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**

\[
\text{isubexprs} :: (\exp \ast \exp) \set
\]

**and** \( \text{isubexprs'} :: [\exp, \exp] \Rightarrow \text{bool} \) \( (- \in \text{isubexprs'}(-) [80,80] 80) \)

**where**

\[
e' \in \text{isubexprs}(e) \equiv (e', e) \in \text{isubexprs}
\]

| \text{se-field} & : & e \in \text{isubexprs}(&\text{FieldProj } e fis)
| \text{se-invkrecv} & : & e \in \text{isubexprs}(MethodInvk m es)
| \text{se-invkarg} & : & [ [ ei \in \text{set es} ] ] \Rightarrow ei \in \text{isubexprs}(MethodInvk m es)
| \text{se-newarg} & : & [ [ ei \in \text{set es} ] ] \Rightarrow ei \in \text{isubexprs}(New \ C es)
| \text{se-cast} & : & e \in \text{isubexprs}(&\text{Cast } C e)

**Abbreviation**

\[
\text{subexprs} :: [\exp, \exp] \Rightarrow \text{bool} \) \( (- \in \text{subexprs'}(-) [80,80] 80) \)

**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{v}s), where \( \overline{v}s \) is a list of values.

**Inductive**

\[
\text{vals} :: [\exp \set \Rightarrow \text{bool} (\text{vals'}(-) [80] 80)
\]

**and** \( \text{val} :: [\exp] \Rightarrow \text{bool} (\text{val'}(-) [80] 80) \)

**where**

\[
\text{vals-nil} : \text{vals}([],)
\]

| \text{vals-cons} & : & \text{val}(\overline{v}h); \text{vals}(\overline{v}t) ] \Rightarrow \text{vals}((\overline{v}h \# \overline{v}t))
| \text{val} & : & [ [ \text{vals}(\overline{v}s) ] ] \Rightarrow \text{val}(\text{New } C \overline{v}s)

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

**Primrec**

\[
\text{substs} :: (\varName \Rightarrow \exp) \Rightarrow \exp \Rightarrow \exp
\]

**and** \( \text{subst-list1} :: (\varName \Rightarrow \exp) \Rightarrow \exp \text{list} \Rightarrow \exp \text{list} \)

**and** \( \text{subst-list2} :: (\varName \Rightarrow \exp) \Rightarrow \exp \text{list} \Rightarrow \exp \text{list} \)

**where**

\[
\text{substs } \sigma \ (\text{Var } x) = \begin{cases} \text{case } (\sigma(x)) \text{ of None } \Rightarrow (\text{Var } x) & \text{Some } p \Rightarrow p \end{cases}
\]

| \text{substs } \sigma \ (\text{FieldProj } e f) & = & \text{FieldProj } (\text{substs } \sigma e) f
| \text{substs } \sigma \ (\text{MethodInvk } m \overline{e} \overline{s}) & = & \text{MethodInvk } (\text{substs } \sigma e) m (\text{subst-list1 } \sigma \overline{e} \overline{s})
| \text{substs } \sigma \ (\text{New } C \overline{v}s) & = & \text{New } C (\text{subst-list2 } \sigma \overline{v}s)

4
1.5 Lookup

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

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

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

1.6 Variable Definition Accessors

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

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

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

1.7 Subtyping Relation

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

**inductive**

```
subtyping :: [classTable, className, className] ⇒ bool (- ⊢ - <: - [80,80,80] 80)
where
  s-refl : CT ⊢ 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 (- ⊢ ¬ <: - [80,80,80] 80)
where CTS ⊢ S ¬ <: T ≡ ¬ CTS ⊢ S <: T
```

**inductive**

```
subtypings :: [classTable, className list, className list] ⇒ bool (- ⊢ + - <: - [80,80,80] 80)
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 (fields('·',·) = - [80,80,80] 80)
where
  f-obj: fields(CT, Object) = []
  | f-class: [ CT(C) = Some(CDef); cSuper CDef = D; cFields CDef = Cf; fields(CT,D) = Dg; DgCf = Dg @ Cf ]
  ⇒ fields(CT,C) = DgCf
```

## 1.9 mtype Relation

The **mtype** relation, written $\text{mtype}(CT, m, C) = Cs \to C_0$ 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'(-,-,-) = - → [80,80,80,80] 80)
where
  mt-class:
  [ CT(C) = Some(CDef);
    lookup (cMethods CDef) (λmd.(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) (λmd.(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'(-,-,-))
= - . - [80,80,80,80] 80)
where
  mb-class:
  [ CT(C) = Some(CDef);
    lookup (cMethods CDef) (λmd.(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) (λmd.(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 (?· ?· ⊢ ?·)
 and typing :: [classTable, varCtx, exp, className] ⇒ bool (?· ?· ⊢ ?·)

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-invok :
  [ 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 ⊢ C < 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;}

8
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:**

**Assumptions:**
- \( CT \vdash C \rightarrow; D; \)
- \( CT \vdash D \rightarrow; C \)

**Conclusion:**
- \( CT; \Gamma \vdash \text{Cast} C e0 : C \)

**Proof**

1. **Fix** \( es Cs \)
2. **Let** \( \forall i \vdash \Gamma; es : Cs \rightarrow (\forall i < \text{length es}. \ P \ C T \; \Gamma \; (es!i) \; (Cs!i)) \)
3. **Have** \( \forall i \vdash (\forall i \rightarrow \; ?P) \)
4. **Proof (induct rule: typings-typing.induct)**
   - **Case** \((ts-nil CT \; \Gamma) \) **Show** \( ? \text{case by auto} \)
   - **Next**
     - **Case** \((ts-cons CT \; \Gamma \; e0 \; C0 \; es \; Cs) \)
       - **Show** \( ? \text{case proof} \)
         - **Fix** \( i \)
         - **Show** \( i < \text{length} \; (e0 \# es) \rightarrow P \; CT \; \Gamma \; (((e0 \# es)!i) \; ((C0 \# Cs)!i)) \; \text{using} \; ts-cons \)
       - **By** (cases \( i, \text{auto} \))
     - **Qed**
   - **Next**
     - **Case** \((t-var \; CT \; \Gamma) \) **Show** \( ? \text{case using} \; \text{assms by auto} \)
     - **Next**
       - **Case** \((t-field \; CT \; \Gamma) \) **Show** \( ? \text{case using} \; \text{assms by auto} \)
     - **Next**
       - **Case** \((t-invk \; CT \; \Gamma) \) **Show** \( ? \text{case using} \; \text{assms by auto} \)
     - **Next**
       - **Case** \((t-new \; CT \; \Gamma) \) **Show** \( ? \text{case using} \; \text{assms by auto} \)
case \( t\text{-ucast} \) then show \( \text{using assms by auto} \)
next

case \( t\text{-dcast} \) then show \( \text{using assms by auto} \)
next

case \( t\text{-scast} \) then show \( \text{using assms by auto} \)

qed

thus \( \text{thesis using assms by auto} \)

qed

1.12 Method Typing Relation

A method definition \( md \), declared in a class \( C \), is well-typed, written \( CT \vdash md \text{OK IN } C \) 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 \( \text{method-typing} :: [\text{classTable}, \text{methodDef}, \text{className}] \Rightarrow \text{bool} \) \( (- \vdash - \text{OK IN -} [80,80,80] 80) \)

where

\( m\text{-typing} : \)

\[ [ \begin{array}{l}
CT(C) = \text{Some}(C\text{Def});
cName C\text{Def} = C;
cSuper C\text{Def} = D;
mName m\text{Def} = m;
\lambda md.(mName md = m)) = \text{Some}(m\text{Def});
\text{mReturn mDef} = C\theta; \text{mParams mDef} = Cxs; \text{mBody mDef} = e\theta;
\text{varDefs-types Cxs} = Cs;
\text{varDefs-names Cxs} = xs;
\Gamma = (\text{map-upds empty xs Cs})((\text{this} \mapsto C);
CT;\Gamma \vdash e\theta : E\theta;
CT \vdash E\theta \triangleleft C\theta;
\forall Ds D\theta. (\text{mtype}(CT,m,D) = Ds \rightarrow D\theta) \implies (Cs=Ds \land C\theta=D\theta) ]
\implies CT \vdash m\text{Def} \text{OK IN } C \]

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

where

\( ms\text{-nil} : \)

\[ CT \vdash + \] OK IN C

| \( ms\text{-cons} : \)

\[ [ CT \vdash m \text{OK IN C};
CT \vdash + ms \text{OK IN C }] \]
\implies CT \vdash (m \neq ms) \text{OK IN C} \]
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.

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

where

\[
\text{t-class:} \quad [\begin{array}{l}
\text{cName CDef} = C; \\
\text{cSuper CDef} = D; \\
\text{cConstructor CDef} = KDef; \\
\text{cMethods CDef} = M; \\
\text{kName KDef} = C; \\
\text{kParams KDef} = (Dg@Cf); \\
\text{kSuper KDef} = \text{varDefs-names Dg}; \\
\text{kHints KDef} = \text{varDefs-names Cf}; \\
\text{fields}(CT,D) = Dg; \\
CT \vdash M \text{OK IN C }
\end{array}]
\]

\[\Rightarrow CT \vdash \text{CDef OK}\]

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 \( C \).

\[
\text{inductive} \quad \text{ct-typing} :: \text{classTable} \Rightarrow \text{bool} (- \text{OK} 80)
\]

where

\[
\text{ct-all-ok:} \quad [\begin{array}{l}
\text{Object} \notin \text{dom}(CT); \\
\forall C \text{ CDef}. \text{CT}(C) = \text{Some}(CDef) \longrightarrow (CT \vdash \text{CDef OK}) \wedge (\text{cName CDef} = C) \\
\end{array}]
\]

\[\Rightarrow CT \text{ OK}\]

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.

\[
\text{inductive} \quad \text{reduction} :: [\text{classTable}, \text{exp}, \text{exp}] \Rightarrow \text{bool} (- \vdash - \rightarrow - [80,80,80] 80)
\]

where

\[
\text{r-field:} \quad [\begin{array}{l}
\text{fields}(CT,C) = Cf; \\
\text{lookup2 Cf es (\lambda fd.(vdName fd = fi)) = Some(ei)}
\end{array}]
\]

\[\Rightarrow CT \vdash \text{FieldProj (New C es) fi \rightarrow ei}\]

\[
\text{r-invk:} \quad [\begin{array}{l}
\text{mbody}(CT,m,C) = xs . e0;
\end{array}]
\]

11
substs ((map-upds empty xs ds)(this \mapsto (New C es))) e0 = e0’

\[\implies CT \vdash \text{MethodInvk} \ (\text{New C es}) \ m \ ds \to e0’\]

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

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

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

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

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

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

inductive reductions :: [classTable, exp, exp] \Rightarrow \text{bool} \ (- \vdash - \to\to [80,80,80] 80)
where
rs-refl: CT \vdash e \to\to e
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:
assumes ei \in \text{set} \ es
shows \exists el er. es = el@ei#er
using assms
proof (induct es)
case Nil thus ?case by auto
next
case (Cons esh est)
{ assume esh = ei
  with Cons have ?case by blast
}
moreover {
  assume esh ≠ ei
  with Cons have ei ∈ set est by auto
  with Cons obtain el er where esh ≠ est = (esh ≠ el) @ (el ≠ er) by auto
  hence ?case by blast }
ultimately show ?case by blast
qed

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

2.1.2 Maps

lemma map-shuffle:
  assumes \( \text{length xs} = \text{length ys} \)
  shows \( \text{\{xs}\mapsto→\text{ys,x\mapsto→y}\} = \{\text{xs@[x]}\mapsto→\{\text{ys@[y]}\}\} \)
using asms
by (induct xs ys rule: list-induct2) (auto simp add: map-upds-append1)

lemma map-upds-index:
  assumes \( \text{length xs} = \text{length As} \)
  and \( \text{length xs} = \text{length As} \)
  shows \( \exists i. (\text{As}!i = \text{Ai}) \)
  \( \land \ (i < \text{length As}) \)
  \( \land \ (\forall (\text{Bs}::'c \text{ list}). ((\text{length Bs} = \text{length As}) \Rightarrow ((\text{xs}\mapsto→\text{Bs}) \text{x} = \text{Some} (\text{Bs} !i)))) \)
(is \( \exists i. \ ?P i \text{ xs As} \)
is \( \exists i. (?P1 i \text{ As}) \land (?P2 i \text{ As}) \land (\forall \text{Bs}::('c \text{ list}). (?P3 i \text{ xs As Bs}))) \)
using asms
proof (induct xs As rule: list-induct2)
assume \( \square[\mapsto→] \) \( x = \text{Some} \text{ Ai} \)
moreover have \( \neg[\square[\mapsto→]] \) \( x = \text{Some} \text{ Ai} \) by auto
ultimately show \( \exists i. \ ?P1 i \square \) by contradiction
next
fix \( xa \text{ xs y ys} \)
assumption length-xs-ys: \( \text{length xs} = \text{length ys} \)
and IH: \( \text{\{xs\mapsto→ys\} x = \text{Some} \text{ Ai} \Rightarrow \exists i. ?P1 i \text{ xs ys} \)
and map-eq-Some: \( \text{\{xa \# xs \mapsto→ y \# ys\} x = \text{Some} \text{ Ai} \)
then have map-decomp: \( \text{\{xa\#xs \mapsto→ y\#ys\} = [xa\#x\mapsto→y] ++ [xs\mapsto→ys] \) by fastforce
show \( \exists i. ?P i (xa\#xs) (y \# y\)s
proof\(\) (cases \(\{xs \mapsto ys\}\))

case \(\text{Some } Ai'\)
\quad hence \((\{xa \mapsto y\} + + [xs \mapsto ys]) \; x = \text{Some } Ai'\) by (rule map-add-find-right)
\quad hence \(P\): \(\{xs \mapsto ys\}\) \(x = \text{Some } Ai\) using map-eq-Some Some by simp

from \(\text{IH}[\text{OF } P]\) obtain \(i\) where
\(\text{R1: } y_s ! \; i = Ai\)
\quad and \(\text{R2: } i < \text{length } ys\)
\quad and \(\text{pre-r3: } \forall (Bs::'c)\). \(?P3\ i \; xs \; Bs\) by fastforce

\{ fix \(Bs::'c\) list 
\quad assume \(\text{length-Bs: length } Bs = \text{length } (y\#ys)\)
\quad then obtain \(n\) where \(\text{length } (y\#ys) = \text{Suc } n\) by auto
\quad with \(\text{length-Bs}\) obtain \(b\) \(bs\) where \(Bs\text{-def: } Bs = b\#bs\) by (auto simp add:length-Suc-conv)
\quad with \(\text{length-Bs}\) have \(\text{length } ys = \text{length } bs\) by simp
\quad with \(\text{pre-r3}\) have \((\{xa \mapsto b\} + + [xs \mapsto bs]) \; x = \text{Some } (bs!1)\) by (auto simp only:map-add-find-right)
\quad with \(\text{pre-r3}\) \(Bs\text{-def}\) \(\text{length-Bs}\) have \(?P3\ (i+1) \; (xa\#xs) \; (y\#ys)\) \(Bs\) by simp
\}\n
\quad with \(\text{R1 R2}\) have \(?P\ (i+1) \; (xa\#xs) \; (y\#ys)\) by auto
\quad thus \(?\text{thesis }..\)

next
case None
\quad with \(\text{map-decomp map-eq-Some}\) have \((xa \mapsto y) \; x = \text{Some } Ai\) by (auto simp only:map-add-SomeD)
\quad hence \(\text{ai-def: } y = Ai\) and \(x\text{-eq-xa}:x=xa\) by (auto simp only:map-upd-Some-unfold)

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

\section{2.2 FJ Lemmas}

\subsection{2.2.1 Substitution}

\textbf{lemma} \(\text{subst-list1-eq-map-substs} :\)
\(\forall \sigma. \text{subst-list1 } \sigma \; l = \text{map } (\text{subs} \; \sigma) \; l\)
lemma subst-list2-eq-map-substs:
\[ \forall \sigma. \text{subst-list2} \sigma l = \text{map} (\text{subs}t \sigma) l \]
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:
\[ \text{lookup} l f = \text{Some} r \implies f r \]
proof (induct \( l \))
case Nil thus ?case by simp
next
case (Cons \( h t \)) thus ?case by (cases \( f h \)) (auto simp add: lookup. simps)
qed

lemma lookup-hd:
\[ [ \text{length} \ l > 0; f (l!0) ] ] \implies \text{lookup} l f = \text{Some} (l!0) \]
by (induct \( l \)) auto

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

lemma lookup-index:
assumes \( \text{lookup} l1 f = \text{Some} e \)
shows \( \exists i < (\text{length} l1). e = l1!i \land ((\text{length} l1 = \text{length} l2) \implies \text{lookup2} l1 l2 f = \text{Some} (l2!i)) \)
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 {\nassume 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)
}\nultimately have ?case by auto

}
moreover \{ 
  assume \( \neg (f \ h1) \)
  hence \( \text{lookup } t1 \ f = \text{Some } e \)
  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 Cons by blast

  hence \( \text{Suc } i < \text{length } ((h1 \ # t1) \ # t2) \)
  and \( e = ((h1 \ # t1) \ # t2) ! (\text{Suc } i)) \)
  using Cons by (auto simp add: lookup.simps)
\}

ultimately have \(?\text{case}\) by auto

\}

ultimately show \(?\text{case}\) by auto

qed

lemma lookup2-index:
\[ \forall l2. \ (\text{lookup2 } l1 \ l2 \ f = \text{Some } e; \]
\[ \text{length } l1 = \text{length } l2 \] \implies \exists i < \text{length } l2. \ e = (l2!i) \land \text{lookup } l1 \ f = \text{Some } ((l2!i)) \]

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 \( h2 \ l2 \) where \( \text{l2-def}: l2 = h2 \ # t2 \) by (auto simp add: length-Suc-conv)
  \{ 
    assume \( \text{asm} : f \ h1 \)
    hence \( e = h2 \) using Cons \( \text{l2-def}\) by (auto simp add: lookup2.simps)
    hence \( 0 < \text{length } (h2 \ # t2) \} \land \text{lookup } (h1 \ # t1) \ f = \text{Some } ((h1 \ # t1) ! 0) \)
    using Cons by (auto simp add: lookup2.simps)
  
    hence \(?\text{case}\) using \( \text{l2-def}\) by auto
  \}

moreover \{ 
  assume \( \text{asm} : f \ h1 \)
  hence \( \exists i < \text{length } t2. \ e = t2 ! i \land \text{lookup } t1 \ f = \text{Some } ((t1 ! i)) \) using Cons \( \text{l2-def}\) 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 \( (\text{Suc } i) < \text{length } (h2 \ # t2) \land e = ((h2 \ # t2) ! (\text{Suc } i)) \land \text{lookup } (h1 \ # t1) \ f = \text{Some } ((h1 \ # t1) ! (\text{Suc } i)) \)
  using Cons by (force simp add: lookup.simps)
hence \(?\text{case using} \ l2\text{-def by auto}\)
\}
ultimately show \(?\text{case by auto}\)
qed

\textbf{lemma} \ lookup-append:
\begin{itemize}
\item \textbf{assumes} \ lookup \( l \ f = \text{Some} \ r \)
\item \textbf{shows} \ lookup \( (\mathcal{\theta} l') f = \text{Some} \ r \)
\item \textbf{using} \ \texttt{assms by(induct \ l) auto}
\end{itemize}

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

\textbf{2.2.3 Functional}

These lemmas prove that several relations are actually functions

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

\textbf{lemma} \ mbody-functional:
\begin{itemize}
\item \textbf{assumes} \ \text{mbody}\( (\texttt{CT}, m, C) = xs . e0 \)
\item \textbf{and} \ \text{mbody}\( (\texttt{CT}, m, C) = ys . d0 \)
\item \textbf{shows} \ \( xs = ys \wedge e0 = d0 \)
\item \textbf{using} \ \texttt{assms by\ induct (auto \ elim:mbody.cases)}
\end{itemize}

\textbf{lemma} \ fields-functional:
\begin{itemize}
\item \textbf{assumes} \ \text{fields}\( (\texttt{CT}, C) = Cf \)
\item \textbf{and} \ \texttt{CT} \ OK
\item \textbf{shows} \ \( \bigwedge \text{Cf!' . [ \text{fields}(\texttt{CT}, C) = Cf] \implies Cf = Cf'} \)
\item \textbf{using} \ \texttt{assms}
\item \textbf{proof \ induct}
\item \textbf{case} \ (\texttt{f-obj} \texttt{CT})
\item hence \ \texttt{CT(Object)} = \texttt{None} \ \textbf{by (auto \ elim: ct-typing.cases)}
\item \textbf{thus} \ ?\text{case \ using} \ \texttt{f-obj by (auto \ elim: fields.cases)}
\item \textbf{next}
\item \textbf{case} \ (\texttt{f-class} \texttt{CT} \texttt{C CDef D Cf Dg Cg Cf Cg'})
\item hence \ \texttt{f-class-inv:}
\( CT \ C = \text{Some CDef} \) \& \( (\text{cSuper CDef} = D) \) \& \( (\text{cFields CDef} = Cf) \)

and \( CT \ \text{OK by fastforce} + \)

hence \( \text{c-not-obj:} C \neq \text{Object by (force elim:ct-typing.cases)} \)

from \( f\)-\( \text{class} \) have \( \text{flds:fields(CT, C)} = DgCf' \) by fastforce

then obtain \( Dg' \) where

\[ \text{fields(CT, D)} = Dg' \]

and \( DgCf' = Dg' \odot Cf \)

using \( f\)-\( \text{class-inv c-not-obj by (auto elim:fields.cases)} \)

hence \( Dg' = Dg \) using \( f\)-\( \text{class by auto} \)

thus \( \text{?case using } (DgCf = Dg \odot Cf) \) and \( (DgCf' = Dg' \odot Cf) \) by force

\[ \text{qed} \]

\[ 2.2.4 \text{ Subtyping and Typing} \]

\textbf{lemma typings-lengths:}

\textbf{assumes} \( CT; \Gamma \vdash \text{es:Cs} \)

\textbf{shows} \( \text{length es} = \text{length Cs} \)

\textbf{using} \( \text{assms by (induct es Cs) (auto elim:typings.cases)} \)

\textbf{lemma typings-index:}

\textbf{assumes} \( CT; \Gamma \vdash \text{es:Cs} \)

\textbf{shows} \( \forall i. \ [ i < \text{length es} ] \implies CT; \Gamma \vdash (\text{es}!i) : (\text{Cs}!i) \)

\textbf{proof}

\( \text{have length es} = \text{length Cs using assms by (auto simp: typings-lengths)} \)

\( \text{thus } \forall i. \ [ i < \text{length es} ] \implies CT; \Gamma \vdash (\text{es}!i) : (\text{Cs}!i) \)

\textbf{using assms}

\textbf{proof (induct es Cs rule: list-induct2)}

\textbf{case Nil thus } \text{?case by auto}

\textbf{next}

\textbf{case } (\text{Cons esh est hCs tCs i})

\textbf{thus } \text{?case by (cases i) (auto elim:typings.cases)}

\textbf{qed}

\textbf{lemma subtypings-index:}

\textbf{assumes} \( CT \vdash C < : Ds \)

\textbf{shows} \( \forall i. \ [ i < \text{length Cs} ] \implies CT \vdash (\text{Cs}!i) < : (\text{Ds}!i) \)

\textbf{using assms}

\textbf{proof}

\textbf{induct}

\textbf{case ss-nil thus } \text{?case by auto}

\textbf{next}

\textbf{case } (ss-cons hCs CT tCs hDs tDs i)

\textbf{thus } \text{?case by (cases i, auto)}

\textbf{qed}

\textbf{lemma subtyping-append:}

\textbf{assumes} \( CT \vdash C < : Ds \)

and \( CT \vdash C < : D \)

\textbf{shows} \( CT \vdash (\text{Cs}@[C]) < : (\text{Ds}@[D]) \)

\textbf{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@[|])
proof −
  have length es = length Cs using assms by (simp-all add: typings-lengths)
  thus CT;Γ ⊢+ (es@e[|]) : (Cs@[|]) 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 @ [|])
  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) @ [|]) by(auto simp add: typings-typing.ts-cons)
qed
qed

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

lemma ith-subtyping: \(\wedge Ds. [ CT ⊢+ (Cs@(h#t)) <: Ds ] \) ⇒ CT ⊢ h <: (Ds!(length Cs))
proof (induct Cs, auto elim: subtypings.cases)
qed

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

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

qed

lemma ith-typing-sub:
\[ \forall Cs. [\ [ CT;\Gamma \vdash + (es@(^h#t)) : Cs; CT;\Gamma \vdash h' : Ci'; CT \vdash Ci' \vdash (Cs!(\text{length } es))] \] \Rightarrow \exists Cs'. (CT;\Gamma \vdash + (es@(^h'\#t)) : Cs' \land CT \vdash Cs' \vdash : Cs) \]
proof (induct es)
case Nil
then obtain \( hCs \ tCs \)
where \( ts : CT;\Gamma \vdash + t : tCs \)
and Cs-def: \( Cs = hCs \# tCs \) by(auto elim:typings.cases)
from Cs-def Nil have \( CT \vdash hCs' \vdash : hCs \) by auto
with Cs-def have \( CT \vdash + (Ci'\#tCs) \vdash : Cs \) by(auto simp add:subtypings.ss-cons subtypings-refl)
moreover from ts Nil have \( CT;\Gamma \vdash + (h'\#t) : (Ci'\#tCs) \) by(auto simp add:typings-typing.ts-cons)
ultimately show ?case by auto
next
case (Cons eh et)
then obtain \( hCs \ tCs \)
where \( CT;\Gamma \vdash ch : hCs \)
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 \( CT \vdash + tCs' \vdash : tCs \)
by auto
ultimately have \( CT;\Gamma \vdash (eh\#(et@(^h'\#t))) : (hCs\#tCs') \)
and \( CT \vdash (hCs\#tCs') \vdash : Cs \)
by(auto simp add:typings-typing.ts-cons subtypings.ss-cons subtypings.s-refl)
thus ?case by auto
qed

lemma mem-typings:
\[ \forall Cs. [\ [ CT;\Gamma \vdash es:Cs; ei \in \text{set } es]] \Rightarrow \exists Ci. CT;\Gamma \vdash ei:Ci \]
proof (induct es)
case Nil thus ?case by auto
next
case (Cons eh et) thus ?case
  by (cases ei=eh, auto elim:typings.cases)
qed

lemma typings-proj:
assumes CT;Γ ⊢ + ds : As
and CT ⊢ As <: Bs
and length ds = length As
and length ds = length Bs
and i < length ds
shows CT;Γ ⊢ ![i] : ![i] and CT ⊢ ![i] <: ![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 $\text{CT} \vdash Da <: D$ 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 $\forall C. [\text{CT};\text{empty} \vdash e0 : C] \Rightarrow \exists D. \text{CT};\text{empty} \vdash e1 : D$
using assms
  by (induct rule:isubexprs.induct) (auto elim:typing.cases simp add:mem-typings)

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

lemma isubexpr-reduct:
  assumes $d1 \in \text{isubexprs}(e1)$
  shows $\forall d2. [\text{CT} \vdash d1 \rightarrow d2] \Rightarrow \exists e2. \text{CT} \vdash e1 \rightarrow e2$
using assms mem-ith
  by (auto elim:isubexpr.cases intro:reduction.intros,
       force intro:reduction.intros,
       force intro:reduction.intros)

lemma subexpr-reduct:
  assumes $d1 \in \text{subexprs}(e1)$
  shows $\forall d2. [\text{CT} \vdash d1 \rightarrow d2] \Rightarrow \exists e2. \text{CT} \vdash e1 \rightarrow e2$
using assms
  by (induct rule:rtrancl.induct) (auto, force simp add:isubexpr-reduct)
end

3 FJSound: Type Soundness

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

3.1 Method Type and Body Connection

lemma mtype-mbody:
fixes $Cs :: \text{nat list}$
assumes $\text{mtype}(CT, m, C) = Cs \rightarrow C0$
shows $\exists xs \cdot \text{mbody}(CT, m, C) = xs \cdot e \land \text{length} \; xs = \text{length} \; Cs$
using assms
proof (induct rule: mtype.induct)
  case (mt-class $C0 \; Cs \; C \; CDef \; CT \; m \; mDef$)
  thus $?case$
    by (force simp add: varDefs-types-def varDefs-names-def elim: mtype.cases intro: mbody.mb-class)
next
  case (mt-super $CT \; C0 \; CDef \; m \; D \; Cs \; C$)
  then obtain $xs \; e$
    where $\text{mbody}(CT, m, D) = xs \cdot e$ and $\text{length} \; xs = \text{length} \; Cs$
    by auto
  thus $?case$ using mt-super by (auto intro: mbody.mb-super)
qed

lemma mtype-mbody-length:
assumes $\text{mt} \colon \text{mtype}(CT, m, C) = Cs \rightarrow C0$
and $\text{mb} \colon \text{mbody}(CT, m, C) = xs \cdot e$
shows $\text{length} \; xs = \text{length} \; Cs$
proof
  from mtype-mbody[OF $\text{mt}$]
  obtain $xs' \; e'$
    where $\text{mb2} \colon \text{mbody}(CT, m, C) = xs' \cdot e'$
    and $\text{length} \; xs' = \text{length} \; Cs$
    by auto
  with mbody-functional[OF $\text{mb} \; \text{mb2}$]
  show $?thesis$ by auto
qed

3.2 Method Types and Field Declarations of Subtypes

lemma A-1-1:
assumes $CT \vdash C \triangleleft D$ and $CT \text{ OK}$
shows $(\text{mtype}(CT, m, D) = Cs \rightarrow C0) \implies (\text{mtype}(CT, m, C) = Cs \rightarrow C0)$
using assms
proof (induct rule: subtyping.induct)
  case (s-refl $C \; CT$)
  show $?case$ by fact
next
  case (s-trans $C \; CT \; D \; E$)
  thus $?case$ by auto
next
  case (s-super $CT \; C \; CDef \; D$)
  hence $CT \vdash CDef \text{ OK}$ and $\text{cName} \; CDef = C$
    by (auto elim: ct-typing.cases)
  with s-super obtain $M$
    where $M \colon CT \vdash M \text{ OK IN C}$ and $\text{cMethods} \colon \text{cMethods} \; CDef = M$
    by (auto elim: class-typing.cases)
  let $?lookup-m = \text{lookup} \; M \; (\lambda md. \; (\text{mName} \; md \; = m))$
  show $?case$
    proof (cases $\exists mDef. \; ?lookup-m = \text{Some} \; mDef$)
      case True
      case False
then obtain $mDef$ where $m: ?lookup-m = Some mDef$ by (rule exE)
hence $mDef-name: mName mDef = m$ by (rule lookup-true)

have $\text{CT} \vdash mDef \text{ OK IN C using } M m$ by (auto simp add: method-typings-lookup)
then obtain $CDef'$ $m'$ $D'$ $Cs'$ $C0'$
where $\text{CT}: \text{CT C} = \text{Some CDef'}$
and $cSuper CDef' = D'$
and $mName mDef = m'$
and $mReturn: mReturn mDef = C0'$
and $\forall Ds D0. (\text{mtype}(\text{CT}, m', D') = Ds \to D0) \longrightarrow Cs' = Ds \land C0' = D0$
by (auto elim: method-typing. cases)

with $s-super mDef-name$ have $CDef = CDef'$
and $D = D'$
and $m = m'$
and $\forall Ds D0. (\text{mtype}(\text{CT}, m, D) = Ds \to D0) \longrightarrow Cs' = Ds \land C0' = D0$
by auto
thus $\text{thesis}$ using $s-super cMethods m \text{ CT mReturn varDefs-types}$ by (auto intro: mtype. intros)

next

case False

hence $?lookup-m = \text{None}$ by (simp add: lookup-split)
then show $\text{thesis}$ using $s-super cMethods$ by (auto simp add: mtype. intros)

qed

lemma sub-fields:
assumes $\text{CT} \vdash C <: D$
shows $\forall Dg. \text{fields}(\text{CT}, D) = Dg \Longrightarrow \exists Cf. \text{fields}(\text{CT}, C) = (Dg@Cf)$
using assms
proof induct
  case (s-refl CT C)
  hence $\text{fields}(\text{CT}, C) = (Dg@[[]])$ by simp
  thus $?case$ ..
next
  case (s-trans CT C D E)
  then obtain $Df Cf$ where $\text{fields}(\text{CT}, C) = ((Dg@Df)@Cf)$ by force
  thus $?case$ by auto
next
  case (s-super CT C CDef D Dg)
  then obtain $Cf$ where $\text{cFields CDef} = Cf$ by force
  with $s-super$ have $\text{fields}(\text{CT}, C) = (Dg@Cf)$ by (simp add: f-class)
  thus $?case$ ..

qed

3.3 Substitution Lemma

lemma $A-1-2$: assumes $CT \text{ OK}$
and $\Gamma = \Gamma_1 ++ \Gamma_2$
and $\Gamma_2 = \{x \mapsto Bs\}$
and $\text{length } xs = \text{length } ds$
and $\text{length } Bs = \text{length } ds$
and $\exists As. CT;\Gamma_1 \vdash ds : As \land CT \vdash As <: Bs$
shows $CT;\Gamma \vdash es;Ds \implies \exists Cs. (CT;\Gamma_1 \vdash (\{ds/ xs\}es);Cs \land CT \vdash Cs <: Ds)$ (is ?TYPINGS $\implies \ ?P1$)
and $CT;\Gamma \vdash e;D \implies \exists C. (CT;\Gamma_1 \vdash ((ds/ xs)e);C \land CT \vdash C <: D)$ (is ?TYPING $\implies \ ?P2$)

**proof**

- let $\text{COMMON-ASMS} = (CT \text{ OK}) \land (\Gamma = \Gamma_1 ++ \Gamma_2) \land (\Gamma_2 = \{xs \mapsto Bs\}) \land (\text{length } Bs = \text{length } ds) \land (\exists As. CT;\Gamma_1 \vdash ds : As \land CT \vdash As <: Bs)$

- have $\text{RESULT: } (\text{?TYPINGS} \implies \text{?COMMON-ASMS} \implies \ ?P1)$

- and $\text{?TYPING} \implies \text{?COMMON-ASMS} \implies \ ?P2$)

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

case (ts-nil CT \Gamma)

show $\ ?case$

**proof** (rule impI)

- have $(\text{CT};\Gamma_1 \vdash (\{ds/ xs\}es));Cs \land (CT \vdash Cs <: []))$ by (auto simp add: typings-typing intros substtyping-intros)

- then show $\exists Cs.(\text{CT};\Gamma_1 \vdash (\{ds/ xs\}es);Cs) \land (CT \vdash Cs <: [])$ by auto

qed

next

case (ts-cons CT \Gamma e0 C0 es Cs')

show $\ ?case$

**proof** (rule impI)

- assume asms: $(CT \text{ OK}) \land (\Gamma = \Gamma_1 ++ \Gamma_2) \land (\Gamma_2 = \{xs \mapsto Bs\}) \land (\text{length } Bs = \text{length } ds) \land (\exists As. CT;\Gamma_1 \vdash ds : As \land CT \vdash As <: Bs)$

- with ts-cons have $e0$-typ: $CT;\Gamma \vdash e0 : C0$ by fastforce

- with ts-cons asms have

  \[ \exists C.(\text{CT};\Gamma_1 \vdash (\{ds/ xs\}e0);C) \land (CT \vdash C <: C0) \]

  and $\exists Cs.(\text{CT};\Gamma_1 \vdash [ds/ xs]es ; Cs) \land (CT \vdash Cs <: Cs')$

  by auto

- then obtain $C Cs$ where

  $(\text{CT};\Gamma_1 \vdash (\{ds/ xs\} e0); C) \land (CT \vdash C <: C0)$

  and $(\text{CT};\Gamma_1 \vdash [ds/ xs]es ; Cs) \land (CT \vdash Cs <: Cs')$ by auto

- hence $\text{CT};\Gamma_1 \vdash [ds/ xs]{e0 # es} : (C # Cs)$

  and $\text{CT} \vdash (C # Cs) <: (C0 # Cs')$

  by (auto simp add: typings-typing intros substtyping-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

case (t-var \Gamma x C' CT)

show $\ ?case$

**proof** (rule impI)

- assume asms: $(CT \text{ OK}) \land (\Gamma = \Gamma_1 ++ \Gamma_2) \land (\Gamma_2 = \{xs \mapsto Bs\}) \land (\text{length } Bs = \text{length } ds) \land (\exists As. CT;\Gamma_1 \vdash ds : As \land CT \vdash As <: Bs)$

25
hence
lengths: length $ds = length \, Bs$
and $G$-def: $\Gamma = \Gamma 1 ++ \Gamma 2$
and $G$-2-def : $\Gamma 2 = [xs \mapsto Bs]$ by auto
from lengths $G$-2-def have same-doms: $dom([xs \mapsto ds]) = dom(\Gamma 2)$ by auto
from asms show $\exists C. CT;\Gamma 1 \vdash subs \, [xs \mapsto ds]$ $(\Var \, x) : C \land CT \vdash C$
\:\\< C'
proof (cases $\Gamma 2 \, x$)
case None
with $G$-def t-var have $G$-1-x: $\Gamma 1 \, x = Some \, C'$ by (simp add: map-add-Some-iff)
from None same-doms have $x \notin dom([xs \mapsto ds])$ by (auto simp only: domIff)
\begin{itemize}
\item hence $[xs \mapsto ds]x = None$ by (auto simp only: map-add-Some-iff)
\item hence $(ds/xs)(\Var \, x) = (\Var \, x)$ by auto
\end{itemize}
with $G$-1-x have
\begin{itemize}
\item $CT;\Gamma 1 \vdash (ds/xs)(\Var \, x) : C'$ and $CT \vdash C' \lhd C'$
\item by (auto simp add: typings-typing.intros subtyping.intros)
\item thus $?thesis$ by auto
\end{itemize}
next
case (Some $Bi$)
with $G$-def t-var have $\, c'\cdot eq\cdot bi$: $C' = Bi$ by (auto simp add: map-add-SomeD)
from lengths $xs = length \, ds$: asms have $length \, xs = length \, Bs$ by simp
with Some $G$-2-def have $\exists \, i. (Bs/i = Bi) \land (i < length \, Bs) \land$
\begin{itemize}
\item $(\forall \, i.((length \, l = length \, Bs) \rightarrow ([xs \mapsto l] \, x = Some \, (!!i))))$
\end{itemize}
by (auto simp add: map-upds-index)
then obtain $i$ where $bs\cdot i\cdot proj$: $(Bs/i = Bi)$
and $i\cdot len$: $i < length \, Bs$
and $P$: $((\forall \, l::exp \, list).((length \, l = length \, Bs) \longrightarrow ([xs \mapsto l] \, x = Some \, (!!i))))$)
\begin{itemize}
\item by fastforce
\end{itemize}
from lengths $P$ have $subst\cdot x$: $([xs \mapsto ds]x = Some \, (ds/i))$ by auto
from asms obtain $As$ where as-ex:$CT;\Gamma 1 \vdash ds : As \land CT \vdash As \lhd As$ \\\:< $Bs$ by fastforce
hence
\begin{itemize}
\item $length \, As = length \, Bs$ by (auto simp add: subtypings-length)
\item $proj\cdot i$: $CT;\Gamma 1 \vdash ds/i : As/i \land CT \vdash As/i \lhd Bs/i$
\end{itemize}
using $i\cdot len$ lengths as-ex by (auto simp add: typings-proj)
\begin{itemize}
\item $CT;\Gamma 1 \vdash (ds/xs)(\Var \, x) : As/i \land CT \vdash As/i \lhd C'$
\item using $\, c'\cdot eq\cdot bi \, bs\cdot i\cdot proj \, subst\cdot x$ by auto
\item thus $?thesis$ ..
\end{itemize}
qed
qed
next
case $(t\cdot field \, CT \, \Gamma \, e0 \, C0 \, Cf \, fi \, fDef \, Ci)$
\begin{itemize}
\item show $?case$
\end{itemize}
proof (rule impI)
assume asms: $(CT \, OK) \land (\Gamma = \Gamma 1 ++ \Gamma 2) \land$
\begin{itemize}
\item $(\Gamma 2 = [xs \mapsto Bs]) \land (length \, Bs = length \, ds) \land (\exists \, As. CT;\Gamma 1 \vdash + \, ds : As \land CT \vdash + \, As \lhd Bs)$
\end{itemize}
from $t\cdot field$ have $flds$: $fields(CT,C0) = Cf$ 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;ΓI ⊢ (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;ΓI ⊢ (ds/xs)(FieldProj e0 fi) : C ∧ CT ⊢ C <: C

by auto

qed

next

case(t-invk CT Γ e0 C0 m Ds C es Cs)

show ?case

proof (rule impI)

assume asms: (CT OK) ∧ (Γ = ΓI ++ Γ2) ∧ (Γ2 = [xs [→→] Bs])

∧ (length Bs = length ds) ∧ (∃ As. CT;ΓI ⊢+ ds : As ∧ CT ⊢+ As <: Bs)

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 subs' subs]

lens

subst-e

have CT;ΓI ⊢ (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;ΓI ⊢ (ds/xs)(MethodInvk e0 m es) : C' ∧ CT ⊢ 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 ↦ Bsi]) ∧ (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 vlds: varDefs-types Df = Ds
by auto
from t-new asms obtain Cs' where
es-typ: CT; Γ1 ⊢+ (ds/xs)es : Cs' and subst1: CT ⊢+ Cs <: Cs by auto
have subst-e: (ds/xs)(New C es) = New C ([ds/xs]es)
by(auto simp add: subst-list2-eq-map-substs)
from es-typ subtypings-trans[OF subst1 subs] flds subst-e len vlds
have CT; Γ1 ⊢ (ds/xs)(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; Γ1 ⊢ (ds/xs)(New C es) : C' ∧ CT ⊢ C' <: C
by auto
qed
next
case(t-ucast CT Γ e0 D C)
show ?case
proof(rule impI)
assume asms: (CT OK) ∧ (Γ = Γ1 ++ Γ2) ∧ (Γ2 = [xs ↦ Bsi]) ∧ (length Bs = length ds) ∧ (∃ As. CT; Γ1 ⊢+ ds : As ∧ CT ⊢+ As <: Bs)
from t-ucast asms obtain C' where e0-typ: CT; Γ1 ⊢ (ds/xs)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; Γ1 ⊢ (ds/xs)(Cast C e0) : C by(auto simp add: typings-typing.intros)
moreover have CT ⊢ C <: C by (rule s-refl)
ultimately show ∃ C'. CT; Γ1 ⊢ (ds/xs)(Cast C e0) : C' ∧ CT ⊢ C' <: C
by auto
qed
next
case(t-dcast CT Γ e0 D C)
show ?case
proof(rule impI)
assume asms: (CT OK) ∧ (Γ = Γ1 ++ Γ2) ∧ (Γ2 = [xs ↦ Bsi]) ∧ (length Bs = length ds) ∧ (∃ As. CT; Γ1 ⊢+ ds : As ∧ CT ⊢+ As <: Bs)
from t-dcast asms obtain C' where e0-typ: CT; Γ1 ⊢ (ds/xs)e0 : C' by auto
have (CT ⊢ C' <: C) ∨
  (C =′ C' ∧ CT ⊢ C <: C') ∨
  (CT ⊢ C = C' <: C') ∨
  (CT ⊢ C = C' <: C') ∨
moreover
  { assume CT ⊢ C' <: C
    with e0-typ have CT; Γ1 ⊢ (ds/xs) (Cast C e0) : C by (auto simp add:
 typings-typing.intros)
    |
    moreover
    { assume \((C \neq C' \land CT \vdash C <: C')\)
      with e0-typ have \(CT;\Gamma I \vdash (\text{ds}/\text{xs})\) \((\text{Cast} C e0) : C\) by (auto simp add: typings-typing.intros)
    }
    moreover
    { assume \((CT \vdash C \not<: C' \land CT \vdash C' \not<: C)\)
      with e0-typ have \(CT;\Gamma I \vdash (\text{ds}/\text{xs})\) \((\text{Cast} C e0) : C\) by (auto simp add: typings-typing.intros)
    }

 ultimately have \(CT;\Gamma I \vdash (\text{ds}/\text{xs})\) \((\text{Cast} C e0) : C\) by auto

 moreover have \(CT \vdash C <: C\) by (rule s-refl)

 ultimately show \(\exists C'. CT;\Gamma I \vdash (\text{ds}/\text{xs})(\text{Cast} C e0) : C' \land CT \vdash C' <: C\)

 by auto

 qed

ext next

case\((t\text{-cast} CT \Gamma e0 D C)\)

 show ?case

 proof (rule impI)

 assume asms: \((CT \text{ OK}) \land (\Gamma = \Gamma I ++ \Gamma 2) \land (\Gamma 2 = \{xs \mapsto Bs\}) \land (\text{length } Bs = \text{length } ds) \land (\exists As. CT;\Gamma 1 \vdash ds : As \land CT \vdash As <: Bs)\)

 from \(t\text{-cast asms} \) obtain \(C'\) \(\text{ where} \) e0-typ: \(CT;\Gamma I \vdash (\text{ds}/\text{xs}) e0 : C'\)

 and sub1: \(CT \vdash C' <: D\)

 and nsu1: \(CT \vdash C : D\)

 and nsu2: \(CT \vdash C' : D\) by auto

 from \(\text{not-subtypes[OF sub1 nsu1 nsu2]}\) have \(CT \vdash C' \not<: C\) \(\text{ by fastforce}\)

 moreover have \(CT \vdash C : D\) \(\text{ using sub1 by (rule s-trans)}\)

 with nsu1 show False by auto

 qed

 ultimately have \(CT;\Gamma I \vdash (\text{ds}/\text{xs})\) \((\text{Cast} C e0) : C\) using e0-typ by (auto simp add: typings-typing.intros)

 thus \(\exists C'. CT;\Gamma I \vdash (\text{ds}/\text{xs})(\text{Cast} C e0) : C' \land CT \vdash C' <: C\) by (auto simp add: s-refl)

 qed

 thus \(?\text{TYPINGS} \Rightarrow \ ?P1\) \(\text{ and} \ ?\text{TYPING} \Rightarrow \ ?P2\) \(\text{ using asms by auto}\)

 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 \((CT;\Gamma 2 \vdash+ es : Cs) \Rightarrow (CT;\Gamma 1++\Gamma 2 \vdash+ es : Cs)\) \((\text{is} \ ?P1 \Rightarrow \ ?P2)\)
and \( CT;\Gamma_2 \vdash e : C \Rightarrow CT;\Gamma_1;+;\Gamma_2 \vdash e : C \) (is \( ?Q1 \Rightarrow ?Q2 \))

proof -
have \((?P1 \Rightarrow ?P2) \land (?Q1 \Rightarrow ?Q2)\)
by (induct rule: typings-typing, induct, auto simp add: map-add-find-right typings-typing,intros)

thus \(?P1 \Rightarrow ?P2\) and \(?Q1 \Rightarrow ?Q2\) by auto
qed

3.5 Method Body Typing Lemma

lemma A-1-4:
assumes ct-ok: \( CT \ OK \)
and mb:mbody(\( CT,m,C \)) = \( xs . e \)
and mt:mtype(\( CT,m,C \)) = \( Ds \Rightarrow D \)
shows \( \exists D0 C0. \ (CT \vdash C << D0) \land \ (CT \vdash C0 << D) \land \ (CT;[xs[\rightarrow]Ds,this[\rightarrow]C] \vdash e : C0)\)
using mb ct-ok mt proof(induct rule: mbody.induct)
case (mb-class \( CT \ C \ CDef \ m \ mDef \ zs \ e \))
hence
\( m-param:varDefs-types (mParams mDef) = Ds \)
and \( m-ret:mReturn mDef = D \)
and \( CT \vdash CDef OK \)
and cName CDef = \( C \)
by (auto elim:mtype.cases ct-typing.cases)
hence \( CT \vdash (cMethods CDef) \ OK \ IN \ C \) by (auto elim:class-typing.cases)
hence \( CT \vdash mDef \ OK \ IN \ C \) using mb-class by(auto simp add:method-typings-lookup)
hence \( \exists E0. \ ((CT;[xs[\rightarrow]Ds,this[\rightarrow]C] \vdash e : E0) \land (CT \vdash E0 << D))\)
using mb-class m-param m-ret by(auto elim:method-typing.cases)
then obtain \( E0 \)
where \( CT;[xs[\rightarrow]Ds,this[\rightarrow]C] \vdash e : E0 \)
and \( CT \vdash E0 << D \)
and \( CT \vdash C << C \) by (auto simp add: s-refl)
thus \(?case\) by blast

next
case (mb-super \( CT \ C \ CDef \ m \ Da \ xs \ e \))
hence ct: \( CT \ OK \)
and IH: \( [CT \ OK; \ mtype(CT,m,Da) = Ds \Rightarrow D] \)
\( \Rightarrow \exists D0 C0. \ (CT \vdash Da << D0) \land (CT \vdash C0 << D) \land (CT;[xs[\rightarrow]Ds,this[\rightarrow]E0] \vdash e:C0) \) by fastforce+
from mb-super have c-sub-da: \( CT \vdash C << Da \) by (auto simp add:s-super)
from mb-super have mt:mtype(\( CT,m,Da \)) = \( Ds \Rightarrow D \) by (auto elim: mtype.cases)
from IH[OF ct mt] obtain \( D0 C0 \)
where s1: \( CT \vdash Da << D0 \)
and \( CT \vdash C0 << D \)
and \( CT;[xs[\rightarrow]Ds,this[\rightarrow]D0] \vdash e : C0 \) by auto
thus \(?case\) using s-trans[OF c-sub-da s1] by blast
qed
3.6 Subject Reduction Theorem

theorem Thm-2-4-1:
assumes CT ⊢ 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')
then obtain Ca Cf' fDef
where new-typ: CT;Γ ⊢ New Ca es : Ca'
and flds: fields(CT,Ca) = Cf
and fields(CT,Ca') = Cf'
and lookup: lookup CF' (λfd. vdName fd = fi) = Some fDef
and C-def: vType fDef = C by (auto elim: typing.cases)
hence Ca-Ca': Ca = Ca' by (auto elim:typing.cases)
with flds' have CF-CF': Cf = Cf'' by(auto 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' = e@i
and lookup CF (λfd. vdName fd = fi) = Some (CF[i]) by auto
moreover
with C-def Ds-def lookup lookup2 have Dsl'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: typings-index subtypings-index typings-lengths)
ultimately show ?case by auto
next
case (r-invk CT m Ca xs e ds es e')
from r-invk have mb: mbBody(CT,m,Ca) = xs . e by fastforce
from r-invk obtain Ca' Ds Cs
where CT;Γ ⊢ New Ca es : Ca'
and mttype(CT,m,Ca') = Cs → C
and ds-typs: CT;Γ ⊢+ ds : Ds
and Ds-subs: CT ⊢+ Ds <: Cs
and \( \Pi \): length \( ds = length \) \( Cs \) by (auto elim:typing_cases)

hence new-typ: \( CT ; \Gamma \vdash New \) \( Ca \) es : \( Ca \)

and mt: mtype(\( CT , m , Ca \)) = \( Cs \rightarrow C \) by (auto elim:typing_cases)

from ds-typs new-typ have \( CT , \Gamma \vdash ( ds \circ {[New \ Ca \ es]} ) : (Ds \circ {[Ca]}) \)

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: \( CT ; [xs \mapsto C] , this \rightarrow Da \) \( \vdash e : E \) by auto

moreover with Ds-subs have \( CT \vdash ( Ds \circ {[Ca]}) : (Cs \circ {[Da]}) \)

by (auto simp add: subtyping-append)

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

by auto

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

by (simp only:A-1-3)

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

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

by (force simp only:map-shuffle)

let \( \?\Gamma 1 = \Gamma \) and \( \?\Gamma 2 = [xs@[this]] \rightarrow (Cs@[Da]) \)

have g-def: \( \?\Gamma 1 \uplus \?\Gamma 2 = (\?\Gamma 1 \uplus \?\Gamma 2) \) and g2-def: \( \?\Gamma 2 = \?\Gamma 2 \) by auto

from A-1-2[OF - g-def g2-def - - ex] e0-typ3 r-invk \( \Pi \) mtype-mbody-length[OF mt mb]

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

and \( E'\text{-sub-E} : CT \vdash E' < : E \) by force

moreover from \( e'\text{-typ} \) \( \Pi \) mtype-mbody-length[OF mt mb]

have \( CT ; \Gamma \vdash \text{subs} [xs \mapsto ds , this \rightarrow (New \ Ca \ es)] e : E' \)

by (auto simp only:map-shuffle)

moreover from \( E'\text{-sub-E} \) \( E\text{-sub-C} \) have \( CT \vdash E' < : C \) by (rule subtyping,s-trans)

ultimately show \( ?\text{case using} \ r\text{-invk} \) by auto

next

case \( (r\text{-cast} \ CT \ Ca \ D \ es) \)

then obtain \( Ca' \)

where \( C = D \)

and \( CT ; \Gamma \vdash New \ Ca \ es : Ca' \) by (auto elim:typing_cases)

thus \( ?\text{case using} \ r\text{-cast} \) by (auto elim:typing_cases)

next

case \( (r\text{-field} \ CT \ e0 \ e0' \ f) \)

then obtain \( C0 \ Cf \ f d \) where \( CT ; \Gamma \vdash e0 : C0 \)

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

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

and \( vdType fd = C \)

by (auto elim:typing_cases)

moreover with \( r\text{-field} \) obtain \( C' \)

where \( CT ; \Gamma \vdash e0' : C' \)

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

moreover from sub-fields[OF - Cf-def] obtain \( Cf' \)

32
where \( \text{fields}(CT, C') = (\text{elf} @ \text{elf}') \) by rule (\( \text{rule } (CT \vdash C' <: C0) \))
m辦理 with \( \text{fd-def} \) have lookup (\( \text{elf} @ \text{elf}' \)) (\( \lambda \text{fd. } (\text{vdName fd = f}) \)) = Some \( \text{fd} \)
by (\( \text{simp add:lookup-append} \))
ultimately have \( CT; \Gamma \vdash \text{FieldProj e0'} f : C \) by (\( \text{auto simp add:typings-typing.t-field} \))
thus \( ?\text{case by } (\text{auto simp add:subtyping.s-refl}) \)
next
case (\( \text{rc-invk-recc} \) CT e0 e0' m es C)
then obtain C0 Ds Cs
where \( \text{ct-ok}: CT \text{ OK} \)
and \( CT; \Gamma \vdash e0 : C0 \)
and \( \text{mt: mtype}(CT, m, C0) = Ds \to C \)
and \( CT; \Gamma \vdash e0 : C0 \)
and \( \text{length es = length Ds} \)
and \( CT \vdash e0' <: Ds \)
by (\( \text{auto elim:typing.cases} \))
m辦理 with \( \text{rc-invk-recc} \) obtain C0'
where \( \text{CT; } \Gamma \vdash e0' : C0' \)
and \( CT \vdash C0' <: C0 \) by \( \text{auto} \)
m辦理 with \( \text{A-1-1} \) [\( \text{OF - ct-ok mt} \)] have \( \text{mttype}(CT, m, C0') = Ds \to C \) by \( \text{simp} \)
ultimately have \( CT; \Gamma \vdash \text{MethodInvk e0} m \) es C by (\( \text{auto simp add:typings-typing.t-invk} \))
thus \( ?\text{case by } (\text{auto simp add:subtyping.s-refl}) \)
next
case (\( \text{rc-invk-arg} \) CT ei ei' e0 m er C)
then obtain Cs Ds C0
where \( \text{typs: } CT; \Gamma \vdash+ (\text{el}@[ei#er]) : Cs \)
and \( \text{e0-typ: } CT; \Gamma \vdash e0 : C0 \)
and \( \text{mt: mtype}(CT, m, C0) = Ds \to C \)
and \( \text{Cs-sub-Ds: } CT \vdash+ Cs <: Ds \)
and \( \text{len: length (el@[ei#er]) = length Ds} \)
by (\( \text{auto elim:typing.cases} \))
hence \( CT; \Gamma \vdash ei: (Cs!(\text{length el})) \) by (\( \text{simp add:ith-typing} \))
with \( \text{rc-invk-arg} \) obtain Ci'
where \( \text{ei-typ: } CT; \Gamma \vdash ei': Ci' \)
and \( \text{Ci-sub: } CT \vdash Ci' <: (Cs!(\text{length el})) \)
by \( \text{auto} \)
from \( \text{ith-typing-sub}[\text{OF typs ei-typ Ci-sub}] \) obtain Cs'
where \( \text{es'-typs: } CT; \Gamma \vdash+ (\text{el}@[ei#er]) : Cs' \)
and \( \text{Cs'-sub-Cs: } CT \vdash+ Cs' <: Cs \) by \( \text{auto} \)
from \( \text{len have} \) length (el@[ei#er]) = length Ds by \( \text{simp} \)
with \( \text{es'-typs} \) subtypings-trans[\( \text{OF Cs'-sub-Cs Cs-sub-Ds} \)] e0-typ mt have
\( CT; \Gamma \vdash \text{MethodInvk e0 m } (\text{el}@[ei#er]) : C \)
by (\( \text{auto simp add:typings-typing.t-invk} \))
thus \( ?\text{case by } (\text{auto simp add:subtyping.s-refl}) \)
next
case (\( \text{rc-new-arg} \) CT ei ei' Ca el er C)
then obtain Cs Df Ds
where \( \text{typs: } CT; \Gamma \vdash+ (\text{el}@[ei#er]) : Cs \)
and \( \text{flds} \): fields\((CT,C) = Df \)
and \( \text{len} \): length \((el@\text{#er})\) = length \(Df \)
and \( Ds-def \): varDefs\(\text{-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;\(\Gamma \vdash \text{ei} \colon (Cs!(\text{length } el)) \) by (simp add:ith-typing)
with rc-new-arg obtain Ci'
  where ei-typ: CT;\(\Gamma \vdash \text{ei'} \colon Ci' \)
  and Ci-sub: CT \(\vdash \text{Ci'} \:<\ (Cs!(\text{length } el)) \)
by auto
from ith-typing-sub[of tys ei-typ Ci-sub] obtain Cs'
  where es'-tys: CT;\(\Gamma \vdash \) \((el@\text{#er})\) \:<\ Cs'
  and Cs'-sub-Cs: CT \(\vdash \) \(Cs' \:<\ C \)
by auto
have (CT;\(\Gamma \vdash \text{New} \ Ca \ (el@\text{#er}) \colon C \)
by (auto simp add:typings-typing.t-new)
thus ?case by (auto simp add:subtyping.s-refl)
next
case (rc-cast CT \(e_0 \) \(e_0' \colon C \ Ca) \)
then obtain D
  where CT;\(\Gamma \vdash \) \(e_0 \colon D \)
  and Ca-def: Ca = C
by (auto elim:typing.cases)
with rc-cast obtain D'
  where \(e_0'\)-typ: CT;\(\Gamma \vdash \) \(e_0' \colon D' \) and CT \(\vdash \) \(D' \:<\ D \)
by auto
have (CT \(\vdash \) \(D' \:<\ C \) \) \(\lor \)
  \((C \neq D' \land CT \vdash \) \(C \:<\ D') \) \(\lor \)
  \((CT \vdash \) \(C \:<\ D' \land CT \vdash \) \(D' \:<\ C \) \)
by blast
moreover {
  assume CT \(\vdash \) \(D' \:<\ C \)
  with \(e_0'\)-typ have CT;\(\Gamma \vdash \text{Cast} \ C \ e_0' \colon C \)
  by (auto simp add:typings-typing.t-ucast)
}
moreover {
  assume \((C \neq D' \land CT \vdash \) \(C \:<\ D') \)
  with \(e_0'\)-typ have CT;\(\Gamma \vdash \text{Cast} \ C \ e_0' \colon C \)
  by (auto simp add:typings-typing.t-dcast)
}
moreover {
  assume \((CT \vdash \) C \(<\ D' \land CT \vdash \) D' \(<\ C \) \)
  with \(e_0'\)-typ have CT;\(\Gamma \vdash \text{Cast} \ C \ e_0' \colon C \)
  by (auto simp add:typings-typing.t-scast)
}
ultimately have CT;\(\Gamma \vdash \text{Cast} \ C \ e_0' \colon C \)
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

34
shows $\bigwedge C. \{ CT;\Gamma \vdash e : C \} \implies \exists C'. (CT;\Gamma \vdash e' : C' \land CT \vdash C' <: C)$

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: $\bigwedge D. [CT;\Gamma \vdash e' : D; CT OK] \implies \exists E. CT;\Gamma \vdash e'' : E \land CT \vdash E <: D$
  by auto

  from Thm-2-4-1[of e-step ct-ok e-typ] obtain D where
  s-trans $CT;\Gamma \vdash e' : D; CT OK$
  by auto

  moreover from s-trans[of E-sub-D D-sub-C] have
  $CT \vdash E <: D$ by auto

  ultimately show ?thesis
  by auto

qed

3.8 Progress

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

theorem Thm-2-4-2-1:
  assumes CT;empty \vdash e : C
  and FieldProj (New C0 es) fi \in subexprs(e)
  shows $\exists Cf fDef. fields(CT, C0) = Cf \land lookup Cf (\lambda fd. (vdName fd = fi)) = Some fDef$

proof

  obtain Ci where
  CT;empty \vdash (FieldProj (New C0 es) fi) : Ci
  using assms by (force simp add: subexpr-typing)

  then obtain Cf fDef C0'
    where CT;empty \vdash (New C0 es) : C0'
    and fields(CT, C0') = Cf
    and lookup Cf (\lambda 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;empty \vdash e : C
  and MethodInvk (New C0 es) m ds \in subexprs(e)
  shows $\exists xs e0. mbody(CT,m,C0) = xs . e0 \land length xs = length ds$

proof

obtain \( D \) where \( CT;\emptyset \vdash \) MethodInvk \((\text{New } C0 \ es)\) \( m \ ds : D \)
using assms by (force simp add:subexpr-typing)

then obtain \( C0' \) \( Cs \)
where \( CT;\emptyset \vdash \) \( (\text{New } C0 \ es) : C0' \)
and \( \text{mt:mtyp}(CT,m,C0') = Cs \rightarrow D \)
and \( \text{length } ds = \text{length } Cs \)
by (auto elim:typing.cases)

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

qed

lemma closed-subterm-split:
assumes \( CT;\Gamma \vdash e : C \) and \( \Gamma = \emptyset \)
shows
\[ (\exists C0 \ es \ fi. \ (\text{FieldProj} \ (\text{New } C0 \ es) \ fi) \in \text{subexprs}(e)) \]
\[ \lor (\exists C0 \ es \ m \ ds. \ (\text{MethodInvk} \ (\text{New } C0 \ es) \ m \ ds) \in \text{subexprs}(e)) \]
\[ \lor (\exists C0 \ D \ es. \ (\text{Cast } D \ (\text{New } C0 \ es)) \in \text{subexprs}(e)) \]
\[ \lor \text{val(e)} \ (\text{is } ?F \ e \ \lor \ ?M \ e \ \lor \ ?C \ e \ \lor \ ?V \ e \ \text{is } ?IH \ e) \]
using assms

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

case 1 thus \( ?\text{case} \) using assms by auto

next
case \( 2 \ C \ CT \Gamma x \) thus \( ?\text{case} \) by auto

next
case \( 3 \ C0 \ Ct \ Cf \ Ci \Gamma e0 fDef fi \)

have \( s1 : e0 \in \text{subexprs}(\text{FieldProj } e0 \ fi) \) by(auto simp add:isubexprs.intros)
from \( 3 \) have \( ?IH e0 \) by auto
moreover
\{ assume \( ?F e0 \) then obtain \( C0 \ es \ fi' \) where \( s2 : \text{FieldProj} \ (\text{New } C0 \ es) \ fi' \in \text{subexprs}(e0) \) by auto
from rtrancl-trans[OF \( s2 s1 \)] have \( ?\text{case} \) by auto
\}

moreover \{ assume \( ?M e0 \) then obtain \( C0 \ es \ m \ ds \) where \( s2 : \text{MethodInvk} \ (\text{New } C0 \ es) \ m \ ds \in \text{subexprs}(e0) \) by auto
from rtrancl-trans[OF \( s2 s1 \)] have \( ?\text{case} \) by auto
\}

moreover \{ assume \( ?C e0 \) then obtain \( C0 \ D \ es \) where \( s2 : \text{Cast } D \ (\text{New } C0 \ es) \in \text{subexprs}(e0) \) by auto
from rtrancl-trans[OF \( s2 s1 \)] have \( ?\text{case} \) 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:isubexprs.intros)
\}

ultimately show \( ?\text{case} \) by blast

next
case \( 4 \ C0 \ CT \ Cs \ Ds \Gamma e0 \ es \ m \)

have \( s1 : e0 \in \text{subexprs}(\text{MethodInvk } e0 \ m \ es) \) by(auto simp add:isubexprs.intros)

36
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:isubexprs.intros)
} últimately show ?case by blast
next
case (5 C CT Cs Df Ds Γ es)
  hence
    length es = length Cs
    ∨ i, [i < length es; CT;Γ ⊢ (es!i) : (Cs!i); Γ = empty] =⇒ ?IH (es!i)
    and CT;Γ ⊢ + es : Cs
    by (auto simp add: typings-lengths)
  hence (∃ i < length es. (?F (es!i) ∨ ?M (es!i) ∨ ?C (es!i)) ∨ (vals(es)) (is ?Q es))
proof (induct es Cs rule:list-induct2)
case Nil thus ?Q [] by (auto intro:vals-val.intros)
next
case (Cons h t Ch Ct)
  with 5 have h-t-typs: CT;Γ ⊢ (h#t) : (Ch#Ct)
    and OIH: ∃ i. [i < length (h#t); CT;Γ ⊢ ((h#t)!i) : ((Ch#Ct)!i); Γ = empty] =⇒ ?IH ((h#t)!i)
    and G-def: Γ = empty
    by auto
  from h-t-typs have
    h-typ: CT;Γ ⊢ (h#t)0 : (Ch#Ct)0
    and t-typs: CT;Γ ⊢ + t : Ct
    by (auto elim:typings.cases)
  { fix i assume i < length t
    hence s-i: Suc i < length (h#t) by auto
      from OIH(OF s-i) have [i < length t; CT;Γ ⊢ (!i) : (Ct!i); Γ = empty]
      =⇒ ?IH (!i) by auto }
  with t-typs have ?Q t using Cons by auto

37
moreover \{ 
  assume \( \exists i < \text{length } t \). (\?F (t\!i) \lor \?M (t\!i) \lor \?C (t\!i)) 
  then obtain \( i \) 
    where \( i < \text{length } t \) 
    and \( \?F (t\!i) \lor \?M (t\!i) \lor \?C (t\!i) \) by force 
  hence \( (\?C ((\text{Suc } i) < \text{length } (\text{h\#t})) \land (\?F ((\text{h\#t})!(\text{Suc } i)) \lor \?M ((\text{h\#t})!(\text{Suc } i)) \lor \?C ((\text{h\#t})!(\text{Suc } i)))) \) by auto 
  hence \( \exists i < \text{length } (\text{h\#t}). (\?F ((\text{h\#t})!i) \lor \?M ((\text{h\#t})!i) \lor \?C ((\text{h\#t})!i)) \) 
\} 

hence \( \exists Q (\text{h\#t}) \) by auto 
\}

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

ultimately show \( \exists Q(\text{h\#t}) \) 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 \( i < \text{length } es \) and \( r: \?F (es\!i) \lor \?M (es\!i) \lor \?C(es\!i) \) by force 
  from \text{i\!th-mem}[\text{OF i-len}] have \( s1: \text{es\!i} \in \text{subexprs}(\text{New C es}) \) by (\text{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 \( C0 \) \( D \) \( es\!' D' \) where \( s2: \text{Cast} D (\text{New C0 es'}) \) \( es\!' D' \in \text{subexprs}(es\!i) \) 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 

38
hence \textit{case by auto}

\textbf{next}

\begin{itemize}
\item \textbf{case (6 C CT D Γ e0)}
\item \textbf{have s1: e0 ∈ subexprs(Cast C e0)} \textbf{by(auto simp add:subexprs.intros)}
\end{itemize}

from 6 have \textit{IH e0 by auto}

moreover

\begin{itemize}
\item \textbf{assume ?F e0}
\item then obtain C0 es fi where s2: FieldProj (New C0 es) fi ∈ subexprs(e0)
\end{itemize}

\begin{itemize}
\item \textbf{from rtrancl-trans[OF s2 s1] have \textit{case by auto}}
\end{itemize}

moreover

\begin{itemize}
\item \textbf{assume ?M e0}
\item then obtain C0 es m ds where s2: MethodInvk (New C0 es) m ds ∈ subexprs(e0)
\end{itemize}

\begin{itemize}
\item \textbf{from rtrancl-trans[OF s2 s1] have \textit{case by auto}}
\end{itemize}

moreover

\begin{itemize}
\item \textbf{assume ?C e0}
\item then obtain C0 D′ es where s2: Cast D′ (New C0 es) ∈ subexprs(e0)
\end{itemize}

\begin{itemize}
\item \textbf{from rtrancl-trans[OF s2 s1] have \textit{case by auto}}
\end{itemize}

moreover

\begin{itemize}
\item \textbf{assume ?V e0}
\item then obtain C0 es fi where s2: FieldProj (New C0 es) fi ∈ subexprs(e0)
\end{itemize}

\begin{itemize}
\item \textbf{from rtrancl-trans[OF s2 s1] have \textit{case by auto}}
\end{itemize}

moreover

\begin{itemize}
\item \textbf{assume ?F e0}
\item then obtain C0 es fi where s2: FieldProj (New C0 es) fi ∈ subexprs(e0)
\end{itemize}

\begin{itemize}
\item \textbf{from rtrancl-trans[OF s2 s1] have \textit{case by auto}}
\end{itemize}

moreover

\begin{itemize}
\item \textbf{assume ?M e0}
\item then obtain C0 es m ds where s2: MethodInvk (New C0 es) m ds ∈ subexprs(e0)
\end{itemize}

\begin{itemize}
\item \textbf{from rtrancl-trans[OF s2 s1] have \textit{case by auto}}
\end{itemize}

moreover

\begin{itemize}
\item \textbf{assume ?C e0}
\item then obtain C0 D′ es where s2: Cast D′ (New C0 es) ∈ subexprs(e0)
\end{itemize}

\begin{itemize}
\item \textbf{from rtrancl-trans[OF s2 s1] have \textit{case by auto}}
\end{itemize}

moreover

\begin{itemize}
\item \textbf{assume ?V e0}
\item then obtain C0 es fi where s2: FieldProj (New C0 es) fi ∈ subexprs(e0)
\end{itemize}

\begin{itemize}
\item \textbf{from rtrancl-trans[OF s2 s1] have \textit{case by auto}}
\end{itemize}

moreover

\begin{itemize}
\item \textbf{assume ?C e0}
\item then obtain C0 D′ es where s2: Cast D′ (New C0 es) ∈ subexprs(e0)
\end{itemize}

\begin{itemize}
\item \textbf{from rtrancl-trans[OF s2 s1] have \textit{case by auto}}
\end{itemize}
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
  }
  moreover {
    assume ?M e0
    then obtain C0 es m ds where s2: MethodInvk (New C0 es) m ds ∈ subexprs(e0)
    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
  }
  moreover {
    assume ?V e0
    then obtain C0 es' where e0 = (New C0 es') and vals(es') by (force elim:val_cases)
    hence ?case by(force intro:isubexprs.intros)
  }
  ultimately show ?case by blast

qed

3.9 Type Soundness Theorem

theorem Thm-2-4-3:
  assumes e-typ: CT;empty ⊢ e : C
  and ct-ok: CT OK
  and multisteps: CT ⊢ e →* e1
  and no-step: ¬(∃ e2. CT ⊢ e1 → e2)
  shows (val(e1) ∧ (∃ D. CT;empty ⊢ e1 : D ∧ CT ⊢ D <: C))
  ∨ (∃ D C es. (Cast D (New C es) ∈ subexprs(e1) ∧ CT ⊢ C ⊲ D <: D))
proof –
  from assms Cor-2-4-1-multi[OF multisteps ct-ok e-typ] obtain C1
    where e1-typ: CT;empty ⊢ e1 : C1
    and C1-sub-C: CT ⊢ C1 <: C by auto
from e1-typ have \((\exists C0 es . \text{FieldProj (New C0 es) } f) \in \text{subexprs} (e1)\) \\
\lor (\exists C0 es m ds . \text{MethodInvk (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 \text{fp: FieldProj (New C0 es) } f \in \text{subexprs} (e1) by auto 
  
  then obtain Ci where CT;empty \vdash \text{FieldProj (New C0 es) } f : Ci using e1-typ by (force simp add: subexpr-typing) 
  
  then obtain C0' where new-typ: CT;empty \vdash \text{New C0 es} : C0' by (force elim: typing.cases) 
  
hence C0 = C0' by (auto elim: typing.cases) 
  
  with new-typ obtain Df where f1: fields (CT,C0) = Df and lens: length es = length Df by (auto elim: typing.cases) 
  
  from Thm-2-4-2-1[OF e1-typ fp] obtain Cf fDef 
  
  where f2: fields (CT,C0) = Cf 
  and \text{lookup2} Cf (\lambda fd. vdName fd = f) = \text{Some} (\text{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 \text{lkup}] obtain i where 
  
i < \text{length} Cf 
  and fDef = Cf ! i 
  and (length Cf = length es) \rightarrow \text{lookup2} Cf es (\lambda fd. vdName fd = f) = \text{Some} (\text{es}!i) by auto 
  
  ultimately have \text{lookup2} Cf es (\lambda fd. vdName fd = f) = \text{Some} (\text{es}!i) by auto 
  
  with f2 have CT \vdash \text{FieldProj (New C0 es) } f \to (\text{es}!i) by (auto intro: reduction.intros) 
  
  with \text{fp have} \exists e2. CT \vdash e1 \to e2 by (simp add: subexpr-reduct) 
  
  with no-step have \text{thesis} by auto 
  \} moreover \{ 
  assume ?M e1 
  
  then obtain C0 es m ds where \text{mi: MethodInvk (New C0 es) } m ds \in \text{subexprs} (e1) by auto 
  
  then obtain D where CT;empty \vdash \text{MethodInvk (New C0 es) } m ds : D using e1-typ by (force simp add: subexpr-typing) 
  
  then obtain C0' Es E 
  
  where m-typ: CT;empty \vdash \text{New C0 es} : C0' 
  and \text{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 \text{mb: mbody} (CT, m, C0) = xs . e0 and length xs = length ds by auto 
  
  hence CT \vdash (\text{MethodInvk (New C0 es) } m ds) \to (\text{substs}[xs]\to\text{ds},\text{this} \to (\text{New C0 es})e0) by (auto simp add: reduction.intros) 
  
  with \text{mi have} \exists e2. CT \vdash e1 \to e2 by (simp add: subexpr-reduct) 
  
  with no-step have \text{thesis} by auto 
  \} moreover \{ 
  assume ?C e1 
  
  then obtain C0 D es where c-def: \text{Cast D (New C0 es)} \in \text{subexprs} (e1) by
auto

then obtain $D'$ where $CT;empty \vdash Cast D (New C0 es) : D'$ using $e1$-typ
by (force simp add:subexpr-typing)

then obtain $C0'$ where new-typ: $CT;empty \vdash New C0 es : C0'$ and $D$-eq-$D'$:
$D = D'$ by (auto elim:typing.cases)

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

hence $\textit{thesis}$ proof (cases $CT \vdash C0 <: D$)

  case True
  hence $CT \vdash Cast D (New C0 es) \rightarrow (New C0 es)$ by (auto simp add:reduction.intros)
  with $c$-def have $\exists e2. CT \vdash e1 \rightarrow e2$ by (simp add:subexpr-reduct)
  with no-step show $\textit{thesis}$ by auto

next

  case False
  with $c$-def show $\textit{thesis}$ by auto

qed

} moreover {

assume $\forall e1$

hence $\textit{thesis}$ using $\textit{assms}$ by (auto simp add:Cor-2-4-1-multi)

} ultimately show $\textit{thesis}$ by blast

qed

end

theory Execute
imports FJSound
begin

4 Executing FeatherweightJava programs

We execute FeatherweightJava programs using the predicate compiler.

code-pred (modes: $i ::= i \Rightarrow i \Rightarrow i \Rightarrow \textit{bool}$,

$i ::= i \Rightarrow o \Rightarrow \textit{bool}$ as supertypes-of) subtyping.

thm subtyping.equation

The reduction relation requires that we inverse the $op \circ$ 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 $xs$ $ys$ $zs$ $\Rightarrow$ 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} \texttt{rc-invk-arg'}: \(CT \vdash e_i \rightarrow e_i' \implies \text{append el (} e_i \# e_r \text{) } e' \implies \text{append el (} e_i' \# e_r \text{)} e'' \implies CT \vdash \text{MethodInvk } e \ m \ e' \rightarrow \text{MethodInvk } e \ m \ e''\)
\textit{unfolding append-def by simp (rule reduction.intros(6))}

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

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

\texttt{code-pred} (modes: \(i \Rightarrow i \Rightarrow i \Rightarrow \text{bool, } i \Rightarrow i \Rightarrow o \Rightarrow \text{bool as reduce})
\texttt{reduction}
\texttt{proof –}
\texttt{ case append}
\texttt{ from this show thesis}
\textit{ unfolding append-def by (cases xa) fastforce+}
\texttt{next}
\texttt{ case reduction}
\texttt{ from reduction.prems show thesis}
\texttt{ proof (cases rule: reduction.cases)
\texttt{ case r-field}}
\textit{ with reduction(1) show thesis by fastforce}
\texttt{next}
\texttt{ case r-inv}
\textit{ with reduction(2) show thesis by fastforce}
\texttt{next}
\texttt{ case r-cast}
\textit{ with reduction(3) show thesis by fastforce}
\texttt{next}
\texttt{ case re-field}
\textit{ with reduction(4) show thesis by fastforce}
\texttt{next}
\texttt{ case re-invk-recv}
\textit{ with reduction(5) show thesis by fastforce}
\texttt{next}
\texttt{ case re-invk-arg}
\textit{ with reduction(6) show thesis}
\textit{ unfolding append-def by fastforce}
\texttt{next}
\texttt{ case re-new-arg}
\textit{ with reduction(7) show thesis}
\textit{ unfolding append-def by fastforce}
\texttt{next}
\texttt{ case re-cast}
\textit{ with reduction(8) show thesis by fastforce}
\texttt{qed}
We also make the class typing executable: this requires that we derive rules for `method-typing`.

**definition** `method-typing-aux` where

\[
\text{method-typing-aux } CT \ m \ D \ Cs \ C = (\neg (\forall Ds \ D0. \ \text{mtype}(CT,m,D) = Ds \to D0 \to Cs = Ds \land C = D0))
\]

**lemma** `method-typing-aux`:

\[
(\forall Ds \ D0. \ \text{mtype}(CT,m,D) = Ds \to D0 \to Cs = Ds \land C = D0) = (\neg \text{method-typing-aux } CT \ m \ D \ Cs \ C)
\]

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

**lemma** `[code-pred-intro]`:

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

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

**lemma** `[code-pred-intro]`:

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

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

**declare** `method-typing.intros`[unfolded `method-typing-aux`, `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]]))

45
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 A [], New B []], New A []]) ffst) fsnd) fsnd →∗ x}

end

theory Featherweight-Java
imports FJSound Execute
begin

end

References
