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 \texttt{nat}s. We use the finite maps defined in \texttt{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 (\texttt{Object} and \texttt{this}).

1.1.1 Type definitions

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

type-synonym varCtx = varName → className

1.1.2 Constants

definition
  Object :: className where
  Object = 0

definition
  this :: varName where
  this == 0

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

1.1.4 Methods

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

1.1.5 Constructors

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

1.1.6 Classes

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

type-synonym classTable = className → classDef

1.2 Sub-expression Relation

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

\[
\text{inductive-set } \quad \text{isubexprs :: } (\text{exp} \ast \text{exp}) \set \quad \text{and } \text{isubexprs'} :: [\text{exp},\text{exp}] \Rightarrow \text{bool } (\cdot \in \text{isubexprs'}(\cdot)) [80,80] 80
\]

where

\[
e' \in \text{isubexprs}(e) \equiv (e',e) \in \text{isubexprs} \\
\text{se-field} : e \in \text{isubexprs}(\text{FieldProj } e \text{ fi}) \\
\text{se-invkrecv} : e \in \text{isubexprs}(\text{MethodInvk } e \text{ m es}) \\
\text{se-invkarg} : [\{ e_i \in \text{set es} \}] \Rightarrow e_i \in \text{isubexprs}(\text{New C es}) \\
\text{se-cast} : e \in \text{isubexprs}(\text{Cast C e})
\]

abbreviation

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

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

1.3 Values

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

\[
\text{inductive } \quad \text{vals :: } [\text{exp list}] \Rightarrow \text{bool } (\text{vals'}(\cdot)) [80] 80 \\
\text{and } \text{val :: } [\text{exp}] \Rightarrow \text{bool } (\text{val'}(\cdot)) [80] 80
\]

where

\[
\text{vals-nil} : \text{vals}([\text{ }]) \\
\text{vals-cons} : [\text{ val(vh); val(vt) }] \Rightarrow \text{vals}((\text{vh} \# \text{ vt})) \\
\text{val} : [\text{ val(vs) }] \Rightarrow \text{val}(\text{New C vs})
\]

1.4 Substitution

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

\[
\text{primrec } \text{substs :: } (\text{varName} \Rightarrow \text{exp}) \Rightarrow \text{exp} \Rightarrow \text{exp} \\
\text{and } \text{subst-list1 :: } (\text{varName} \Rightarrow \text{exp}) \Rightarrow \text{exp list} \Rightarrow \text{exp list} \\
\text{and } \text{subst-list2 :: } (\text{varName} \Rightarrow \text{exp}) \Rightarrow \text{exp list} \Rightarrow \text{exp list} \text{ where}
\]

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

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

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

1.5 Lookup  
The function lookup $f$ f function returns an option containing the first element of $l$ satisfying $f$, or None if no such element exists  
primrec lookup :: 'a list ⇒ ('a ⇒ bool) ⇒ 'a option  
where  
lookup [] $P$ = None  
| lookup (h#t) $P$ = (if $P$ h then Some h else lookup t $P$)  

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

1.6 Variable Definition Accessors  
This section contains several helper functions for reading off the names and types of variable definitions (e.g., in field and method parameter declarations).  
definition  
varDefs-names :: varDef list ⇒ varName list where  
varDefs-names = map vdName  
definition  
varDefs-types :: varDef list ⇒ className list where  
varDefs-types = map vdType  

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

inductive subtyping :: $[\text{classTable}, \text{className}, \text{className}] \Rightarrow \text{bool}$

where

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

abbreviation neg-subtyping :: $[\text{classTable}, \text{className}, \text{className}] \Rightarrow \text{bool}$

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

inductive subtypings :: $[\text{classTable}, \text{className list}, \text{className list}] \Rightarrow \text{bool}$

where

\[
\begin{align*}
ss-nil &: CT \vdash + [] <: [] \\
| ss-cons &: \left[ \begin{array}{l} CT \vdash C0 <: D0; \quad CT \vdash + Cs <: Ds \end{array} \right] \implies CT \vdash + (C0 \# Cs) <: (D0 \# Ds) \\
\end{align*}
\]

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 :: $[\text{classTable}, \text{className}, \text{varDef list}] \Rightarrow \text{bool}$

where

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

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) (\lambda md.(mName md = m)) = Some(mDef);
| varDefs-types (mParams mDef) = Bs;
| mReturn mDef = B ]
\[ \quad \Rightarrow mtype(CT,m,C) = Bs → B
\]

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

1.10 mbbody Relation

The mtype relation, written mbbody(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.

\[\]

\[\]

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 ]
\[ \quad \Rightarrow mbbody(CT,m,C) = xs . e
\]

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

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.

\[\]

\[\]
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);
    vdType fDef = Ci ]
  ⇒ CT;Γ ⊢ FieldProj e0 fi : Ci

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

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

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

| t-dcast :
  [ CT;Γ ⊢ e0 : D;
    CT ⊢ C ≺ D; C \neq D ]
  ⇒ CT;Γ ⊢ Cast C e0 : C

| t-scast :
  [ CT;Γ ⊢ e0 : D;]
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** \textit{typing-induct:}

**Assumptions:** $CT; \Gamma \vdash e : C$ (is $\forall T$)

and $\forall C \ CT \ \Gamma \ x \ . \ C \leadsto P \ CT \ \Gamma \ (\mathit{Var} \ x) \ C$

and $\forall C0 \ CT \ CF \ Ci \ \Gamma \ e0 \ fDef \ fi. \ [CT; \Gamma \vdash e0 : C0; \ P \ CT \ \Gamma \ e0 \ C0; \ fields(CT,C0) = CF; \ lookup \ CF \ (\lambda fd. \ vdName \ fd = fi) = \ Some \ fDef; \ vdType \ fDef = Ci] \Longrightarrow P \ CT \ \Gamma \ (\mathit{FieldProj} \ e0 \ fi) \ Ci$

and $\forall C \ C0 \ CT \ Cs \ Ds \ \Gamma \ e0 \ es \ m. \ [CT; \Gamma \vdash e0 : C0; \ P \ CT \ \Gamma \ e0 \ C0; \ mtype(CT,m,C0) = Ds \rightarrow C; \ CT; \Gamma \vdash+ \ es : Cs; \ \Gamma \ i . \ i < \ length \ es \ ] \Longrightarrow P \ CT \ \Gamma \ (\mathit{es}!i) \ (Cs!i)$

and $\forall C \ C0 \ CT \ Cs \ Dj \ Ds \ \Gamma \ es. \ [fields(CT,C) = Dj; \ length \ es = length \ Dj; \ \mathit{varDefs-t}ypes \ Dj = Ds; \ CT;\Gamma \vdash+ \ es : Cs; \ \Gamma \ i . \ i < \ length \ es \ ] \Longrightarrow P \ CT \ \Gamma \ (\mathit{es}!i) \ (Cs!i); \ CT \vdash+ \ Cs <; \ Ds] \Longrightarrow P \ CT \ \Gamma \ (\mathit{New} \ C \ es) \ C$

and $\forall C \ C0 \ CT \ D \ e0. \ [CT;\Gamma \vdash e0 : D; \ P \ CT \ \Gamma \ e0 \ D; \ CT \vdash D <; C] \Longrightarrow P \ CT \ \Gamma \ (\mathit{Cast} \ C \ e0) \ C$

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

**Proof**

1. Fix $es$ $Cs$
2. Let $\forall IH = CT; \Gamma \vdash+ \ es : Cs \leadsto (\forall i < \ length \ es. \ P \ CT \ \Gamma \ (es!i) \ (Cs!i))$
3. Have $\forall IH \land (\forall T \rightarrow \forall P)$
4. Proof (induct rule: typings-typing.induct)
   - Case (ts-nil CT $\Gamma$) show $\forall case$ by auto
   - Next
   - Case (ts-cons CT $\Gamma$ $e0$ $C0$ $es$ $Cs$)
     - Show $\forall case$ proof
       - Fix $i$
       - Show $i < \ length \ (e0\#es) \leadsto P \ CT \ \Gamma \ ((e0\#es)!i) \ ((C0\#Cs)!i)$ using ts-cons
     - By (cases $i$, auto)
   - Qed
   - Next
   - Case t-var then show $\forall case$ using $\forall case$ by auto
   - Next
   - Case t-field then show $\forall case$ using $\forall case$ by auto
   - Next
   - Case t-invK then show $\forall case$ using $\forall case$ by auto
   - Next
   - Case t-new then show $\forall case$ using $\forall case$ by auto
   - Next
case t-ucast then show \( \text{case using assms by auto} \)
next
case t-dcast then show \( \text{case using assms by auto} \)
next
case t-scast then show \( \text{case using assms by auto} \)
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 method-typing :: \([\text{classTable}, \text{methodDef}, \text{className}] \Rightarrow \text{bool} \) (- \( \vdash \) - \text{OK IN} - [80,80,80] 80)**

**where**

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

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

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

**where**

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

\( CT \vdash [] \text{ OK IN } C \)

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

\( [ CT \vdash m \text{ OK IN } C ;
CT \vdash ms \text{ OK IN } C ] \]
\[
\Rightarrow CT \vdash (m \# ms) \text{ OK IN } C
\]
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.

**inductive**

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

**where**

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

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**

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

**where**

\[
\text{ct-all-ok}: [\begin{array}{l}
\forall C \text{ CDef}. \text{CT}(C) = \text{Some}(CDef) \longrightarrow (CT \vdash CDef \text{ OK}) \land (\text{cName CDef} = C)
\end{array}] 
\Rightarrow CT \text{ OK}
\]

1.15 Evaluation Relation

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

**inductive**

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

**where**

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

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

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

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

| rc-invk-recev: \\
[ CT \vdash e0 \to e0’ ] \\
\implies CT \vdash MethodInvk e0 m es \to MethodInvk e0’ m es

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

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

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

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

end

2 FJAux: Auxiliary Lemmas

theory FJAux imports FJDefs
begin

2.1 Non-FJ Lemmas

2.1.1 Lists

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

12
case Nil thus ?case by auto
next
case (Cons esh est)
{ assume esh = ei
  with Cons have ?case by blast
}
moreover {
  assume esh ≠ ei
  with Cons have ei ∈ set est by auto
  with Cons obtain el er where esh ≠ est = (esh≠el) @ (ei≠er) by auto
  hence ?case by blast }
ultimately show ?case by blast
qed

lemma ith-mem: \( \forall i. \; [i < \text{length es}] \rightarrow es!i \in \text{set es} \)
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 length xs = length ys
shows [xs|→|ys,x|→|y] = [(xs@[x]|→|(ys@[y])]
using assms
by (induct xs ys rule:list-induct2) (auto simp add:map-upds-append1)

lemma map-upds-index:
assumes length xs = length As
and [xs|→|As|x = Some Ai]
shows \( \exists i. (As!i = Ai) \)
∧ (i < length As)
∧ (\forall Bs::('c list).((length Bs = length As)
\rightarrow ([xs|→|Bs] x = Some (Bs !i))))
(is \( \exists i. \; ?P i \; x \; As \))
is \( \exists i. (?P1 i \; As) \land (?P2 : As) \land (\forall Bs::('c list).(?P3 \; i \; xs \; As \; Bs)) \)
using assms
proof (induct xs As rule:list-induct2)
assume \( \[(x|→|y)]\ x = Some Ai \)
moreover have \( \neg[[]|→|[]] \ x = Some Ai \) by auto
ultimately show \( \exists i. \; ?P \; i \; [] \) by contradiction
next
fix xa xs y ys
assume length-xs-ys: length xs = length ys
and IH: [xs |→| ys] x = Some Ai =⇒ \( \exists i. \; ?P \; i \; xs \)
and map-eq-Some: [xa ≠ xs |→| y ≠ y] x = Some Ai
then have map-decomp: [xa≠xs |→| y#ys] = [xa|→|y] ++ [xs|→|ys] by fastforce
show \( \exists i. \; ?P \; i \; (xa≠xs) \) (y ≠ ys)
proof\((\text{cases } [xs \mapsto] ys] x)\)
\begin{align*}
\text{case} (\text{Some } Ai') \\
\text{hence } ([xa \mapsto y] ++ [xs \mapsto] ys]) x = \text{Some } Ai' \text{ by (rule map-add-find-right)}
\end{align*}

\text{hence } P: [xs \mapsto] ys] x = \text{Some } Ai \text{ using map-eq-Some Some by simp}

\text{from } \text{IH}[OF P] \text{ obtain } i \text{ where}
\begin{align*}
R1: y! i = A!
\text{and } R2: i < \text{length } ys
\text{and } pre-r3: \forall (Bs::'c list). \ ?P \ i \ xs \ ys \ Bs \text{ by fastforce}
\end{align*}

\{ fix Bs::'c list
\begin{align*}
\text{assume } \text{length-Bs: length } Bs = \text{length } (y#ys)
\text{then obtain } n \text{ where length } (y#ys) = \text{Suc } n \text{ by auto}
\end{align*}

\text{with } length-Bs \text{ obtain } b \ bs \text{ where Bs-def: Bs = } b#bs \text{ by (auto simp add:length-Suc-conv)}

\text{with } length-Bs \text{ have length } ys = \text{length } bs \text{ by simp}
\text{with } pre-r3 \text{ have } ([xa\mapsto-b] ++ [xs\mapsto] bs] x = \text{Some } (bs!1) \text{ by (auto simp only:map-add-find-right)}

\text{with } pre-r3 \text{ Bs-def length-Bs have } ?P3 (i+1) (xa#xs) (y#ys) Bs \text{ by simp}
\}

\text{with } R1 R2 \text{ have } ?P (i+1) (xa#xs) (y#ys) \text{ by auto}
\text{thus } ?thesis ..
\text{next}
\text{case None}
\text{with } map-decomp map-eq-Some \text{ have } [xa\mapsto] x = \text{Some } Ai \text{ by (auto simp only:map-add-SomeD)}
\text{hence ai-def: y = A! and } x\text{-eq-xa : } x = xa \text{ by (auto simp only:map-upd-Some-unfold)}

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

\subsection*{2.2 FJ Lemmas}

\subsection*{2.2.1 Substitution}

\textbf{lemma} subst-list1-eq-map-substs :
\begin{align*}
\forall \sigma. \text{subst-list1 } \sigma \ l = \text{map } (\text{subs ts } \sigma) \ l
\end{align*}
by \((\text{induct } l, \text{simp-all})\)

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

### 2.2.2 Lookup

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

**Lemma lookup-true**:
\[
\text{lookup } l \ f = \text{Some } r \Longrightarrow f \ r
\]
proof\((\text{induct } l)\)
- case Nil thus \(?\text{case by simp}\)
next
- case (Cons \(h\ t\)) thus \(?\text{case by } (\text{cases } f \ h)\) (auto simp add:lookup.simps)
  qed

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

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

**Lemma lookup-index**:
- assumes \(\text{lookup } l1 \ f = \text{Some } e\)
- shows \(\forall l2. \exists i < (\text{length } l2). e = l1\!\!i \land ((\text{length } l1 = \text{length } l2) \Longrightarrow \text{lookup2 } l1 \ l2 \ f = \text{Some } (l2\!\!i))\)
  using assms
proof\((\text{induct } l1)\)
- case Nil thus \(?\text{case by } auto\)
  qed

next
- case (Cons \(h1 \ t1\))
  \{ assume asm:\(f \ h1\)
  hence \(0 < \text{length } (h1 \# t1) \land e = (h1 \# t1)!0\)
  using Cons by \((\text{auto simp add:lookup.simps})\)
  moreover \{
  assume length: \(h1 \# t1) = \text{length } l2\)
  hence \(l2 = \text{Suc } (\text{length } t1)\) by auto
  then obtain \(h2 \ t2\) where l2-def: \(l2 = h2\#t2\) by \((\text{auto simp add: length-Suc-conv})\)
  hence \(\text{lookup2 } (h1 \# t1) \ l2 \ f = \text{Some } (l2\!\!0)\) using \(\text{asm by } \text{auto simp add: lookup2.simps})\)
\}
  ultimately have \(?\text{case by } auto\)
moreover 
  assume asm:\ - (f h1) 
  hence lookup \rt \ f = Some e 
    using Cons by (auto simp add: lookup.simps) 
then obtain i where 
  i<length \rt 
  and e = \rt ! i 
  and ih:(length \rt = length (\rt \tl) \implies lookup2 \rt \tl f = Some (\rt ! i)) 
    using Cons by blast 
  hence Suc i < length (h1#\rt) \land e = (h1#\rt)!(Suc i) using Cons by auto 
moreover 
  assume length (h1 # \rt) = length \tl by auto 
then obtain h2 \tl where \tl-def:\tl = h2#\tl by (auto simp add: length-Suc-conv) 
  hence lookup2 \rt \tl f = Some (\tl ! i) using ih \tl-def lens by auto 
  hence lookup2 \h1 # \rt \tl f = Some ((\tl!Suc i))  
    using asm \tl-def by(auto simp add: lookup2.simps) 
  } 
ultimately have \?case by auto 
} 
ultimately show \?case by auto 

done

lemma lookup2-index: 
  \\forall \tl. [ lookup2 \sl \tl f = Some e; 
  length \sl = length \tl ] \implies \exists i < (length \tl). e = (\tl!i) \land lookup \sl f = Some ((\tl!i)) 
proof (induct \sl) 
  case Nil thus \?case by auto 
next 
  case (Cons h1 \rt) 
  hence length \rt = Suc (length \rt) by auto 
then obtain h2 \tl where \tl-def:\tl = h2#\tl by (auto simp add: length-Suc-conv) 
  \{ assume asm:\ - h1 
  hence e = h2 using Cons \tl-def by (auto simp add:lookup2.simps) 
  hence 0<length (h2#\tl) \land e = (h2#\tl) ! 0 \land lookup (h1 # \rt) f = Some ((h1 # \rl) ! 0) 
    using asm by (auto simp add: lookup.simps) 
  hence \?case using \tl-def by auto 
  \} 
moreover 
  \{ assume asm:\ - (f h1) 
  hence \exists i<length \tl. e = \tl ! i \land lookup \rt f = Some (\rt ! i) using Cons \tl-def by auto 
  then obtain i where \i<length \tl \land e = \tl ! i \land lookup \rt f = Some (\rt ! i) 
    by auto 
  hence (Suc i) < length(h2#\tl) \land e = ((h2#\tl) ! (Suc i)) \land lookup (h1#\tl) f = Some ((h1#\rl) ! (Suc i)) 
    using asm by (force simp add: lookup.simps) 
  \}
hence ?case using l2-def by auto
}
ultimately show ?case by auto
qed

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

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

2.2.3 Functional

These lemmas prove that several relations are actually functions

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

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

lemma fields-functional:
assumes fields(CT,C) = Cf
and CT OK
shows ⋀ Cf'. [ fields(CT,C) = Cf ] ⇒ Cf = Cf'
using assms
proof induct
case (f-obj CT)
hence CT(Object) = None by (auto elim: ct-typing.cases)
thus ?case using f-obj by (auto elim: fields.cases)
next
case (f-class CT C CDef D Cf Dg Cf Dg Cf')
hence f-class-inv:
\( (CT \ C = \text{Some CDef}) \land (cSuper \ CDef = D) \land (cFields \ CDef = Cf) \)

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

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

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

then obtain \( Dg' \) where

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

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

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

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

thus \( ?\text{case using} \ (DgCf = Dg \circ Cf) \)

and \( \langle DgCf' = Dg' \circ Cf \rangle \) by \( \text{force} \)

qed

2.2.4 Subtyping and Typing

**lemma typings-lengths:** assumes \( CT;\Gamma \vdash \) es Cs shows \( \text{length es} = \text{length Cs} \)

using \( \text{assms} \) by \( \text{induct es Cs \ rule: list-induct2} \)

\( \text{lemma typings-index:} \)

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

proof −

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

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

using \( \text{assms} \)

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

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

next

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

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

qed

qed

\( \text{lemma subtypings-index:} \)

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

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

using \( \text{assms} \)

proof \( \text{induct} \)

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

next

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

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

qed

\( \text{lemma subtyping-append:} \)

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

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

shows \( CT \vdash (Cs@[C]) <: (Ds@[D]) \)

using \( \text{assms} \)
by (induct rule: subtypings.induct) (auto simp add: subtypings.intros elim: subtypings.cases)

lemma typings-append:
  assumes CT;Γ ⊢+ es : Cs
  and CT;Γ ⊢ e : C
  shows CT;Γ ⊢+ (es @ [e]) : (Cs @ [C])
proof −
  have length es = length Cs using assms by(simp-all add: typings-lengths)
  thus CT;Γ ⊢+ (es @ [e]) : (Cs @ [C]) using assms
  proof (induct es Cs rule: list-induct2)
    have CT;Γ ⊢ ([] : Cs) 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 ⊢ x : 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 # xxs) @ [e]) : ((y # ys) @ [C]) by (auto simp add: typings-typing, ts-cons)
  qed
  qed

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

lemma ith-subtyping: \( \forall Ds. \ [ CT ⊢+ (Cs ![h # t]) <: Ds ] \implies CT ⊢ h <: (Ds ![length Cs]) \)
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: \( \forall Ds Es. \ [ CT ⊢+ Cs <: Ds; CT ⊢+ Ds <: Es ] \implies CT ⊢+ Cs <: Es \)
proof (induct Cs)
  case Nil thus ?case
    by (auto elim: subtypings.cases simp add: subtypings.ss-nil)
next
  case (Cons hCs tCs)

19
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:
\[
\forall Cs. \ (CT;\Gamma \vdash+ (es@@h\#t)) : Cs; \\
CT;\Gamma \vdash h' : Ci'; \\
CT \vdash Ci' <: (Cs!(\text{length es})) \] \\
\Rightarrow \exists Cs'. \ (CT;\Gamma \vdash+ (es@@h'\#t)) : Cs' \land CT \vdash+ Cs' <: 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 \(CT \vdash+ Ci' <: hCs\) by auto

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

moreover from \(ts\) Nil have \(CT;\Gamma \vdash+ (Ci'\#tCs) : (Cs'\#tCs)\) 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 \(CT \vdash+ tCs' <: tCs\)
by auto

ultimately have
\(CT;\Gamma \vdash+ (ch@@he@@h'@@t) : (hCs\#tCs')\)
and \(CT \vdash+ (hCs\#tCs') <: Cs\)
by(auto simp add:typings-typing.ts-cons subtypings.ss-cons subtyping.s-refl)

thus ?case by auto

qed

lemma memtypings:
\[
\forall Cs. \ (CT;\Gamma \vdash es:Cs; \ ei \in set es] \Rightarrow \exists Ci. \ CT;\Gamma \vdash ei:Ci
\]

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 ⊢ ds!i : As!i and CT ⊢ As!i <: Bs!i
  using assms by (auto simp add: typings-index subtypings-index)

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

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

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

thus False using \texttt{s-super} by auto

qed

thus \texttt{?case} using \texttt{s-super} by (auto simp add: not-subtypes-aux)

qed

2.2.5 Sub-Expressions

\textbf{lemma} isubexpr-typing:
\hspace{1em} \textbf{assumes} e1 $\in$ isubexprs(e0)
\hspace{1em} \textbf{shows} $\forall C. [ CT; Map.empty \vdash e0 : C ] \implies \exists D. CT; Map.empty \vdash e1 : D$
\hspace{1em} \textbf{using} assms
\hspace{1em} \textbf{by} (induct rule:isubexprs.induct) (auto elim:typing.cases simp add:mem-typings)

\textbf{lemma} subexpr-typing:
\hspace{1em} \textbf{assumes} e1 $\in$ subexprs(e0)
\hspace{1em} \textbf{shows} $\forall C. [ CT; Map.empty \vdash e0 : C ] \implies \exists D. CT; Map.empty \vdash e1 : D$
\hspace{1em} \textbf{using} assms
\hspace{1em} \textbf{by} (induct rule:rtrancl.induct) (auto, force simp add:isubexpr-typing)

\textbf{lemma} isubexpr-reduct:
\hspace{1em} \textbf{assumes} d1 $\in$ isubexprs(e1)
\hspace{1em} \textbf{shows} $\forall d2. [ CT \vdash d1 \rightarrow d2 ] \implies \exists e2. CT \vdash e1 \rightarrow e2$
\hspace{1em} \textbf{using} assms mem-ith
\hspace{1em} \textbf{by} induct
\hspace{1em} (auto elim:isubexprs.cases intro:reduction.intros,
\hspace{2em} force intro:reduction.intros,
\hspace{2em} force intro:reduction.intros)

\textbf{lemma} subexpr-reduct:
\hspace{1em} \textbf{assumes} d1 $\in$ subexprs(e1)
\hspace{1em} \textbf{shows} $\forall d2. [ CT \vdash d1 \rightarrow d2 ] \implies \exists e2. CT \vdash e1 \rightarrow e2$
\hspace{1em} \textbf{using} assms
\hspace{1em} \textbf{by} (induct rule:rtrancl.induct) (auto, force simp add:isubexpr-reduct)

end

3 FJSound: Type Soundness

\textbf{theory} FJSound imports FJAux
\textbf{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

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

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

3.2 Method Types and Field Declarations of Subtypes

lemma $\text{A-1-1}$:
assumes $CT \vdash C :: D$ and $CT \text{ OK}$
shows $(\text{mt}(CT,m,D) = Cs \rightarrow C0) \implies (\text{mt}(CT,m,C) = Cs \rightarrow C0)$
using $\text{assms}$
proof (induct rule: $\text{subtyping}.\text{induct}$)
  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 $CT C (?Def D)$)
  hence $CT \vdash C ?Def \text{ OK } \text{ and } \text{ cName } C ?Def = C$
    by (auto elim:$\text{ct-typing}.\text{cases}$)
  with s-super obtain $M$
    where $M: CT \vdash M \text{ OK } IN C$ and $\text{ cMethods: cMethods } C ?Def = M$
    by (auto elim:$\text{class-typing}.\text{cases}$)
  let $?lookup-m = \text{lookup } M (\lambda md. (\text{mName } md = m))$
  show $?case$
    proof (cases $\exists m ?Def. \ ?lookup-m = \text{Some } m ?Def$)
      case True
then obtain \( m \text{Def} \) where \( m = \text{lookup}\_m = \text{Some} \ m \text{Def} \) by (rule \( \text{exE} \))

hence \( m \text{Def-name} = \text{Name} m \text{Def} = m \) by (rule \( \text{lookup-true} \))

have \( CT \vdash m \text{Def} \ OK \ IN \ C \) using \( M \ m \) by (auto simp add: method-typings-lookup)

then obtain \( C \text{Def}' m' D' Cs' C0' \)

where \( CT: CT \ C = \text{Some} \ C \text{Def}' \)
and \( c\text{Super} \ C \text{Def}' = D' \)
and \( \text{Name} m \text{Def} = m' \)
and \( m\text{Return}: m\text{Return} \text{Def} = C0' \)
and \( \text{varDefs-types}: \text{varDefs-types} (m\text{Params} m \text{Def}) = Cs' \)
and \( \forall Ds D0. (m\text{type}(CT, m', D') = Ds \to D0) \to Cs' = Ds \land C0' = D0 \)
by (auto elim: method-typing. cases)

with \( s\text{-super} \ m \text{Def-name} \) have \( C \text{Def} = C \text{Def}' \)
and \( D = D' \)
and \( m = m' \)
and \( \forall Ds D0. (m\text{type}(CT, m, D) = Ds \to D0) \to Cs' = Ds \land C0' = D0 \)
by auto

thus \( \text{thesis} \) using \( s\text{-super} \ c\text{Methods} \ m \text{CT} \ m\text{Return} \) \( \text{varDefs-types} \) by (auto intro: mtype. intros)

next
case False

hence \( \text{lookup} = \text{None} \) by (simp add: lookup-split)

then show \( \text{thesis} \) using \( s\text{-super} \ c\text{Methods} \) by (auto simp add: mtype. intros)

qed

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

3.3 Substitution Lemma

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

proof –

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

have RESULT: $(\; ?\text{TYPINGS} \; \rightarrow \; ?\text{COMMON-ASMS} \; \rightarrow \; ?P1)$
\land $(\; ?\text{TYPING} \; \rightarrow \; ?\text{COMMON-ASMS} \; \rightarrow \; ?P2)$

proof (induct rule: typings-typing.induct)

case (ts-nil CT $\Gamma$)

show $\forall$case

proof (rule impI)

have $(CT;\Gamma_1 \vdash \vdash ([ds/xs][[]][]):[] \land (CT \vdash \vdash [] <: []))$
by (auto simp add: typings-typing.intros subtypings-intros)

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

qed

next

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

show $\exists$case

proof (rule impI)

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

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

with ts-cons asms have

$\exists C.(CT;\Gamma_1 \vdash \vdash (ds/xs) e0 : C) \land (CT \vdash C <: C0)$

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

then obtain $C \; Cs$ where

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

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

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

and $CT \vdash \vdash (C#Cs) <: (C0#Cs')$ by (auto simp add: typings-typing.intros subtypings-intros)

then show $\exists Cs. \; CT;\Gamma_1 \vdash \vdash map \; (\text{substs} \; [xs \mapsto] \; \text{ds}) \; (e0 \; \# \; es) : Cs \land CT \vdash \vdash Cs <: (C0 \; \# \; Cs')$
by auto

qed

next

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

show $\exists$case

proof (rule impI)

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


hence

\[ \text{lengths: length } ds = \text{length } Bs \]

and \( G\text{-def}: \Gamma = \Gamma 1 \vdash \Gamma 2 \)
and \( G2\text{-def}: \Gamma 2 = [xs \mapsto] Bs \) by auto

from lengths \( G2\text{-def} \) have same-dom\( s \): \( \text{dom}([xs \mapsto] ds) = \text{dom}(\Gamma 2) \) by auto
from asms show \( \exists C. \ CT ; \Gamma I \vdash \text{subs} [xs \mapsto] ds \) \( \text{(Var } x) : C \land CT \vdash C \)
\(<: C' \)

proof (cases \( \Gamma 2 \ x) \)

case None

with \( G\text{-def } \ t\text{-var have } G1\text{-x: } \Gamma I x = \text{Some } C' \) by (simp add: map-add(Some-iiff))
from None same-dom\( s \) have \( x \notin \text{dom}([xs \mapsto] ds) \) by (auto simp only:domIff)

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

hence \( \text{ds} /xs\) \( \text{(Var } x) = \text{(Var } x) \) by auto

with \( G1\text{-x} \) have

\( CT ; \Gamma I \vdash (\text{ds} /xs)(\text{Var } x) : C' \land CT \vdash C' <: C' \)

by (auto simp add: typings-typing.intro subtype.intros)

thus \( \text{?thesis} \) by auto

next

case (\( \text{Some } Bi) \)

with \( G\text{-def } \ t\text{-var have } c'\text{-eq-bi: } C' = Bi \) by (auto simp add: map-add(SomeD))
from lengths \( x = \text{length } ds \) have length \( x = \text{length } Bs \) by simp

with \( \text{Some } G2\text{-def have } \exists i. (Bs!i = Bi) \land (i < \text{length } Bs) \land \)

\( \forall l.((\text{length } l = \text{length } Bs) \imp ([xs \mapsto] l \ x = \text{Some } (i!i))) \)

by (auto simp add: map-upds-index)

then obtain \( i \) where \( bs\text{-i-proj: } (Bs!i = Bi) \)

and \( i\text{-len: } i < \text{length } Bs \)

and \( \text{P: } (\forall l. \text{exp list}.((\text{length } l = \text{length } Bs) \imp ([xs \mapsto] l \ x = \text{Some } (i!i)))) \)

by fastforce

from lengths \( P \) have \( \text{subst-x: } ([xs \mapsto] ds \ x = \text{Some } (ds!i)) \) by auto
from asms obtain \( As \) where \( \text{as-ex: } CT ; \Gamma I \vdash \text{ds} : As \land CT \vdash As!i <: Bs \)

by fastforce

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

hence \( \text{proj-i: } CT ; \Gamma I \vdash ds!i : As!i \land CT \vdash As!i <: Bs!i \)

using \( i\text{-len } \text{lengths } as\text{-ex} \) by (auto simp add: typings-proj)

hence \( CT ; \Gamma I \vdash (\text{ds} /xs)(\text{Var } x) : As!i \land CT \vdash As!i <: C' \)

using \( c'\text{-eq-bi } bs\text{-i-proj } \text{subst-x} \) by auto

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

qed

next

case (\( t\text{-field } CT \Gamma \ c0 \ C0 \ Cf \) \( f\text{-Def } Ci) \)

show \( ?\text{case} \)

proof (rule impI)

assume asms: \( (CT \text{ OK}) \land (\Gamma = \Gamma I \vdash \Gamma 2) \land \)

\( (\Gamma 2 = [xs \mapsto] Bs) \land (\text{length } Bs = \text{length } ds) \land (\exists As. CT ; \Gamma I \vdash \text{ds} : As \land CT \vdash As <: Bs) \)

from \( t\text{-field } \text{have } \text{flds: } \text{fields}(CT,C0) = Cf \) by fastforce
from t-field asms obtain C where e0-typ: CT;Γ I ⊢ (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)

moreover have CT ⊢ C <: C0 by (simp add:subtyping.intros)
ultimately show ∃ C. CT;Γ I ⊢ (ds/xs)(FieldProj e0 fi) : C ∧ CT ⊢ C <: C by auto

next
case(t-invk CT Γ e0 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;Γ I ⊢+ ds : As ∧ CT ⊢+ As <: Bs)
hence ct-ok: CT OK ..

from t-invk have mtyp: mtype(CT,m,C0) = Ds → C
and subs: CT ⊢+ Cm <: Ds
and lens: length es = length Ds
by auto

from t-invk asms obtain C' where
e0-typ: CT;Γ I ⊢ (ds/xs)e0 : C' and sub': CT ⊢ C' <: C0 by auto

from t-invk asms obtain Cs' where
es-typ: CT;Γ I ⊢+ (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

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

moreover have CT ⊢ C <: C by(simp add:subtyping.intros)
ultimately show ∃ C'. CT;Γ I ⊢ (ds/xs)(MethodInvk e0 m es) : C' ∧ CT ⊢ C' <: C by auto

qed

next
case(t-new CT Df es Ds Γ Cs)

show ?case

27
proof (rule impI)
  assume asms: \((CT \text{ OK}) \land (\Gamma = \Gamma_1 \vdash \Gamma_2) \land (\Gamma_2 = [xs \mapsto B_2])\) \land (length 
  \text{ Bs} = \text{ length \ ds}) \land (\exists As. CT; \Gamma_1 \vdash + \ ds : As \land CT \vdash + As <: \text{ Bs})
  hence ct-ok: CT \text{ OK} ..
  from t-new have
    subs: CT \vdash + Cs <: \text{ Ds}
    and flds: fields(CT,C) = Df
    and len: length es = length Df
    and vlds: \text{ varDefs-types Df} = Ds
    by auto
  from t-new asms obtain Cs' where
    es-typ: CT; \Gamma_1 \vdash (\text{ ds/\text{ x}s})es : Cs' and subst: CT \vdash + Cs <: Cs by auto
    have subst-e: (\text{ ds/\text{ x}s})(\text{ New C es}) = \text{ New C} (\text{ (ds/\text{ x}s)es})
      by (auto simp add: subst-list2-eq-map-substs)
  have CT; \Gamma_1 \vdash (\text{ ds/\text{ x}s})(\text{ New C es}) : C by (auto simp add: typings-typing-intros)
  moreover have \text{ CT} \vdash C <: C by (simp add: subtyping-intros)
  ultimately show \(\exists C'. CT; \Gamma_1 \vdash (\text{ ds/\text{ x}s})(\text{ New C es}) : C' \land CT \vdash C' <: C\)
  by auto
qed

next
  case (t-ucast CT \Gamma e0 D C)
  show \(\gamma\) case
  proof (rule impI)
    assume asms: \((CT \text{ OK}) \land (\Gamma = \Gamma_1 \vdash \Gamma_2) \land (\Gamma_2 = [xs \mapsto B_2])\) \land (length 
    \text{ Bs} = \text{ length \ ds}) \land (\exists As. CT; \Gamma_1 \vdash + \ ds : As \land CT \vdash + As <: \text{ Bs})
    from t-ucast asms obtain C' where e0-typ: CT; \Gamma_1 \vdash (\text{ ds/\text{ x}s)e0 : C')
      and sub1: CT \vdash C' <: D
      and sub2: CT \vdash D <: C by auto
    from sub1 sub2 have CT \vdash C' <: C by (rule s-trans)
    with e0-typ have CT; \Gamma_1 \vdash (\text{ ds/\text{ x}s})(\text{ Cast C e0}) : C by (auto simp add: 
      typings-typing-intros)
    moreover have CT \vdash C <: C by (rule s-refl)
    ultimately show \(\exists C'. CT; \Gamma_1 \vdash (\text{ ds/\text{ x}s})(\text{ Cast C e0}) : C' \land CT \vdash C' <: C\)
    by auto
  qed
next
  case (t-dcast CT \Gamma e0 D C)
  show \(\gamma\) case
  proof (rule impI)
    assume asms: \((CT \text{ OK}) \land (\Gamma = \Gamma_1 \vdash \Gamma_2) \land (\Gamma_2 = [xs \mapsto B_2])\) \land (length 
      \text{ Bs} = \text{ length \ ds}) \land (\exists As. CT; \Gamma_1 \vdash + \ ds : As \land CT \vdash + As <: \text{ Bs})
    from t-dcast asms obtain C' where e0-typ: CT; \Gamma_1 \vdash (\text{ ds/\text{ x}s)e0 : C') by auto
      have (CT \vdash C' <: C) \lor
        (C \neq C' \land CT \vdash \text{ C <: C'}) \lor
        (CT \vdash C \hookrightarrow C' \land CT \vdash C' \hookrightarrow C) by blast
      moreover
      \{ assume CT \vdash C' <: C
      with e0-typ have CT; \Gamma_1 \vdash (\text{ ds/\text{ x}s})(\text{ Cast C e0}) : C by (auto simp add: 
        typings-typing-intros)
      \}
  qed

28
 typings-typing.intros
}

moreover
{ assume \((C \neq C' \land CT \vdash C <: C')\)
  with e0-typ have \(CT;\Gamma I \vdash (ds/xs)\) \((Cast C e0) : C\) by (auto simp add: typings-typing.intros)
}

moreover
{ assume \((CT \vdash C \nless; C' \land CT \vdash C' \nless; C)\)
  with e0-typ have \(CT;\Gamma I \vdash (ds/xs)\) \((Cast C e0) : C\) by (auto simp add: typings-typing.intros)
}

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

qed

next

case (t-scast CT \(\Gamma\) e0 D C)

show \(?case\)

proof (rule impI)

assume asms: \((CT OK) \land (\Gamma = \Gamma I ++ \Gamma 2) \land (\Gamma 2 = [xs \mapsto-> Bs]) \land (length Bs = length ds) \land (\exists As. CT;\Gamma 1 \vdash ds : As \land CT \vdash As <: Bs)\)

from t-scast asms obtain \(C'\) where e0-typ: \(CT;\Gamma I \vdash (ds/xs)e0 : C'\)
  and sub1: \(CT \vdash C' <: D\)
  and nsub1: \(CT \vdash C \nless;: D\)
  and nsub2: \(CT \vdash D \nless;: C\) by auto

from not-subtypes[OF sub1 nsub1 nsub2] have \(CT \vdash C' \nless;: C\) by fastforce

moreover have \(CT \vdash C \nless;: C'\)

proof (rule ccontr)

assume \(\neg CT \vdash C \nless;: C'\)

hence \(CT \vdash C <: C'\) by auto

hence \(CT \vdash C <: D\) using sub1 by (rule s-trans)

with nsub1 show False by auto

qed

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

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

qed

thus \(?TYPINGS \Longrightarrow ?P1\) and \(?TYPING \Longrightarrow ?P2\) 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 \((CT;\Gamma 2 \vdash+ es : Cs) \Longrightarrow (CT;\Gamma I++\Gamma 2 \vdash+ es : Cs)\) (is \(?P1 \Longrightarrow ?P2\)
A-1-4

3.5 Method Body Typing Lemma

Lemma A-1-4:
assumes ct-ok: $CT \vdash OK$
and mb:mbody$(CT,m,C) = xs . e$
and mt:mtype$(CT,m,C) = Ds \rightarrow D$
shows $\exists D0 C0. (CT \vdash C <: D0) \wedge$
$(CT \vdash C0 <: D) \wedge$
$(CT;[xs[\mapsto]Ds](this \mapsto D0) \vdash e : C0)$
using mb ct-ok mt proof(induct rule: mbbody.induct)
case (mb-class $CT \ C CDef m mDef \ xs \ e$

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

hence $CT \vdash (cMethods CDef) \ OK \ IN \ C$ by (auto elim: class-typing.cases)

hence $CT \vdash mDef OK \ IN \ C$ using mb-class by(auto simp add: method-typings-lookup)

hence $\exists E0. ((CT;[xs[\mapsto]Ds, this \mapsto C] \vdash e : E0) \wedge (CT \vdash E0 <: D))$
using mb-class m-param m-ret by(auto elim: method-typing.cases)

then obtain $E0$
where $CT;[xs[\mapsto]Ds, this \mapsto C] \vdash e : E0$
and $CT \vdash E0 <: D$
and $CT \vdash C <: C$ by (auto simp add: s-refl)

thus ?case by blast

next
case (mb-super $CT \ C CDef m Da \ xs \ e$

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

qed
3.6 Subject Reduction Theorem

**Theorem Thm-2-4-1:**

- Assumes $CT \vdash e \rightarrow e'$
- And $CT$ OK
- Shows $\forall C. [\forall C'. (CT; \Gamma \vdash e': C' \land CT \vdash C' \ll C)]$

**Using** assms

**Proof** (induct rule: reduction_induct)

- Case $(r$-field $CT \vdash Ca \ Cf \ es \ f i \ e')$
- Hence $CT; \Gamma \vdash FieldProj (New Ca es) Fi : C$
- And $ct-ok$: $CT$ OK
- And fields $\operatorname{fields}(CT; Ca) \Rightarrow Cf$
- And lookup $\operatorname{lookup2 Cf es} (\lambda fd. \operatorname{vdName fd = fi}) = \operatorname{Some} e'$ by fastforce+

**Then obtain** $Ca' Cf' fDef$

- Where $\operatorname{new-typ}: CT; \Gamma \vdash New Ca es : Ca'$
- And fields $\operatorname{fields}(CT; Ca') = Cf'$
- And lookup $\operatorname{lookup Cf'} (\lambda fd. \operatorname{vdName fd = fi}) = \operatorname{Some} fDef$
- And $C-def$: $\vdashType fDef = C$ by (auto elim: typing.cases)

**Hence** $Ca-Ca': Ca = Ca'$ by (auto elim:typing.cases)

- With fields $\operatorname{have}$ $Cf-Cf': Cf = Cf''$ by (auto simp add:fields-functional[OF fields ct-ok])

**From new-typ obtain** $Cs Ds Cf''$

- Where fields $\operatorname{fields}(CT; Ca') = Cf''$
- And es-typs: $CT; \Gamma \vdash es:Cs$
- And Ds-def: $\vdash\operatorname{varDefTypes} Cf'' = Ds$
- And length-Cf-es: $\operatorname{length Cf''} = \operatorname{length es}$
- And subs: $CT \vdash es <: Ds$

- By (auto elim:typing.cases)

- With $Ca-Ca'$ have $Cf-Cf''$, $Cf = Cf''$ by (auto simp add:fields-functional[OF fields ct-ok])

**From length-Cf-es Cf-Cf'' lookup index[OF lookup2] obtain** $i$ where

- I-bound: $i < \operatorname{length es}$
- And $e' = e A i$
- And lookup $\operatorname{lookup Cf} (\lambda fd. \operatorname{vdName fd = fi}) = \operatorname{Some} C(fA i)$ by auto

Moreover

- With $C-def Ds-def$ $\operatorname{lookup lookup2}$ have $Ds!i = C'$
- Using $Ca-Ca'$ $Cf-Cf'$ $Cf-Cf''$ i-bound length-Cf-es fields'

- By (auto simp add:mth-map varDefTypes-def fields-functional[OF fields ct-ok])

**Moreover with** subs es-typs have

- $CT; \Gamma \vdash (es!i);(Cs!i)$ and $CT \vdash (Cs!i) <: (Ds!i)$ using i-bound

- By (auto simp add:typings-index subtypings-index typings-lengths)

**Ultimately show** $\vdash case$ by auto

**Next**

- Case $(r$-invk $CT \vdash m Ca \ xs \ es \ e)'
- From $r$-invk have $mb: \lambda \operatorname{body}(CT,m, Ca) = xs . e$ by fastforce
- From $r$-invk obtain $Ca' Ds Cs$
  - Where $CT; \Gamma \vdash New Ca es : Ca'$
  - And $\vdash\operatorname{mtypes}(CT,m, Ca') = Cs \rightarrow C$
  - And $Ds-typs: CT; \Gamma \vdash+ Ds : Ds$
  - And $Ds-subss: CT \vdash+ Ds <: Cs$

31
and UI: 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: typings-append)
moreover from A-I-4[of - mb mt] r-invk obtain Da E
  where CT ⊢ Ca <: Da
and E-sub-C: CT ⊢ E <: C
and e0-typ1: CT;[xs[→]Cs,this→Da] ⊢ e : E by auto
moreover with Ds-subs have CT ⊢+ (Ds@[Ca]) <: (Cs@[Da])
by (auto simp add: subtyping-append)
ultimately have ex: ∃As. CT;Γ ⊢+ (ds@[New Ca es]) : As ∧ CT ⊢+ As <:
(Cs@[Da])
by auto
from e0-typ1 have e0-typ2: CT;(Γ ++ [xs[→]Cs,this→Da]) ⊢ e : E
by (simp only: A-I-3)
from e0-typ2 mtype-mbody-length[of mt mb]
have e0-typ3: CT;(Γ ++ [(xs@[this])[→](Cs@[Da])]) ⊢ e : E
by (force simp only:mapping_shuffle)
let ?Γ1 = Γ and ?Γ2 = [(xs@[this])[→](Cs@[Da])]
have g-def: (?Γ1 ++ ?Γ2) = (?Γ1 ++ ?Γ2) and g2-def: ?Γ2 = ?Γ2 by auto
from A-I-2[of - g-def g2-def - - ex] e0-typ3 r-invk UI mtype-mbody-length[of mt mb]
obtain E' where e'-typ: CT;Γ ⊢ subs [(xs@[this])[→](ds@[New Ca es])] e : E'
  and E'-sub-E: CT ⊢ E' <: E by force
moreover from e'-typ UI mtype-mbody-length[of mt mb]
have CT;Γ ⊢ subs [xs[→]ds,this→(New Ca es)] e : E'
by (auto simp only:mapping_shuffle)
moreover from E'-sub-E E-sub-C have CT ⊢ E' <: C by (rule subtyping.s-trans)
ultimately show ?case using r-invk by auto
next
case (r-cast CT Ca D es)
then obtain Ca'
  where C = D
and CT;Γ ⊢ New Ca es : Ca' by (auto elim:typing.cases)
thus ?case using r-cast by (auto elim:typing.cases)
next
case (re-field CT e0 e0')
then obtain C0 Cf fd where CT;Γ ⊢ e0 : C0
  and Cf-def: fields(CT,C0) = Cf
  and fd-def: lookup Cf (λfd. (vdName fd = f)) = Some fd
  and vdType fd = C
by (auto elim:typing.cases)
moreover with re-field obtain C'
  where CT;Γ ⊢ e0' : C'
  and CT ⊢ C' <: C0 by auto
moreover from sub-fields[of - Cf-def] obtain Cf'
where \( \text{fields}(CT, C') = (\text{If}@\text{If}') \) by rule (rule \( \Gamma \vdash C' <: C0 \))

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

by (simp add:lookup-append)

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

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

next
case \( (\text{rc-invk-recev} CT e0 e0' m es C) \)
then obtain \( C0 \) \( Ds \) \( Cs \)
where \( \text{ct-ok} : CT \ O K \)
and \( CT;\Gamma \vdash e0 : C0 \)
and \( \text{mt:} \text{mtype}(CT,m,C0) = Ds \rightarrow C \)
and \( CT;\Gamma \vdash es : Cs \)
and \( \text{length es} = \text{length Ds} \)
and \( CT \vdash Cs <: Ds \)

by (auto elim:typing.cases)

moreover with \( \text{rc-invk-receive} \) obtain \( C0' \)
where \( CT;\Gamma \vdash e0' : C0' \)
and \( CT \vdash C0' <: C0 \) by auto

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

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

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

next
case \( (\text{rc-invk-arg} CT ei ei' e0 m el er C) \)
then obtain \( Cs \) \( Ds \) \( C0 \)
where \( \text{typs} : CT;\Gamma \vdash+ (el@((ei##er))) : Cs \)
and \( e0\text{-typ} : CT;\Gamma \vdash e0 : C0 \)
and \( \text{mt:} \text{mtype}(CT,m,C0) = Ds \rightarrow C \)
and \( Cs\text{-sub-Ds}: CT \vdash+ Cs <: Ds \)
and \( \text{len:} \text{length} (el@((ei##er))) = \text{length Ds} \)

by (auto elim:typing.cases)

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

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

by auto

from \( \text{ith-typing-sub}[\text{OF typs ei-typ Ci-sub}] \) 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 \( \text{len} \) have \( \text{length} (el@((ei'##er))) = \text{length Ds} \) by simp

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

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

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

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

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

3.7 Multi-Step Subject Reduction Theorem

corollary Cor-2-4-1-multi:
  assumes \(CT \vdash e \rightarrow* e'\)
  and \(CT \ OK\)
  hence \(CT; \Gamma \vdash e : C\)
  by(auto elim:typing.cases)

34
shows $\exists C. \ [CT;\Gamma \vdash e : C] \Rightarrow \exists C'. \ (CT;\Gamma \vdash e' : C' \land CT \vdash C' <: C)$

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'')
hence e-typ: $CT;\Gamma \vdash e : C$
and e-step: $CT \vdash e \rightarrow e'$
and ct-ok: $CT OK$

and IH: $\forall D. \ [CT;\Gamma \vdash e' : D; CT OK] \Rightarrow \exists E. \ CT;\Gamma \vdash e'' : E \land CT \vdash E <: D$

by auto

from Thm-2-4-1[OF e-step ct-ok e-typ]
obtain D where $e''$-typ: $CT;\Gamma \vdash e'' : D$
and D-sub-C: $CT \vdash D <: C$

by auto

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

by auto

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

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 $CT;\text{Map.empty} \vdash e : C$
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 $CT;\text{Map.empty} \vdash (\text{FieldProj \ (New C0 es) } fi) : Ci$

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

then obtain Cf fDef C0'

where $CT;\text{Map.empty} \vdash (\text{New C0 es}) : C0'$
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 $CT;\text{Map.empty} \vdash e : C$
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 $CT: Map.empty \vdash MethodInvk (New C0 es) m ds : D$
using \text{assms} by (force simp add:subexprs-typing)
then obtain $C0' Cs$
where $CT: Map.empty \vdash (New C0 es) : C0'$
and $mt: mtype(CT,m,C0') = Cs \rightarrow D$
and $length ds = length Cs$
by (auto elim:typing.cases)
with $mtype-nobody(OF mt)$ show $\text{thesis}$ by (force elim:typing.cases)
qed

lemma $closed-subterm-split$:
assumes $CT: \Gamma \vdash e : C \text{ and } \Gamma = Map.empty$
shows 
\begin{align*}
(\exists C0 es fi. (FieldProj (New C0 es) fi) \in \text{subexprs}(e)) \\
\vee (\exists C0 es m ds. (MethodInvk (New C0 es) m ds) \in \text{subexprs}(e)) \\
\vee (\exists C0 D es. (Cast D (New C0 es)) \in \text{subexprs}(e)) \\
\vee \text{val}(e) \in \{ ?e \in ?M e \vee C e \vee \forall e \in ?IH e \}
\end{align*}
using \text{assms}

proof (induct $CT \times C rule:typing-induct$)
- case $1$ thus $?case$ using \text{assms} by auto
next
- case $(2 C CT \Gamma x)$ thus $?case$ by auto
next
- case $(3 C C t Cf Ci \Gamma e0 fDef fi)$
  have $s1: e0 \in \text{subexprs}(\text{FieldProj } e0 \text{ } fi)$
  by (auto simp add:isubexprs.intros)
  from 3 have $?IH e0$ by auto
  moreover
  { assume $?F e0$
    then obtain $C0 es fi'$ where $s2: FieldProj (New C0 es) fi' \in \text{subexprs}(e0)$
    by auto
    from rtrancl-trans[OF $s2 \ s1$] have $?case$ by auto
  } moreover {
    assume $?M e0$
    then obtain $C0 es m ds$ where $s2: MethodInvk (New C0 es) m ds \in \text{subexprs}(e0)$
    by auto
    from rtrancl-trans[OF $s2 \ s1$] have $?case$ by auto
  } moreover {
    assume $?C e0$
    then obtain $C0 D es$ where $s2: Cast D (New C0 es) \in \text{subexprs}(e0)$ by auto
    from rtrancl-trans[OF $s2 \ s1$] have $?case$ by auto
  } moreover {
    assume $?V e0$
    then obtain $C0 es$ where $e0 = (New C0 es)$ and $\text{vals}(es)$
    by (force elim:val.cases)
    hence $?case$ by (force intro:isubexprs.intros)
  }
ultimately show $?case$ by blast
next
- case $(4 C C0 CT Cs Ds \Gamma e0 es m)$
  have $s1: e0 \in \text{subexprs}(\text{MethodInvk } e0 m es)$
  by (auto simp add:isubexprs.intros)

36
from 4 have ?IH e0 by auto
moreover
\{ 
  assume ?F e0 
  then obtain C0 es fi where s2: FieldProj (New C0 es) fi \in subexprs(e0) by auto
  from rtrancl-trans[OF s2 s1] have ?case by auto 
\}
moreover 
\{ 
  assume ?M e0 
  then obtain C0 es' m' ds where s2: MethodInvk (New C0 es') m' ds \in subexprs(e0) by auto 
  from rtrancl-trans[OF s2 s1] have ?case by auto 
\}
moreover 
\{ 
  assume ?C e0 
  then obtain C0 D es where s2: Cast D (New C0 es) \in subexprs(e0) by auto 
  from rtrancl-trans[OF s2 s1] have ?case by auto 
\}
moreover 
\{ 
  assume ?V e0 
  then obtain C0 es' where e0 = (New C0 es') and vals(es') by (force elim:val.cases) 
  hence ?case by(force intro:subexprs.intros) 
\}
ultimately show ?case by blast
next 
case (5 C CT Cs Df Ds \Gamma es)

hence
  length es = length Cs
  \land i. [i < length es; CT;\Gamma \vdash (es!i) : (Cs!i); \Gamma = Map.empty] \Longrightarrow ?IH (es!i)
  and CT;\Gamma \vdash es : Cs
  by (auto simp add:typings-lengths)

hence (\exists i < length es. (?F (es!i) \lor ?M (es!i) \lor ?C (es!i))) \lor (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;\Gamma \vdash (h#t) : (Ch#Ct)
    and OIH: \land i. [i < length (h#t); CT;\Gamma \vdash ((h#t)!i) : ((Ch#Ct)!i); \Gamma = Map.empty] \Longrightarrow ?IH ((h#t)!i)
    and G-def: \Gamma = Map.empty
    by auto 

from 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:typing.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;\Gamma \vdash (t!i) : (Ct!i); \Gamma = Map.empty] \Longrightarrow ?IH (t!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 \( ?F \ (t!i) \lor \ ?M \ (t!i) \lor \ ?C \ (t!i) \) \text{ by force} 
  hence \( \text{Suc } i < \text{length } ((h\#t)!{(Suc } i)) \) \text{ by force} 
  hence \( \exists i < \text{length } ((h\#t)!{(Suc } i)) \) \text{ by force} 
  \} 

moreover \{ 
  assume \( v\!-\!t : \text{vals}(t) \) 
  from \( \text{OIH}\{ \text{OF - h-typ } G\text{-def} \} \) \text{ have } \?IH \ h \text{ by auto} 
moreover \{ 
  assume \( ?F \ h \lor ?M \ h \lor ?C \ h \) 
  hence \( ?F \ ((h\#t)!0) \lor ?M \ ((h\#t)!0) \lor ?C \ ((h\#t)!0) \) \text{ by auto} 
  hence \( ?Q \ (h\#t) \) \text{ by force} 
\} 

moreover \{ 
  assume \( ?V \ h \) 
  with \( \text{v\!-\!t have vals}((h\#t)) \) \text{ by (force intro:vals-val.intros)} 
  hence \( ?Q((h\#t)) \text{ by auto} \) 
\} 

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

qed 

moreover \{ 
  assume \( \exists i < \text{length } es. \ ?F \ (es!i) \lor ?M \ (es!i) \lor ?C \ (es!i) \) 
  then obtain \( i \) where \( \text{i-len: } i < \text{length } es \text{ and } r: ?F \ (es!i) \lor ?M \ (es!i) \lor ?C \ (es!i) \) \text{ by force} 
    from \( \text{ith-mem}\{ \text{OF i-len} \} \) \text{ have } s1:es!i \in \text{subexprs}(\text{New } C \ es) \text{ by(auto intro:subexprs.se-newarg}) 
    \{ 
      assume \( ?F \ (es!i) \) 
      then obtain \( C0 \ es'i \) \text{ where } s2: \text{FieldProj} (\text{New } C0 \ es') \text{ if } \in \text{subexprs}(es!i) \text{ by auto} 
    \} 
moreover \{ 
  assume \( ?M \ (es!i) \) 
  then obtain \( C0 \ es'm \) \text{ where } s2: \text{MethodInuk} (\text{New } C0 \ es') \text{ m' ds } \in \text{subexprs}(es!i) \text{ by force} 
    from \( \text{rtrancl-trans}\{ \text{OF s2 s1} \} \) \text{ have } ?F(\text{New } C \ es) \lor ?M(\text{New } C \ es) \lor ?C(\text{New } C \ es) \text{ by auto} 
\} 
moreover \{ 
  assume \( ?C \ (es!i) \) 
  then obtain \( C0 \ D \ es'm \) \text{ where } s2: \text{Cast } D \ (\text{New } C0 \ es') \in \text{subexprs}(es!i) \text{ by auto} 
    from \( \text{rtrancl-trans}\{ \text{OF s2 s1} \} \) \text{ have } ?F(\text{New } C \ es) \lor ?M(\text{New } C \ es) \lor ?C(\text{New } C \ es) \text{ by auto} 
\} 

ultimately have \( ?F(\text{New } C \ es) \lor ?M(\text{New } C \ es) \lor ?C(\text{New } C \ es) \text{ using r by blast} \)
hence \( ?\)case by auto

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

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

3.9 Type Soundness Theorem

theorem Thm-2-4-3:
assumes e-typ: CT;Map.empty ⊢ e : C
and ct-ok: CT OK
and multisteps: CT ⊢ e →* e1
and no-step: ¬(∃ e2. CT ⊢ e1 → e2)
shows (val(e1) ∨ (∃ D. CT;Map.empty ⊢ e1 : D ∧ CT ⊢ D <: C))
∨ (∃ D C es. (Cast D (New C C es) ∈ subexprs(e1) ∧ CT ⊢ C ¬<: D))
proof –
from assms Cor-2-4-1-multi[of multisteps ct-ok e-typ] obtain C1
where e1-typ: CT;Map.empty ⊢ e1 : C1
and C1-sub-C: CT ⊢ C1 <: C by auto
from e1-typ have \((\exists C0 \ es \ fi. (\text{FieldProj} \ (\text{New} \ C0 \ es) \ fi) \in \text{subexprs}(e1))\)
\(\lor (\exists C0 \ es \ m \ ds. (\text{MethodInuk} \ (\text{New} \ C0 \ es) \ m \ ds) \in \text{subexprs}(e1))\)
\(\lor (\exists C0 \ D \ es. (\text{Cast} \ D \ (\text{New} \ C0 \ es)) \in \text{subexprs}(e1))\)
\(\lor \text{val}(e1)\) (is ?F e1 \(\lor \ ?M e1 \lor \ ?C e1 \lor \ ?V e1\) by \(\text{simp add: closed-subterm-split}\))
moreover
\{ assume ?F e1
  then obtain C0 es fi where fp: \text{FieldProj} \ (\text{New} \ C0 \ es) \ fi \in \text{subexprs}(e1) \text{ by auto} \\
  then obtain Ci where CT;\text{Map.empty} \vdash \text{FieldProj} \ (\text{New} \ C0 \ es) \ fi : Ci \text{ using e1-typ by \(\text{force simp add:subexpr-typing}\) }
  then obtain C0’ where new-typ: CT;\text{Map.empty} \vdash \text{New} \ C0 \ es : C0’ \text{ by \(\text{force elim: typing.cases}\) }
  hence C0 = C0’ by \(\text{auto elim:typing.cases}\) \\
  with new-typ obtain Df where fi1: \text{fields}(CT,C0) = Df \text{ and } \text{lens} : \text{length es} = \text{length Df} \text{ by \(\text{auto elim:typing.cases}\) }
  from Thm-2-4-2-1[OF e1-typ fp] obtain Cf \text{fDef} \\
  where f2: \text{fields}(CT,C0) = Cf \\
  and \text{lkup}: \text{lookup} \ Cf \ (\lambda fd. \text{vdName} \ fd = \text{fi}) = \text{Some(fDef)} \text{ by \(\text{force}\) }
  moreover from \text{fields-functional}[OF f1 ct-ok f2] \text{length es} = \text{length Cf} \text{ by \(\text{auto}\) }
  moreover from \text{lookup-index}[OF \text{lkup}] obtain i where 
  \(\text{i<length Cf}\)
  and \text{fDef} = Cf ! i \\
  and (\text{length Cf} = \text{length es} ) \rightarrow \text{lookup}2 \text{ Cf} \ es \ (\lambda fd. \text{vdName} \ fd = \text{fi}) = \text{Some} \ 
  \(\text{(es} ! i) \text{ by \(\text{auto}\) }
  ultimately have \text{lookup}2 \text{ Cf} \ es \ (\lambda fd. \text{vdName} \ fd = \text{fi}) = \text{Some (es!i)} \text{ by \(\text{auto}\) }
  with f2 \text{ have } \text{CT} \vdash \text{FieldProj} \ (\text{New} \ C0 \ es) \ fi \rightarrow (\text{es!i}) \text{ by \(\text{auto intro: reduction.intros}\) }
  with fp \text{ have } \exists e2. \text{ CT} \vdash e1 \rightarrow e2 \text{ by \(\text{simp add: subexpr-reduct}\) }
  with no-step have ?thesis by \(\text{auto}\) 
\} moreover \{ 
  assume ?M e1
  then obtain C0 es m ds where mi: \text{MethodInuk} \ (\text{New} \ C0 \ es) \ m \ ds \in \text{subexprs(e1)} \text{ by \(\text{auto}\) }
  then obtain D where CT;\text{Map.empty} \vdash \text{MethodInuk} \ (\text{New} \ C0 \ es) \ m \ ds : D 
  using e1-typ by \(\text{force simp add:subexpr-typing}\) 
  then obtain C0’ Es E \\
  where m-typ: CT;\text{Map.empty} \vdash \text{New} \ C0 \ es : C0’ \\
  and \text{mtype}(CT,m,C0’) = Es \rightarrow E \\
  and \text{length ds} = \text{length Es} \\
  by \(\text{auto elim:typing.cases}\) 
  from Thm-2-4-2-2[OF e1-typ mi] obtain xs e0 where mb: \text{mbody} (CT, m, C0) = xs . e0 \text{ and } \text{length xs} = \text{length ds} \text{ by \(\text{auto}\) }
  hence CT \vdash \text{MethodInuk} \ (\text{New} \ C0 \ es) \ m \ ds \rightarrow (\text{mb} \text{subts}[\text{xs} \rightarrow \text{ds}, \text{this} \rightarrow (\text{New} \ C0 \ es)\text{e0}) \ text{ by \(\text{auto simp add: reduction.intros}\) }
  with mi \text{ have } \exists e2. \text{ CT} \vdash e1 \rightarrow e2 \text{ by \(\text{simp add: subexpr-reduct}\) }
  with no-step have ?thesis by \(\text{auto}\) 
\} moreover \{ 
  assume ?C e1
  then obtain C0 D es where c-def: Cast D \ (\text{New} \ C0 \ es) \in \text{subexprs(e1)} \text{ by }

41
then obtain $D'$ where $CT:\text{Map.empty} \vdash \text{Cast } D \ (\text{New } C0 es) : D'$ using $e1$-typ by (force simp add: subexpr-typing)

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

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

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

case True

hence $CT \vdash \text{Cast } D \ (\text{New } C0 es) \rightarrow (\text{New } C0 es)$ 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 $\text{thesis by auto}$

next

case False

with c-def show $\text{thesis by auto}$

qed

} moreover {

assume $?V e1$

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

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

qed

end

theory Execute

imports FJSound

begin

4 Executing FeatherweightJava programs

We execute FeatherweightJava programs using the predicate compiler.

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

thm subtyping.equation

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

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

lemma [code-pred-intro]: append [] xs xs

unfolding append-def by simp

lemma [code-pred-intro]: append xs ys zs \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:
lemma \textit{rc-invk-arg}': \( CT \vdash ei \rightarrow ei' \Rightarrow \text{append el (ei \# er)} \ e' \Rightarrow \text{append el (ei' \# er)} \ e'' \Rightarrow CT \vdash \text{MethodInvk e m e'} \rightarrow \text{MethodInvk e m e''} \)

unfolding append-def by simp (rule reduction.intros(6))

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

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

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

code-pred (modes: i => i => i => bool, i => i => o => bool as reduce)

\textit{reduction}

\textbf{proof –}
  case append
    from this show thesis
      unfolding append-def by (cases xa) fastforce+
  next
  case reduction
    from reduction.prems show thesis
    proof (cases rule: reduction.cases)
      case r-field
        with reduction(1) show thesis by fastforce
      next
      case r-invk
        with reduction(2) show thesis by fastforce
      next
      case r-cast
        with reduction(3) show thesis by fastforce
      next
      case 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
We also make the class typing executable: this requires that we derive rules for method-typing.

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

lemma method-typing-aux:
    (∀ Ds D0. mtype(CT,m,D) = Ds → D0 → Cs = Ds ∧ C = D0) = (¬ method-typing-aux CT m D Cs C)

unfolding method-typing-aux-def by auto

lemma [code-pred-intro]:
    mtype(CT,m,D) = Ds → D0 ⇒ Cs ≠ Ds ⇒ method-typing-aux CT m D Cs C

unfolding method-typing-aux-def by auto

lemma [code-pred-intro]:
    mtype(CT,m,D) = Ds → D0 ⇒ C ≠ D0 ⇒ method-typing-aux CT m D Cs C

unfolding method-typing-aux-def by auto

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

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

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

proof –
    case class-typing
    from class-typing.cases[OF class-typing.prems, of thesis] this(1) show thesis
    unfolding append-def by fastforce

next
    case method-typing
    from method-typing.cases[OF method-typing.prems, of thesis] this(1) show thesis
    unfolding append-def method-typing-aux-def by fastforce

next
    case method-typing-aux
    from this show thesis
    unfolding method-typing-aux-def by auto

qed
4.1 A simple example

We now execute a simple FJ example program:

\begin{verbatim}
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 ]], cConstructor = ( kName = cPair, kParams = [[ vdName = ffst, 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 ] ] )
\end{verbatim}
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 []]) fst) snd) snd →∗ x}

end
theory Featherweight-Java
imports FJSound Execute
begin

end

References
