lemma [code-pred-intro]:
\[ mtype(CT,m,D) = Ds \to D0 \implies C \neq D0 \implies \text{method-typing-aux } CT m D Cs \]

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 =
abbreviation ffst :: varName where
ffst == 4

abbreviation fsnd :: varName where
fsnd == 5

abbreviation setfst :: methodName where
setfst == 6

abbreviation newfst :: varName where
newfst == 7

definition classPair-Def :: classDef where
classPair-Def = (\) cName = cPair, cSuper = Object,
cFields = ([]) vdName = ffst, vdType = Object [], ([) vdName = fsnd, vdType = Object [],
cConstructor = ([) kName = cPair, kParams = ([) vdName = ffst, vdType = Object [], ([) vdName = fsnd, vdType = Object [], kSuper = [], kInits = [ffst, fsnd]]
, cMethods = ([) mReturn = cPair, mName = setfst, mParams = ([) vdName = newfst, vdType = Object []],
mBody = New cPair [Var newfst, FieldProj (Var this) fsnd] []]]
definition exampleProg :: classTable where
exampleProg = (((\x. None)(A := Some classA-Def))(B := Some classB-Def))(cPair := Some classPair-Def)

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

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

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

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

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