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

\begin{verbatim}
 type-synonym varName = nat
type-synonym methodName = nat
type-synonym className = nat
\end{verbatim}
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**

\( \text{classTable} = \text{className} \rightarrow \text{classDef} \)

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} \ (\cdot \in \text{isubexprs}'(\cdot) \ [80,80] \ 80) \ )

**where**

\[
\begin{align*}
e' \in \text{isubexprs}(e) & \equiv (e',e) \in \text{isubexprs} \\
se-field & : e \in \text{isubexprs}(\text{FieldProj } e \ f) \\
se-invcrecv & : e \in \text{isubexprs}(\text{MethodInvk } e \ m \ es) \\
se-invarg & : \{ e_i \in \text{set } es \} \implies e_i \in \text{isubexprs}(\text{MethodInvk } e \ m \ es) \\
se-newarg & : \{ e_i \in \text{set } es \} \implies e_i \in \text{isubexprs}(\text{New C } es) \\
se-cast & : e \in \text{isubexprs}(\text{Cast C } e)
\end{align*}
\]

**abbreviation**

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

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

1.3 Values

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

**inductive**

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

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

**where**

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

1.4 Substitution

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

**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} \text{ where}

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

\[ \text{sublst1 } \sigma \ [] = [] \]

\[ \text{sublst1 } \sigma \ (h \ # \ t) = (\text{substs } \sigma \ h) \ # (\text{sublst1 } \sigma \ t) \]

\[ \text{sublst2 } \sigma \ [] = [] \]

\[ \text{sublst2 } \sigma \ (h \ # \ t) = (\text{substs } \sigma \ h) \ # (\text{sublst2 } \sigma \ t) \]

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

where
\[(ds/xs)e \equiv \text{substs } (\text{map-upds empty xs ds}) \ e \]

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

where
\[(ds/xs)es \equiv \text{map } (\text{substs } (\text{map-upds empty xs ds})) \ es \]

1.5 Lookup

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

\[ \text{primrec \ lookup} :: 'a \ list \Rightarrow ('a \Rightarrow \text{bool}) \Rightarrow 'a \ option \]

where
\[ \text{lookup} \ [] \ P = \text{None} \]
\[ \text{lookup} \ (h \# t) \ P = (\text{if } P \ h \ \text{then Some } h \ \text{else } \text{lookup} \ t \ P) \]

\[ \text{primrec \ lookup2} :: 'a \ list \Rightarrow 'b \ list \Rightarrow ('a \Rightarrow \text{bool}) \Rightarrow 'b \ option \]

where
\[ \text{lookup2} \ [] \ l2 \ P = \text{None} \]
\[ \text{lookup2} \ (h1 \# t1) \ l2 \ P = (\text{if } P \ h1 \ \text{then Some } (\text{hd} \ l2) \ \text{else } \text{lookup2} \ t1 \ (\text{tl} \ l2) \ P) \]

1.6 Variable Definition Accessors

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

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

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

1.7 Subtyping Relation

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

**inductive**

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

$\quad \text{where}$

$\quad \text{s-refl} : CT \vdash C <: C$

$\quad \text{s-trans} : \begin{array}{l}
CT \vdash C <: D; CT \vdash D <: E \implies CT \vdash C <: E
\end{array}$

$\quad \text{s-super} : \begin{array}{l}
CT(C) = \text{Some}(CDef); \ cSuper CDef = D \implies CT \vdash C <: D
\end{array}$

**abbreviation**

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

$\quad \text{where}$

$\quad \text{CT} \vdash S \not<: T \equiv \neg \text{CT} \vdash S <: T$

**inductive**

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

$\quad \text{where}$

$\quad \text{ss-nil} : CT \vdash +[] <: []$

$\quad \text{ss-cons} : \begin{array}{l}
CT \vdash C_0 <: D_0; CT \vdash +Cs <: Ds \implies CT \vdash +(C_0 \# Cs) <: (D_0 \# Ds)
\end{array}$

1.8 fields Relation

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

**inductive**

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

$\quad \text{where}$

$\quad \text{f-obj}:$

$\quad \quad \text{fields}(CT, \text{Object}) = []$

$\quad \quad \text{f-class}:$

$\quad \quad \quad \begin{array}{l}
CT(C) = \text{Some}(CDef); \ cSuper CDef = D; \ cFields CDef = C_f; \text{fields}(CT, D) = D_g; \ D_gCf = D_g @ C_f
\end{array}$

$\implies \text{fields}(CT, C) = D_gCf$

1.9 mtype Relation

The **mtype** relation, written $\text{mtype}(CT, m, C) = C_s : C_0$ relates a class $C$, method name $m$, and the arrow type $C_s : C_0$. 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 (\(\vdash \) \(+\) \(-\) : \([80, 80, 80]\) \(80\))

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

\[80\) where

\(ts\)-nil : \(CT;\Gamma \vdash [\] : [\]

| ts-cons :
| \[ CT;\Gamma \vdash e0 : C0; \  CT;\Gamma \vdash+ es : Cs \]
| \(⇒ \) \(CT;\Gamma \vdash+ (e0 \# es) : (C0 \# Cs)\)

| t-var :
| \[ \Gamma(x) = Some C \] \(⇒ \) CT;\Gamma \vdash (Var x) : C

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

| t-invk :
| \[ CT;\Gamma \vdash e0 : C0;\]
| \( mtype(CT, m, C0) = Ds \rightarrow C;\)
| \(CT;\Gamma \vdash+ es : Cs;\)
| \(CT \vdash+ Cs <: Ds;\)
| \(length es = length Ds \]
| \(⇒ \) \(CT;\Gamma \vdash MethodInvk e0 m es : C\)

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

| t-ucast :
| \[ CT;\Gamma \vdash e0 : D;\]
| \(CT \vdash D <: C \]
| \(⇒ \) \(CT;\Gamma \vdash Cast C e0 : C\)

| t-dcast :
| \[ CT;\Gamma \vdash e0 : D;\]
| \(CT \vdash C <: D; \ C \neq D \]
| \(⇒ \) \(CT;\Gamma \vdash Cast C e0 : C\)

| t-ecast :
| \[ CT;\Gamma \vdash e0 : D;\]

8
\[ CT \vdash C \not<; D; \]
\[ CT \vdash D \not<; C' \]
\[ \implies CT; \Gamma \vdash \text{Cast } C e0 : C \]

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

**lemma** typing-induct:

- **assumes** \( CT; \Gamma \vdash e : C \) (is \(?T\))
- **and** \( \forall C \ CT \Gamma x. \Gamma x = \text{Some } C \implies P CT \Gamma (\text{Var } x) C \)
- **and** \( \forall C0 \ CT Cf Ci \Gamma e0 \text{ fd fDef }\).
  \[ [CT; \Gamma \vdash e0 : C0; P CT \Gamma e0 C0; \text{fields}(CT,C0) = Cf; \text{lookup } Cf (\lambda fd. \text{vdName } fd = f) = \text{Some } f\text{Def}; \text{vdType } f\text{Def} = C] \implies P \]
  \[ CT \Gamma (\text{FieldProj } e0 f) Ci \]
- **and** \( \forall C C0 \ CT Cs Ds \Gamma e0 es m. \]
  \[ [CT; \Gamma \vdash e0 : C0; P CT \Gamma e0 C0; \text{mtype}(CT,m,C0) = Ds \rightarrow C; CT; \Gamma \vdash+ es : Cs; \forall i. [i < \text{length } es] \implies P \]
  \[ CT \Gamma (\text{es}!i) (Csi); CT \vdash+ Cs <; Ds] \implies P CT \Gamma (\text{MethodInvk } e0 m \text{ es} C) C \]
- **and** \( \forall C CT C D \Gamma e0. \]
  \[ [CT; \Gamma \vdash e0 : D; P CT \Gamma e0 D; CT \vdash D <; C] \implies P CT \Gamma (\text{New } C e0) C \]
  \[ \implies P CT \Gamma (\text{Cast } C e0) C \]
- **shows** \( P CT \Gamma e C \) (is \(?P\))

**proof**

- **fix** \( es Cs \)
- **let** \(?IH=\text{CT}; \Gamma \vdash+ es : Cs \rightarrow (\forall i < \text{length } es. \ P CT \Gamma (\text{es}!i) (Csi))\)
- **have** \(?IH \land (?T \rightarrow ?P)\)
- **proof** **(induct rule: typings-typing.induct)**
  - **case** \( (\text{ts-nil } CT \Gamma) \) **show** \(?case by auto\)
  - **next**
  - **case** \( (\text{ts-cons } CT \Gamma e0 C0 es Cs)\)
  - **show** \(?case proof\)
  - **fix** \( i\)
  - **show** \( i < \text{length } (e0\#es) \rightarrow P CT \Gamma ((e0\#es)!i) ((C0\#Cs)!i) \) **using** \( \text{ts-cons by (cases } i, \text{ auto})\)
  - **qed**
  - **next**
  - **case** \( t\text{-var} \) **then show** \(?case using assms by auto\)
  - **next**
  - **case** \( t\text{-field} \) **then show** \(?case using assms by auto\)
  - **next**
  - **case** \( t\text{-invk} \) **then show** \(?case using assms by auto\)
  - **next**
  - **case** \( t\text{-new} \) **then show** \(?case using assms by auto\)
case t-ucast then show \(\text{?case using } \text{assms by auto}\) next
case t-dcast then show \(\text{?case using } \text{assms by auto}\) next
case t-scast then show \(\text{?case using } \text{assms by auto}\) qed
thus \(\text{?thesis using } \text{assms by auto}\) 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 \(C\).

\[
\text{inductive }\ \text{method-typing} :: [\text{classTable}, \text{methodDef}, \text{className}] \Rightarrow \text{bool}
\]
where
\[
\text{m-typing}:
\]
\[
\begin{align*}
\text{CT}(C) &= \text{Some}(C\text{Def}); \\
\text{cName} \ C\text{Def} &= C; \\
\text{cSuper} \ C\text{Def} &= D; \\
\text{mName} \ m\text{Def} &= m; \\
\text{lookup} \ (\text{cMethods} \ C\text{Def}) (\lambda m. \text{mName} \ md = m)) &= \text{Some}(m\text{Def}); \\
\text{mReturn} \ m\text{Def} &= C0; \\
\text{mParams} \ m\text{Def} &= Cxs; \\
\text{mBody} \ m\text{Def} &= e0; \\
\text{varDefs-types} \ Cxs &= Cs; \\
\text{varDefs-names} \ Cxs &= xs; \\
\Gamma &= (\text{map-upds empty xs Cs}(\text{this} \mapsto C)); \\
\text{CT},\Gamma \vdash e0 : E0; \\
\text{CT} \vdash E0 <: C0; \\
\forall Ds \ D0. \ (\text{mtype}(CT,m,D) = Ds \rightarrow D0) \rightarrow (Cs=Ds \land C0=D0) \] \\
\Rightarrow & \ \text{CT} \vdash m\text{Def} \text{ OK IN } C
\end{align*}
\]

\[
\text{inductive }\ \text{method-typings} :: [\text{classTable}, \text{methodDef list}, \text{className}] \Rightarrow \text{bool}
\]
where
\[
\text{ms-nil} :
\]
\[
\text{CT} \vdash \ [ ] \text{ OK IN } C
\]
\[
\text{ms-cons} :
\]
\[
\text{CT} \vdash \ m \text{ OK IN } C; \\
\text{CT} \vdash \ms \text{ OK IN } C \] \\
\Rightarrow & \ \text{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.

**Inductive**

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

**where**

- **t-class:**
  \[
  \begin{align*}
  & c\text{Name } CDef = C; \\
  & c\text{Super } CDef = D; \\
  & c\text{Constructor } CDef = KDef; \\
  & c\text{Methods } CDef = M; \\
  & k\text{Name } KDef = C; \\
  & k\text{Params } KDef = \{Dg@Cf\}; \\
  & k\text{Super } KDef = \text{varDefs-names } Dg; \\
  & k\text{Inits } KDef = \text{varDefs-names } Cf; \\
  & \text{fields}(CT,D) = Dg; \\
  & CT \vdash M \text{ OK IN } C \]
  
  \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**

\[
\text{ct-typing :: classTable } \Rightarrow \text{bool } \neg \text{OK } 80
\]

**where**

- **ct-all-ok:**
  \[
  \begin{align*}
  & \text{Object } \notin \text{dom}(CT); \\
  & \forall C CDef. CT(C) = \text{Some}(CDef) \Rightarrow (CT \vdash CDef \text{ OK}) \land (\text{cName } CDef = C) \\
  \end{align*}
  \]
  \[
  \Rightarrow CT \text{ OK}
  \]

1.15 Evaluation Relation

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

**Inductive**

\[
\text{reduction :: [classTable, exp, exp] } \Rightarrow \text{bool } \neg \vdash - [80,80,80] 80
\]

**where**

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

- **r-invk:**
  \[
  \begin{align*}
  & \text{mbody}(CT,m,C) = xs . e0; \\
  \end{align*}
  \]
\( \text{subs } ((\text{map-upds empty xs ds})(\text{this }\mapsto (\text{New C es}))) \) \( e_0 = e_0' \] 
\( \implies \text{CT} \vdash \text{MethodInvk} (\text{New C es}) \) \( m \) \( ds \to e_0' \)

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

| re-field: 
\[ \text{CT} \vdash e_0 \to e_0' \] 
\( \implies \text{CT} \vdash \text{FieldProj} e_0 f \to \text{FieldProj} e_0' f \)

| re-invk-recev: 
\[ \text{CT} \vdash e_0 \to e_0' \] 
\( \implies \text{CT} \vdash \text{MethodInvk} e_0 m \) \( es \to \text{MethodInvk} e_0' m \) \( es \)

| re-invk-arg: 
\[ \text{CT} \vdash e_i \to e_i' \] 
\( \implies \text{CT} \vdash \text{MethodInvk} e_0 m (el@ei#er) \to \text{MethodInvk} e_0 m (el@ei'#er) \)

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

| r-cast: 
\[ \text{CT} \vdash e_0 \to e_0' \] 
\( \implies \text{CT} \vdash \text{Cast} C e_0 \to \text{Cast} C e_0' \)

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

\textbf{end}

2 \textbf{FJAux: Auxiliary Lemmas}

\textbf{theory FJAux imports FJDefs}
\textbf{begin}

2.1 \textbf{Non-FJ Lemmas}

2.1.1 Lists

\textbf{lemma} mem-ith: 
\textbf{assumes} \( e_i \in \text{set es} \)
\textbf{shows} \( \exists \) \( el \) \( er \). \( es = el@ei#er \)
\textbf{using} \ assms
\textbf{proof}(\text{induct es})
thus \texttt{?case by auto} \\

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

\begin{align*}
\text{lemma} \ i\text{th-mem}: \quad & \bigwedge i. \ [ i < \text{length es} ] \implies \text{ests}!i \in \text{set es} \\
\text{proof} (\text{induct es}) \\
\texttt{case Nil \texttt{thus \texttt{?case by auto}} \\
next \texttt{case (Cons \textit{h t}) \texttt{thus \texttt{?case by (cases i, auto)}}} \\
qed \\
\end{align*}

2.1.2 Maps

\begin{align*}
\text{lemma} \ \text{map-shuffle}: \\
& \text{assumes \ length \textit{xs} = length \textit{ys}} \\
& \text{shows \ \textit{xs} \mapsto \mapsto \textit{ys}, \textit{x} \mapsto \mapsto \textit{y}} = \textit{\{(xs@\{x\})[\mapsto \mapsto] (ys@\{y\})\}} \\
& \text{using \ assms} \\
& \text{by \ (induct \textit{xs} \textit{ys} \texttt{rule:list-induct2}) (auto \ simp add: map-upds-append1)} \\
\end{align*}

\begin{align*}
\text{lemma} \ \text{map-upds-index}: \\
& \text{assumes \ length \textit{xs} = length \textit{As}} \\
& \text{and \ \textit{xs} \mapsto \mapsto \textit{As} \textit{x} = \text{Some \textit{Ai}}} \\
& \text{shows \ \exists i. (\text{As}!i = \textit{Ai})} \\
& \quad \land (i < \text{length \textit{As}}) \\
& \quad \land (\forall (\text{Bs}::{}c \text{ list}).((\text{length \textit{Bs} = length \textit{As}}) \implies (\textit{xs} \mapsto \mapsto \textit{Bs} \textit{x} = \text{Some \textit{Bs}!i)}))) \\
& \quad (\text{is} \ \exists i. \ (\text{?P1 \ i} \ \textit{As}) \land (\text{?P2 i} \ \textit{As}) \land (\forall \textit{Bs}::({}c \text{ list}).(\text{?P3 \ i \ xs} \ \textit{As} \ \textit{Bs}))) \\
& \text{using \ assms} \\
\text{proof} (\text{induct \textit{xs} \textit{As} \texttt{rule:list-induct2})} \\
& \text{assume \ \texttt{[]\mapsto[[[]]] \ x = \text{Some \textit{Ai}}} \\
& \text{moreover have \ \neg[[[]\mapsto[[[]]]}} x = \text{Some \textit{Ai} by auto} \\
& \text{ultimately show \ \exists i. \ ?P i [] [] by contradiction} \\
\end{align*}

next \texttt{fix \textit{xa} \textit{xs} \textit{y} \textit{ys} \\
\texttt{assume \ length-xa-yys: \text{length} \textit{xs} = \text{length} \textit{ys}} \\
\texttt{and \ \texttt{IH:} [\textit{xs} \mapsto \textit{ys}] \textit{x} = \text{Some \textit{Ai}} \\ \\
\texttt{and \ map-cp-\textit{Some:} [\textit{xa} \# \textit{xs} \mapsto \textit{y} \# \textit{ys}] \textit{x} = \text{Some \textit{Ai}}} \\
\texttt{then have \ \texttt{map-decomp:} [\textit{xa} \# \textit{xs} \mapsto \textit{y} \# \textit{ys}] = [\textit{xa}\mapsto\textit{y}]++] [\textit{xs}\mapsto\textit{ys}] \texttt{by fastforce} \\
\texttt{show \ \exists i. \ ?P i (\textit{xa}\#\textit{xs}) (\textit{y} \# \textit{ys})} \\
\]
proof(cases \([xs\to ys]x\))
case\{Some \(Ai'\)\}
hence \([xa\to y] ++ [xs\to ys] x = Some \(Ai'\)\)\ by\ (rule map-add-find-right)
hence \(P\) : \([xs\to ys]x = Some \(Ai\)\) using \(map-eq\-Some\ Some\ \by\ \text{simp}\)
from \(IH\{OF \(P\)\}\) obtain \(i\) where
\(R1\) : \(ys \downharpoonleft i = Ai\)
and \(R2\) : \(i < \text{length } ys\)
and \(pre\-r3\) : \(\forall (Bs::'c list). \ ?P3 i xs ys B\ by\ fastforce\)
\{ fix \(Bs::'c list\)
assume \(length\-Bs\) : \(length B\ = \text{length } (y\#ys)\)
then obtain \(n\) where \(length (y\#ys) = \text{Suc } n\) by auto
with \(length\-Bs\) obtain \(b\ bs\) where \(Bs\-def\) : \(B = b\#bs\) by (auto simp add:length-Suc-conv)
with \(length\-Bs\) have \(length ys = \text{length } bs\) by simp
with \(pre\-r3\) have \([xa\to b] ++ [xs\to bs] x = Some (bs\#1)\) by(auto simp only:map-add-find-right)
with \(pre\-r3\) \(Bs\-def\) \(length\-Bs\) have \(?P3 (i+1) (xa\#xs) (y\#ys) B\ by simp\)
\}
with \(R1\ \(R2\)\) have \(?P (i+1) (xa\#xs) (y\#ys)\) by auto
thus \(?thesis\) ..
next
case \(None\)
with \(map-decomp\ map-eq\-Some\ have\ \([xa\to y] x = Some \(Ai\)\) by \(auto\ simp\ only:map-add\-SomeD\)
hence \(ai\-def\) : \(y = Ai\ and\ x\#eq\-xa\#x = xa\ by\ \(auto\ simp\ only:map\-upd\-Some\-unfold\)\)
\{ fix \(Bs::'c list\)
assume \(length\-Bs\) : \(length B\ = \text{length } (y\#ys)\)
then obtain \(n\) where \(length (y\#ys) = \text{Suc } n\) by auto
with \(length\-Bs\) obtain \(b\ bs\) where \(Bs\-def\) : \(B = b\#bs\) by (auto simp add:length-Suc-conv)
with \(length\-Bs\) have \(length ys = \text{length } bs\) by simp
hence \(dom([xs\to ys]) = dom([xs\to bs])\) by auto
with \(None\) have \([xs\to bs] x = None\) by \(auto\ simp\ only:domIff\)
moreover from \(x\#eq\-xa\) have \(sing\-map: [xa\to b] x = Some b\) by \(auto\ simp\ only:map\-upd\-Some\-unfold\)
ultimately have \([xa\to b] ++ [xs\to bs] x = Some b\) by \(auto\ simp\ only:map\-add\-Some\-iff\)
with \(Bs\-def\) have \(?P3 0 (xa\#xs) (y\#ys) B\ by simp\ \}
with \(ai\-def\) have \(?P 0 (xa\#xs) (y\#ys)\) by auto
thus \(?thesis\) ..
qed
qed

2.2 FJ Lemmas

2.2.1 Substitution

lemma subst-list1\-eq\-map\-subs : 
\(\forall \sigma. \\text{subst-list1} \ \sigma\ l = \text{map} (\text{subs} \ \sigma)\ l\)
lemma subst-list2-eq-map-substs :
\forall \sigma. \text{subst-list2} \sigma l = \text{map} (\text{subs} \sigma) l
by (induct l, simp-all)

2.2.2 Lookup

lemma lookup-functional:
assumes lookup l f = o1
and lookup l f = o2
shows o1 = o2
using assms by (induct l) auto

lemma lookup-true:
lookup l f = Some r \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: lookup, auto)
qed

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

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

lemma lookup-index:
assumes lookup l1 f = Some e
shows \( \forall l2. \exists i < (\text{length } l2). e = l2!i \land ((\text{length } l1 = \text{length } l2) \implies \text{lookup}2 l1 l2 f = \text{Some} (l2!i)) \)
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, simp)
  moreover \{
    assume length (h1 \# t1) = length l2
    hence length l2 = Suc (length t1) by auto
    then obtain h2 t2 where l2-def:l2 = h2\#t2 by (auto simp add: length-Suc-conv)
    hence lookup2 (h1 \# t1) l2 f = Some (l2!0)
      using asm by (auto simp add: lookup2, simp)
  \}
  ultimately have ?case by auto
moreover \{ 
  assume \( \text{asm} \vdash (f \ h1) \)
  hence lookup \( tl \) \( f = \text{Some} \ e \)
  using Cons by (auto simp add:lookup.simps)

then obtain \( i \) where
  \( i < \text{length} \ tl \)
  and \( e = tl \ ! \ i \)
  and \( \text{ih}: (\text{length} \ tl = \text{length} \ (tl \ l2) \implies \text{lookup2} \ tl \ tl \ l2 \ f = \text{Some} \ ((tl \ l2) \ ! \ i)) \)
  using Cons by blast

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

moreover \{ 
  assume length \((h1 \ # \ tl)\) = length \( l2 \)
  hence lens:length \( l2 = \text{Suc} \ (\text{length} \ tl) \) by auto

then obtain \( h2 \ l2 \) where \( l2\text{-def}:l2 = h2 \# l2 \) by (auto simp add: length-Suc-conv)
  hence lookup2 \( tl \ l2 \ f = \text{Some} \ (tl \ ! \ i) \)
  using ih \( l2\text{-def} \) lens by auto
  hence lookup2 \((h1 \ # \ tl) \ l2 \ f = \text{Some} \ ((l2{\text{(Suc} \ i)}) \)
  using asm \( l2\text{-def} \) by(auto simp add:lookup2.simps)

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

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

qed

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

proof (induct \( tl \))
  case Nil thus \( \text{?case} \) by auto
next
  case (Cons \( h1 \ tl \))
  hence length \( tl = \text{Suc} \ (\text{length} \ tl) \) by auto

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

\}
moreover \{ 
  assume \( \text{asm} \vdash (f \ h1) \)
  hence \( \exists i < \text{length} \ l2. e = l2 ! i \land \text{lookup} \ tl \ f = \text{Some} \ ((tl ! i)) \)
  using Cons \( l2\text{-def} \) by auto

then obtain \( i \) where \( i < \text{length} \ tl \land e = tl ! i \land \text{lookup} \ tl \ f = \text{Some} \ (tl ! i) \)
  by auto
  hence \( (\text{Suc} \ i) < \text{length} \ ((h2 \# l2)) \land e = ((h2 \# l2) ! \ (\text{Suc} \ i)) \land \text{lookup} \ (h1 \# tl) \)
  \( f = \text{Some} \ (\text{(h1 \# tl) ! \ (Suc \ i)}) \)
  using asm by (force simp add: lookup.simps)
hence \( ?\text{case using} \ l2\text{-def by auto} \)
}\)
ultimately show \( ?\text{case by auto} \)
qed

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

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

### 2.2.3 Functional

These lemmas prove that several relations are actually functions

lemma mtype-functional:
assumes mtype(\( CT \, m, C \)) = \( C_s \to C_0 \)
and \( mtype(CT,m,C) = D_s \to D_0 \)
shows \( D_s=C_s \land D_0=C_0 \)
using \( \text{assms by induct (auto elim:mtype.cases)} \)

lemma mbody-functional:
assumes mb1: \( \text{mbody}(CT,m,C) = xs . e_0 \)
and \( \text{mb2: mbody}(CT,m,C) = ys . d_0 \)
shows \( xs = ys \land e_0 = d_0 \)
using \( \text{assms by induct (auto elim:mbody.cases)} \)

lemma fields-functional:
assumes fields(\( CT,C \)) = \( Cf \)
and \( CT \, \text{OK} \)
shows \( \land \, [ \text{fields}(CT,C) = Cf ] \implies Cf = Cf' \)
using \( \text{assms} \)
proof induct
case (f-obj \( CT \))
hence \( CT(\text{Object}) = \text{None} \, \text{by (auto elim: ct-typing.cases)} \)
thus \( ?\text{case using f-obj by (auto elim: fields.cases)} \)
next
case (f-class \( CT \, C \, CDef \, D \, Cf \, Dg \, DgCf \, DgCf' \))
hence \( f\text{-class-inv:} \)
\[(CT \ C = \text{Some CDDef}) \land (c\text{-Super CDDef} = D) \land (c\text{Fields CDDef} = Cf)\]
\[\text{and } CT \text{ OK by } \text{fastforce}\]
\[\text{hence } c\text{-not-obj: } C' \neq \text{Object by (force elim:ct-typing.cases)}\]
\[\text{from } f\text{-class have } \text{flds}\text{:fields}(CT, C) = DgCf' \text{ by } \text{fastforce}\]

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

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

**2.2.4 Subtyping and Typing**

**lemma typings-lengths:** \(\text{assumes } CT; \Gamma \vdash \text{es:Cs} \text{ shows } \text{length es} = \text{length Cs}\)
\[\text{using assms by (induct es Cs rule: list-induct2)}\]

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

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

**lemma subtyping-append:**
\(\text{assumes } CT \vdash \text{Cs <: Ds}\)
\[\text{and } CT \vdash C <: D\]
\[\text{shows } CT \vdash (\text{Cs@[C]}) <: (\text{Ds@[D]})\]
\[\text{using assms}\]

18
by (induct rule:subtyping.induct) (auto simp add:subtyping.intros elim:subtyping.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 add:typings-lengths)
  thus CT;Γ ⊢+ (es@[[e]] : (Cs@[C])) using assms
proof(induct es Cs rule:list-induct2)
  have CT;Γ ⊢ [[;]] by(simp add:typings-typing.ts-nil)
  moreover from assms have CT;Γ ⊢ e : C by simp
  ultimately show CT;Γ ⊢+ ([[[e]]] : (Cs@[C]) by (auto simp add:typings-typing.ts-cons)
next
  fix x xs y ys
  assume length xs = length ys
  and IH: [CT;Γ ⊢+ xs : ys; CT;Γ ⊢ e : C] ⇒ CT;Γ ⊢+ (xs @ [e]) : (ys @ [C])
  and x-xs-typs: CT;Γ ⊢+ (x ≠ xs) : (y ≠ ys)
  and e-typ: CT;Γ ⊢ e : C
  from x-xs-typs have x-typ: CT;Γ ⊢ x : y and CT;Γ ⊢+ xs : ys by(auto elim:typings.cases)
  with IH e-typ have CT;Γ ⊢+ (xs@[[e]] : (ys@[[C]]) by simp
  with x-typ have CT;Γ ⊢+ ((x#xs)@[[e]] : ((x#ys)@[C]) by (auto simp add:typings-typing.ts-cons)
    thus CT;Γ ⊢+ ((x # xs) @ [e]) : ((y # ys) @ [C]) by(auto simp add:typings-typing.ts-cons)
  qed
qed

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

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

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

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

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

lemma mem typings:
\(\forall Cs. [\[ CT; \Gamma \vdash es : Cs; \ \text{ei} \in \text{set es} \] \Rightarrow \exists C_i. CT; \Gamma \vdash \text{ei} : C_i]\]
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 \[ \forall D. \[ \begin{array}{l}
    CT ⊢ D \n=: C; \quad CT ⊢ C \n=: D \end{array} \] \implies CT ⊢ A \n=: 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 \n=: D
  proof (rule ccontr)
    assume \neg CT ⊢ Da \n=: 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 \n=: Da using s-trans by auto
  from da-nsub-d d-nsub-da s-trans show CT ⊢ C \n=: Da by auto
next
  case (s-super CT C CDef D Da)
  have C ≠ Da proof (rule ccontr)
    assume \neg C ≠ Da
    hence C = Da by auto
hence $CT \vdash Da <: D$ using s-super by (auto simp add: subtyping.s-super)
thus False using s-super by auto
qed
thus ?case using s-super by (auto simp add: not-subtypes-aux)
qed

2.2.5 Sub-Expressions

lemma isubexpr-typing:
  assumes $e1 \in \text{isubexprs}(e0)$
  shows $\forall C. [ CT;\emptyset \vdash e0 : C ] \Rightarrow \exists D. CT;\emptyset \vdash e1 : D$
  using assms
  by (induct rule:isubexprs.induct) (auto elim:typing.cases simp add:mem-typings)

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

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

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

3 FJSound: Type Soundness

theory FJSound imports FJAux
begin

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

3.1 Method Type and Body Connection

lemma mtype-mbody:
fixes $Cs :: \text{nat list}$
assumes $\text{mtype}(CT, m, C) = Cs \rightarrow C0$
shows $\exists xs e. \text{mbody}(CT, m, C) = xs \cdot e \land \text{length} \; xs = \text{length} \; Cs$
using assms
proof (induct rule: mtype.induct)
  case (mt-class $C0 \; Cs \; C \; \text{CDef} \; CT \; m \; m\text{Def})$
  thus ?case
    by (force simp add: varDefs-types-def varDefs-names-def elim:mtype.cases intro:mbody.mb-class)
next
  case (mt-super $CT \; C0 \; C \; \text{CDef} \; m \; D \; Cs \; C$)
  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 $\text{mt}: \text{mtype}(CT, m, C) = Cs \rightarrow C0$
and $\text{mb}: \text{mbody}(CT, m, C) = xs \cdot e$
shows $\text{length} \; xs = \text{length} \; Cs$
proof
  from mt-type-mbody[OF 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 mbbody-functional[OF mb mb2] show ?thesis by auto
qed

3.2 Method Types and Field Declarations of Subtypes

lemma A-I-1:
assumes $CT \vdash C <: D \land CT \; \text{OK}$
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 $CT \; C \; \text{CDef} \; D$)
hence $CT \vdash \text{CDef} \; \text{OK}$ and $\text{cName} \; \text{CDef} = C$
  by (auto elim:ct-typing.cases)
  with s-super obtain $M$
    where $M: CT \vdash M \; \text{OK} \; \text{IN} \; C$ and $\text{cMethods}: \text{cMethods} \; \text{CDef} = M$
    by (auto elim:class-typing.cases)
  let $\text{lookup-m} = \text{lookup} \; M \; (\lambda \text{md}. \; (\text{mName} \; \text{md} \; = \text{m}))$
  show ?case
    proof (cases $\exists \; \text{mDef}. \; \text{lookup-m} = \text{Some} \; \text{mDef}$)
      case True
then obtain mDef where m: ?lookup-m = Some mDef by (rule exE)
hence mDef-name: mName mDef = m by (rule lookup-true)

have CT ⊢ mDef OK IN C using M m by (auto simp add: method-typings-lookup)
then obtain CDef m' D' Cs' C0'
where CT: CT C = Some CDef'
and cSuper CDef' = D'
and mName mDef = m'
and mReturn: mReturn mDef = C0'
and varDefs-types: varDefs-types (mParams mDef) = Cs'
and ∀Ds D0. (mtype(CT,m',D') = Ds → D0) → Cs' = Ds ∧ C0' = D0
by (auto elim: method-typing.cases)

with s-super mDef-name have CDef = CDef'
and D = D'
and m = m'
and ∀Ds D0. (mtype(CT,m,D) = Ds → D0) → Cs' = Ds ∧ C0' = D0
by auto
thus ?thesis using s-super cMethods m CT mReturn varDefs-types by (auto intro:mtype.intros)

next
  case False
  hence ?lookup-m = None by (simp add: lookup-split)
  then show ?thesis using s-super cMethods by (auto simp add:mtype.intros)
qed

lemma sub-fields:
  assumes CT ⊢ C <: D
  shows ∀Dg. fields(CT,D) = Dg =⇒ ∃Cf. fields(CT,C) = (Dg@Cf)
using assms
proof induct
  case (s-refl CT C)
  hence fields(CT,C) = (Dg[]) by simp
  thus ?case ..

next
  case (s-trans CT C D E)
  then obtain Df Cf where fields(CT,C) = ((Dg@Df)@Cf) by force
  thus ?case by force

next
  case (s-super CT C CDef D Dg)
  then obtain Cf where cFields CDef = Cf by force
  with s-super have fields(CT,C) = (Dg@Cf) by (simp add: f-class)
  thus ?case ..
qed

3.3 Substitution Lemma

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

proof –
let $\text{?COMMON-ASMS} = (\text{CT OK}) \land (\Gamma = \Gamma I \mathbin{++} \Gamma 2) \land (\Gamma 2 = [xs \mapsto] Bs) \land (\text{length } Bs = \text{length } ds) \land (\exists As. CT;\Gamma I \vdash+ ds : As \land CT \vdash+ As <: Bs)$

have 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 $\exists case$

proof (rule impI)

have $(\text{CT};\Gamma I \vdash+ ([ds/xs][]);[]) \land (CT \vdash+ [] <: [])$

by (auto simp add: typings-typing intros subtypings-intros)

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

qed

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

show $\exists case$

proof (rule impI)

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

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

with ts-cons asms have
$\exists C.(\text{CT};\Gamma I \vdash+ (ds/xs)e0 : C) \land (CT \vdash C <: C0)$

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

by auto

then obtain $C \ Cs$ where
$(\text{CT};\Gamma I \vdash+ (ds/xs)e0 : C) \land (CT \vdash C <: C0)$

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

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

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

by (auto simp add: typings-typing intros subtypings-intros)

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

by auto

qed

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

show $\exists case$

proof (rule impI)

assume asms: $(\text{CT OK}) \land (\Gamma = \Gamma I \mathbin{++} \Gamma 2) \land (\Gamma 2 = [xs \mapsto] Bs) \land (\text{length } Bs = \text{length } ds) \land (\exists As. \ CT;\Gamma I \vdash+ ds : As \land CT \vdash+ As <: Bs)$
hence
lengths: length ds = length Bs
and G-def: Γ = Γ1 ++ Γ2
and G2-def : Γ2 = [xs[→]Bs] by auto
from lengths G2-def have same-doms: dom([xs[→]ds]) = dom(Γ2) by auto
from asms show ∃ C. CT;Γ1 ⊢ substs [xs[→]ds] (Var x) : C ∧ CT ⊢ C <: C'
proof (cases Γ2 x)
case None
with G-def t-var have G1-x: Γ1 x = Some C' by (simp add:map-add- Some-iff)
from None same-doms have x ∉ dom([xs[→]ds]) by (auto simp only: domIff)
  hence [xs[→]ds|x = None by (auto simp only: map-add- Some-iff)
  hence (ds/xs)(Var x) = (Var x) by auto
with G1-x have
  CT;Γ1 ⊢ (ds/xs)(Var x) : C' and CT ⊢ C' <: C'
  by (auto simp add: typings-typing.intros subtyping.intros)
thus ?thesis by auto
next
  case (Some Bi)
with G-def t-var have c'-eq-bi: C' = Bi by (auto simp add: map-add- SomeD)
from length xs = length ds asms have length xs = length Bs by simp
with Some G2-def have ∃ i.(Bs!i = Bi) ∧ (i < length Bs) ∧
  (∀ l.((length l = length Bs) → ([xs[→]l] x = Some (!!i))))
  by (auto simp add: map-upds-index)
then obtain i where bs-i-proj: (Bs!i = Bi)
  and i-len: i < length Bs
  and P: (∀ l:exp list).((length l = length Bs) → ([xs[→]l] x = Some (!!i)))
by fastforce
from lengths P have subst-x: ([xs[→]ds]|x = Some (ds!i)) by auto
from asms obtain As where as-ex:CT;Γ1 ⊢ ds : As ∧ CT ⊢ As <: Bs by fastforce
  hence length As = length Bs by (auto simp add: subtypings-length)
  hence proj-i: CT;Γ1 ⊢ ds!i : As!i ∧ CT ⊢ As!i <: Bs!i
  using i-len lengths as-ex by (auto simp add: typings-proj)
  hence CT;Γ1 ⊢ (ds/xs)(Var x) : As!i ∧ CT ⊢ As!i <: C'
  using c'-eq-bi bs-i-proj subst-x by auto
  thus ?thesis ..
qed
qed
next
  case (t-field CT Γ c0 C0 Cf fDef Ci)
show ?case
proof (rule impI)
  assume asms: (CT OK) ∧ (Γ = Γ1 ++ Γ2) ∧
  (Γ2 = [xs[→]Bs]) ∧ (length Bs = length ds) ∧ (∃ As. CT;Γ1 ⊢ ds : As
∧ CT ⊢ As <: Bs)
from t-field have flds: fields(CT,C0) = Cf by fastforce
from t-field asms obtain $C$ where $e0$-typ: $CT;\Gamma \vdash (ds/xs)e0 : C$ and sub:
$CT \vdash C <: C0$

by auto
from sub-fields[OF sub flds] obtain $Dg$ where flds-$C$: fields($CT,C$) = ($Cf@Dg$) ..
from t-field have lookup-$CfDg$: lookup ($Cf@Dg$) ($\lambda fd. \ vdName \ fd = fi$) = Some $fDef$
by (simp add:lookup-append)
from e0-typ flds-$C$ lookup-$CfDg$ t-field have $CT;\Gamma \vdash (ds/xs)(FieldProj e0 fi) : C$
by (simp add:typings-typingintros)
moreover have $CT \vdash C_i <: C_i$ by (simp add:subtypingintros)
ultimately show $\exists C. \ CT;\Gamma \vdash (ds/xs)(FieldProj e0 fi) : C \land CT \vdash C_i <:$
$C_i$ by auto
next
case(t-inv CK $\Gamma$ $e0$ $C0$ $m$ $Ds$ $C$ $es$ $Cs$)
show ?case
proof (rule impI)
assume asms: ($CT \ OK$) $\land$ ($\Gamma = \Gamma_i \ +++ \ \Gamma_2$) $\land$ ($\Gamma_2 = [xs \mapsto Bs]$)
$\land$ ($length \ Bs = length \ ds$) $\land$ ($\exists \ As. \ CT;\Gamma \vdash + ds : As \land CT \vdash + As <:$
$Bs$)
hence ct-ok: $CT \ OK$ ..
from t-inv CK have mtyp: $mtype(CT,m,C0) = Ds \rightarrow C$
and subs: $CT \vdash + Cs <: Ds$
and lens: $length \ es = length \ Ds$
by auto
from t-inv CK asms obtain $C'$ where
$e0$-typ: $CT;\Gamma \vdash (ds/xs)e0 : C'$ and sub': $CT \vdash C' <: C0$ by auto
from t-inv CK asms obtain $Cs'$ where
$es$-typ: $CT;\Gamma \vdash + [ds/xs]es : Cs'$ and subs': $CT \vdash + Cs' <: Cs$ by auto
have subst-e: ($ds/xs)(MethodInvk e0 m es) = MethodInvk ((ds/xs)e0) m$
($[ds/xs]es$)
by (auto simp add: subst-list1-eq-map-substs)
from
$e0$-typ
$A-I-1[OF subs' ct-ok mtyp]$
$es$-typ
subtypings-trans[OF subs' subs]
lens
subst-e
have $CT;\Gamma \vdash (ds/xs)(MethodInvk e0 m es) : C$ by (auto simp add:typings-typingintros)
moreover have $CT \vdash C <: C$ by (simp add:subtypingintros)
ultimately show $\exists C'. \ CT;\Gamma \vdash (ds/xs)(MethodInvk e0 m es) : C' \land CT \vdash C' <: C$ by auto
qed
next
case(t-new $CT \ C \ Df \ es \ Ds \ \Gamma \ Cs$)
show ?case

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

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

next
  case\((t\text{-dcast } CT \Gamma e0 D C)\)
  show \(?\text{case}\)
  proof (rule impI)
    assume asms: \((CT \text{ OK}) \wedge (\Gamma = \Gamma_1 + + \Gamma_2) \wedge (\Gamma_2 = [xs \to B_2]) \wedge \text{(length } Bs)\) 
    from t-dcast asms obtain C' where 
      e0-typ: \((CT; \Gamma_1 \vdash (ds/\alpha) e0 : C')\) by auto
      have \(CT \vdash C' <; C\) \lor
        \((C \neq C' \land CT \vdash C <; C') \lor
        \((CT \vdash C <; C' \land CT \vdash C' <; C)\) by blast
    moreover 
      assume \(CT \vdash C' <; C\)
      with e0-typ have \(CT; \Gamma_1 \vdash (ds/\alpha) (\text{Cast } C e0) : C\) by (auto simp add:
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 \((CT; \Gamma 2 \vdash \text{es} : \text{Cs}) \Rightarrow (CT; \Gamma 1 ++ \Gamma 2 \vdash \text{es} : \text{Cs})\) (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:mttype($CT,m,C$) = $Ds \rightarrow D$
shows $\exists D0 \ C0 . \ (CT \vdash C <: D0) \land$
 $\quad (CT \vdash C0 <: D) \land$
 $\quad (CT;[xs[\mapsto]Ds](this \mapsto D0) \vdash e : C0)$
using mb ct-ok mt proof (induct rule: mbody.induct)
case (mb-class $CT$ $C$ $CD$ Def $m$ mDef $zs$ $e$)
hence
 $m$-param:varDefs-types ($m$Params $m$Def) = $Ds$
and m-ret:mReturn mDef = $D$
and $CT$ $\vdash$ $C$ Def $OK$
and cName $C$ Def = $C$
by (auto elim:mttype.cases ct-typing.cases)
hence $CT$ $\vdash$ (cMethods $C$ Def) $OK$ 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 $\exists \ E0 \ (. \ (CT;[xs[\mapsto]Ds,this[\mapsto]C] \vdash e : E0) \land (CT \vdash E0 <: D))$
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 $CT$ $C$ $CD$ Def $m$ Da $zs$ $e$)
hence ct: $CT$ $OK$
and IH: $[CT \ OK; \ mttype(CT,m,Da) = Ds \rightarrow D]$
$\Rightarrow \exists D0 \ C0 . \ (CT \vdash Da <: D0) \land (CT \vdash C0 <: D)$
$\land (CT;[xs[\mapsto]Ds, this[\mapsto]D0] \vdash e:C0)$ by fastforce+
from mb-super have c-sub-da: $CT \vdash C <: Da$ by (auto simp add: s-super)
from mb-super have mt:mttype($CT,m,Da$) = $Ds \rightarrow D$ by (auto elim: mttype.cases)
from IH[OF ct mt] obtain $D0 \ C0$
where s1: $CT \vdash Da <: D0$
and $CT$ $\vdash$ $C0$ $<: D$
and $CT;[xs[\mapsto]Ds, this[\mapsto]D0] \vdash e : C0$ by auto
thus $?case$ using s-trans[OF c-sub-da s1] by blast
qed
3.6 Subject Reduction Theorem

theorem Thm-2-4-1:
assumes $CT \vdash e \rightarrow e'$
and $CT \ \text{OK}$
shows $\forall C. \ [CT; \Gamma \vdash e : C ] \Rightarrow \exists C'. \ (CT; \Gamma \vdash e' : C' \land CT \vdash C' <: C)$
using assms

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

hence new-typ: \( CT; \Gamma \vdash \) New \( Ca \) es : \( Ca \)
and mt: mtype(\( CT; m, Ca \)) = \( Cs \to C \) by (auto elim: typing.cases)
from ds-typs new-typ have \( CT; \Gamma \vdash+ (\text{ds @}[\text{New Ca es}]) : (\text{Ds @} [\text{Ca}]) \)
by (simp add: typings-append)

moreover from A-1-4[of - mb mt] r-inv k obtain \( Da \) \( E \)
where \( CT \vdash Ca <: Da \)
and E-sub-C: \( CT \vdash E <: C \)
and e0-typ1: \( CT; [xs[\to]Cs, this\to Da] \vdash e : E \) by auto
moreover with Ds-subs have \( CT \vdash+ (\text{Ds @} [\text{Ca}]) <: (\text{Cs @} [\text{Da}]) \)
by (auto simp add: subtyping-append)
ultimately have \( \exists As. \) \( CT; \Gamma \vdash (\text{ds @} [\text{New Ca es}]) : (As \land CT \vdash+ As <: (\text{Cs @} [\text{Da}]) \)
by auto
from e0-typ1 have e0-typ2: \( CT; (\Gamma ++ [xs[\to]Cs, this\to Da]) \vdash e : E \)
by (simp only: A-1-3)

from e0-typ2 mtype-mbody-length[of mt mb]
have e0-typ3: \( CT; (\Gamma ++ [(xs@[this])[\to](\text{Cs @} [\text{Da}]))) \vdash e : E \)
by (force simp only: map-shuffle)
let \( \Delta \Gamma = \Gamma \) and \( \Delta \Gamma' = (xs@[this] [\to]) (Cs @} [\text{Da}])))

have g-def: \( (\Delta \Gamma ++ \Delta \Gamma') = (\Delta \Gamma' ++ \Delta \Gamma) \)
and g2-def: \( \Delta \Gamma' = \Delta \Gamma' \) by auto
from A-1-2[of - g-def g2-def - - ex] e0-typ3 r-inv k \( \Pi \) mtype-mbody-length[of mt mb]
obtain \( E' \) where \( e'\)-typ: \( CT; \Gamma \vdash \text{subs} [(xs@[this])[\to](\text{ds @} [\text{New Ca es}])] e : E' \)
and \( E'\)-sub-E: \( CT \vdash E' <: E \) by force
moreover from \( e'\)-typ \( \Pi \) mtype-mbody-length[of mt mb]
have \( CT; \Gamma \vdash \text{subs} [xs[\to]ds, this\to (\text{New Ca es})] e : E' \)
by (auto simp only: map-shuffle)
moreover from \( E'\)-sub-E E-sub-C have \( CT \vdash E' <: C \) by (rule subtyping.s-trans)
ultimately show \( \text{case using} \ r-inv k \) by auto

next
case \( r\)-cast \( CT \) \( Ca \) \( D \) \( es \)
then obtain \( Ca' \)
where \( C = D \)
and \( CT; \Gamma \vdash \text{New Ca es} : \text{Ca'} \) by (auto elim: typing.cases)
thus \( \text{case using} \ r\)-cast \( \) by (auto elim: typing.cases)

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

using \textit{assms}

proof \textit{induct}

\textbf{case} (\textit{rs-refl} CT e C) \textbf{thus} \textit{?case} by (auto simp add:subtyping.s-refl)

\textbf{next}

\textbf{case} (\textit{rs-trans} CT e e' e"

\textbf{hence} \textit{e-typ}: CT; \Gamma \vdash e : C

\textbf{and} \textit{e-step}: CT \vdash e \to e'

\textbf{and} \textit{ct-ok}: CT OK

\textbf{and} \textit{IH}: \forall D. \left[ CT; \Gamma \vdash e' : D; CT OK \right] \implies \exists E. CT; \Gamma \vdash e'" : E \land CT \vdash E <: D

\textbf{by} auto

\textbf{from} Thm-2-4-1[OF \textit{e-step ct-ok e-typ} \textit{D-sub-C}]

\textbf{obtain} D where \textit{E-sub-D} \textbf{by} auto

\textbf{moreover} \textbf{from} s-trans[OF \textit{E-sub-D D-sub-C}]

\textbf{have} CT \vdash E <: D by auto

\textbf{ultimately} \textbf{show} \textit{?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.

\textbf{theorem} Thm-2-4-2-1:

\textbf{assumes} CT; empty \vdash e : C

\textbf{and} FieldProj (\textit{New} C0 es) fi \in \textit{subexprs} \(e\)

\textbf{shows} \exists \textit{xs e0}. \textit{mbody}(CT, C0) = Ct \land \textit{lookup} Cf (\lambda fd. (vdName fd = fi)) = Some fDef

\textbf{proof} –

\textbf{obtain} Ci where CT; empty \vdash (FieldProj (\textit{New} C0 es) fi) : Ci

\textbf{using} \textit{assms} by (force simp add:subexpr-typing)

\textbf{then obtain} Cf fDef C0'

\textbf{where} CT; empty \vdash (\textit{New} C0 es) : C0'

\textbf{and} fields(CT, C0') = Cf

\textbf{and} \textit{lookup} Cf (\lambda fd. (vdName fd = fi)) = Some fDef

\textbf{by} (auto elim:typing.cases)

\textbf{thus} \textit{?thesis} by (auto elim:typing.cases)

qed

\textbf{lemma} Thm-2-4-2-2:

\textbf{fixes} es ds :: \textit{exp list}

\textbf{assumes} CT; empty \vdash e : C

\textbf{and} MethodInvk (\textit{New} C0 es) m ds \in \textit{subexprs} \(e\)

\textbf{shows} \exists xs e0. \textit{mbody}(CT, C0) = xs . e0 \land length xs = length ds

\textbf{proof} –
obtain $D$ where $CT;empty \vdash MethodInvk (New C0 es) m ds : D$
using assms by (force simp add:subexpr-typing)
then obtain $C0' Cs$
  where $CT;empty \vdash (New C0 es) : C0'$
  and $m\text{-mtype}(CT,m,C0') = Cs \rightarrow D$
  and $\text{length } ds = \text{length } Cs$
  by (auto elim:typing.cases)
with $m\text{-type-mbody}(OF m)$ show $\text{thesis}$ by (force elim:typing.cases)
qed

lemma closed-subterm-split:
  assumes $CT;\Gamma \vdash e : C$ and $\Gamma = \text{empty}$
  shows $(\exists C0 es fi. (\text{FieldProj (New C0 es) fi}) \in \text{subexprs}(e))$
  $\lor (\exists C0 es m ds. (\text{MethodInvk (New C0 es) m ds}) \in \text{subexprs}(e))$
  $\lor (\exists C0 D es. (\text{Cast D (New C0 es)}) \in \text{subexprs}(e))$
  $\lor \text{val}(e) \in \{?F e \lor ?M e \lor ?C e \lor ?V e \lor ?IH e\}$
using assms
proof (induct $CT \Gamma e C$ rule:typing-induct)
  case 1 thus $\text{thesis}$ using assms by auto
next
  case (2 C CT Γ x) thus $\text{thesis}$ using assms by auto
next
  case (3 C C t Cf Ci Γ e0 fDef fi)
  have $s1 : e0 \in \text{subexprs(FieldProj e0 fi)}$ by(auto simp add:isubexprs.intros)
  from $s1$ have $\text{?IH} e0$ by auto
  moreover
  \{ assume $?F e0$
  then obtain $C0 es fi' where s2: FieldProj (New C0 es) fi' \in \text{subexprs}(e0)$ by auto
  \}
  moreover \{
  assume $?M e0$
  then obtain $C0 es m ds where s2: MethodInvk (New C0 es) m ds \in \text{subexprs}(e0)$ by auto
  \}
  moreover \{
  assume $?C e0$
  then obtain $C0 D es where s2: Cast D (New C0 es) \in \text{subexprs}(e0)$ 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 $\text{thesis}$ by (force intro:isubexprs.intros)
  \}
  ultimately show $\text{thesis}$ by blast
next
  case (4 C C0 CT Cs Ds Γ e0 es m)
  have $s1 : e0 \in \text{subexprs(MethodInvk e0 m es)}$ by(auto simp add:isubexprs.intros)
from \(4\) have \(\ ?IH\ e_0\) by \(\text{auto}\)

moreover

\{\ 
assume \(\ ?F\ e_0\)
then obtain \(C_0\ es\ fi\ where\ s_2: FieldProj\ (\text{New}\ C_0\ es)\ fi\in\subexprs(e_0)\) by \(\text{auto}\)
\}

moreover \{
assume \(\ ?M\ e_0\)
then obtain \(C_0\ es'\ m'\ ds\ where\ s_2: MethodInvk\ (\text{New}\ C_0\ es')\ m'\ ds\in\subexprs(e_0)\) by \(\text{auto}\)
\}

moreover \{
assume \(\ ?C\ e_0\)
then obtain \(C_0\ D\ es\ where\ s_2: Cast\ D\ (\text{New}\ C_0\ es)\in\subexprs(e_0)\) by \(\text{auto}\)
\}

moreover \{
assume \(\ ?V\ e_0\)
then obtain \(\ ?C\ e_0\ as\ es'\ where\ e_0 = (\text{New}\ C_0\ es')\ and\ \vals(es')\ by\ (\text{force elim:val.cases})\)
\}

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

next

\text{proof\ (induct\ es\ Cs\ rule:list-induct2)}
\text{case Nil\ thus\ ?Q\ []\ by\ (auto\ intro:vals-val.intros)}

\text{next}
\text{case\ (Cons\ h\ t\ Ch\ Ct)}

\text{with } 5\ \text{have h-typs}:\ CT;\Gamma\vdash\ (h\#t):\ (Ch\#Ct)

and \(\ ?IH: \bigwedge\ i.\ [i < length\ (h\#t);\ CT;\Gamma\vdash\ ((h\#t)i):\ ((Ch\#Ct)i);\ \Gamma = empty]\ \Longrightarrow\ ?IH\ ((h\#t)i)\)

and \(\ G-def:\ \Gamma = empty\) by \(\text{auto}\)

from h-typs have
\(\ h\#\text{typs}\):\ CT;\Gamma\vdash\ (h\#t)\#0\ :\ (Ch\#Ct)\#0\)

and \(\ t\#\text{typs}\):\ 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 \(\text{auto}\)
from \(\ ?IH\ (OF\ s\#i)\)\ have \(\ ?I\ < length\ t;\ CT;\Gamma\vdash\ (t\#i):\ (Ct\#i);\ \Gamma = empty]\ \Longrightarrow\ ?IH\ ((t\#i))\ by\ \text{auto}\ \}

with t-typs have \(\ ?Q\ t\ using\ \text{Cons}\ by\ \text{auto}\)
moreover {  
  assume ∃i < length t. (?F (t!i)) ∨ (?M (t!i)) ∨ ?C (t!i))  
  then obtain i  
    where i < length t  
    and ?F (t!i) ∨ ?M (t!i) ∨ ?C (t!i) by force  
  hence (Suc i < length (h#t)) ∧ (?F ((h#t)!(Suc i)) ∨ ?M ((h#t)!(Suc i))  
    ∨ ?C ((h#t)!(Suc i))) by auto  
  hence ∃i < length (h#t). (?F ((h#t)!)i) ∨ ?M ((h#t)!)i) ∨ ?C ((h#t)!)i))

...  
  hence ?Q (h#t) by auto
} moreover {  
  assume v-t: vals(t)  
  from OIH[OF - h-typ G-def] have ?IH h by auto  
moreover {  
  assume ?F h ∨ ?M h ∨ ?C h  
  hence ?F ((h#t)!)0) ∨ ?M ((h#t)!)0) ∨ ?C ((h#t)!)0) by auto  
  hence ?Q (h#t) by force
} moreover {  
  assume ?V h  
  with v-t have vals((h#t)) by (force intro:vals-val.intros)  
  hence ?Q(h#t) by auto
} ultimately have ?Q(h#t) by blast  
} ultimately show ?Q(h#t) by blast

qed

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

38
hence ?case by auto

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

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

next
  case (S C CT D Γ e0)
  have s1: e0 ∈ subexprs(Cast C e0) by(auto simp add:isubexprs.intros)
  from s1 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;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 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;empty ⊢ e1 : C1
  and C1-sub-C: CT ⊢ C1 <: C by auto

40
from e1-typ have \((\exists C0 \text{ es } f). (\text{FieldProj } (\text{New } C0 \text{ es } f) \in \text{subexprs}(e1))\)
\(\lor (\exists C0 \text{ es } m \text{ ds}. (\text{MethodInvk } (\text{New } C0 \text{ es } m \text{ ds}) \in \text{subexprs}(e1))\)
\(\lor (\exists C0 D \text{ es}. (\text{Cast } D (\text{New } C0 \text{ es})) \in \text{subexprs}(e1))\)
\(\lor \text{val}(e1)\) \(\text{is } ?F e1 \lor ?M e1 \lor ?C e1 \lor ?V e1\) by (simp add: closed-subterm-split)
moreover
\{ assume ?F e1
  then obtain C0 es fi where fp: FieldProj (New C0 es) fi \in \text{subexprs}(e1) by auto
moreover
  from e1-typ have \(\text{Ci where } CT;\text{empty } \vdash \text{FieldProj } (\text{New } C0 \text{ es } fi : Ci \text{ using e1-typ by (force simp add: subexpr-typing)}\)
then obtain C0\text{'} where new-typ: CT;\text{empty } \vdash \text{New } C0 \text{ es } : C0\text{'} by (force elim: typing.cases)
then obtain C0\text{'} where new-typ: \(\text{CT;empty } \vdash \text{New } C0 \text{ es } : C0\text{'}\) by (force elim: typing.cases)
\(\text{hence } C0 = C0\text{'}\) by (auto elim: typing.cases)
with new-typ obtain Df where fi1: fields(CT,C0) = Df and lens: length es = length Df by(auto elim: typing.cases)
from Thm-2-4-2-1[OF e1-typ fp] obtain Cf fDef
  where f2: fields(CT,C0) = Cf
  and lkup: lookup Cf (\(\lambda fd. \text{vdName } fd = fi\)) = Some(fDef) by force
moreover from fields-functional[OF f1 ct-ok f2] lens have length es = length Cf
moreover from lookup-index[OF lkup] obtain i where
  i<length Cf
  and fDef = Cf \(\vDash i\)
  and (length Cf = length es) \(\rightarrow\) lookup2 Cf es (\(\lambda fd. \text{vdName } fd = fi\)) = Some((es\(\vDash i\)) by auto
ultimately have lookup2 Cf es (\(\lambda fd. \text{vdName } fd = fi\)) = Some (es\(\vDash i\)) by auto
with f2 have CT \(\vdash\) FieldProj(New C0 es) fi \(\rightarrow\) (es\(\vDash i\)) by(auto intro: reduction.intros)
with fp have \(\exists e2\). CT \(\vdash\) e1 \(\rightarrow\) e2 by(simp add: subexpr-reduct)
with no-step have ?thesis by auto
\} moreover \{ assume ?M e1
then obtain C0 es m ds where mi: MethodInvk (New C0 es) m ds \in \text{subexprs}(e1) by auto
then obtain D where CT;\text{empty } \vdash \text{MethodInvk } (\text{New } C0 \text{ es } m \text{ ds } : D \text{ using e1-typ by (force simp add: subexpr-typing)}\)
then obtain C0\text{'} Es E
  where m-typ: CT;\text{empty } \vdash \text{New } C0 \text{ es } : C0\text{'}
  and mtype(CT,m,C0\text{'}) = Es \rightarrow E
  and length ds = length Es
by (auto elim: typing.cases)
from Thm-2-4-2-2[OF e1-typ mi] obtain xs e0 where mb: mbody(CT, m, C0) = xs . e0 and length xs = length ds by auto
hence CT \(\vdash\) (MethodInvk (New C0 es) m ds) \(\rightarrow\) (subs[\(\text{xs}\mapsto\text{ds}\),this\(\rightarrow\)(New C0 es)e0]) by(auto simp add: reduction.intros)
with mi have \(\exists e2\). CT \(\vdash\) e1 \(\rightarrow\) e2 by(simp add: subexpr-reduct)
with no-step have ?thesis by auto
\} moreover \{ assume ?C e1
then obtain C0 D es where c-def: Cast D (New C0 es) \in \text{subexprs}(e1) by
then obtain \( D' \) where \( CT;\emptyset \vdash \text{Cast } D \) \( \text{New } C0 \ es \) : \( D' \) using \( e1\text{-typ} \)
by \((\text{force simp add:subexpr-typing})\)
then obtain \( C0' \) where \( \text{new-typ: } CT;\emptyset \vdash \text{New } C0 \ es \) : \( C0' \) and \( D\text{-eq-D'}\);
\( D = D' \) by \((\text{auto elim:typing.cases})\)
hence \( C0\text{-eq-C0'}\): \( C0 = C0' \) by \((\text{auto elim:typing.cases})\)
hence \(?thesis \) proof\((\text{cases } CT \vdash C0 <: D)\)
case \( \text{True} \)
hence \( CT \vdash \text{Cast } D \) \( \text{New } C0 \ es \) \( \rightarrow \) \( \text{New } C0 \ es \) by \((\text{auto simp add:reduction.intros})\)
with \( \text{c-def} \) have \( \exists e2. CT \vdash e1 \rightarrow e2 \) by \((\text{simp add:subexpr-reduct})\)
with \( \text{no-step} \) show \(?thesis \) by \( \text{auto} \)
next
case \( \text{False} \)
with \( \text{c-def} \) show \(?thesis \) by \( \text{auto} \)
qed

moreover \{ 
assume \(?V \) \( e1 \)
hence \(?thesis \) using \( \text{assms} \) by \((\text{auto simp add:Cor-2-4-1-multi})\)
\}
ultimately show \(?thesis \) by \( \text{blast} \)
qed

end

theory \text{Execute}
imports \text{FJSound}
begin

4 Executing FeatherweightJava programs

We execute FeatherweightJava programs using the predicate compiler.

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

\textbf{thm} \( \text{subtyping.equation} \)
The reduction relation requires that we inverse the \( \text{op @ function} \). Therefore,
we define a new predicate append and derive introduction rules.

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

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

\textbf{lemma}[\text{code-pred-intro}]: \text{append} \ \text{xs} \ \text{ys} \ \text{zs} \ \Rightarrow \ \text{append} \ (\text{x#xs}) \ \text{ys} \ (\text{x#zs})
\textbf{unfolding} \ \text{append-def} \ \text{by} \ \text{simp}

With this at hand, we derive new introduction rules for the reduction relation:
lemma re-invk-arg': $\forall e_i e_i' e_i'' e \in E, \exists e_i'' e' e'' \in E$ such that
\[ CT \vdash e_i \rightarrow e_i' \Rightarrow \append{e_i}{e_i'} \Rightarrow \append{e_i''}{e''} \Rightarrow \MethodInvk{e}{m}{e} \rightarrow \MethodInvk{e'}{m}{e''} \]
unfolding append-def by simp (rule reduction.intros(6))

lemma re-new-arg': $\forall e_i e_i' e_i'' e \in E$ such that
\[ CT \vdash e_i \rightarrow e_i' \Rightarrow \append{e_i}{e_i'} \Rightarrow \append{e_i''}{e''} \Rightarrow \NewC{e} \rightarrow \NewC{e'} \]
unfolding append-def by simp (rule reduction.intros(7))

lemmas [code-pred-intro] = reduction.intros(1-5)
re-invk-arg' re-new-arg' reduction.intros(8)

code-pred (modes: $i \Rightarrow i \Rightarrow i \Rightarrow bool, i \Rightarrow i \Rightarrow o \Rightarrow bool$ as reduce)
reduction
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 rc-field
    with reduction(4) show thesis by fastforce
next
  case rc-invk-recv
    with reduction(5) show thesis by fastforce
next
  case rc-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:

abbreviation $A :: className$
where $A == Suc 0$

abbreviation $B :: className$
where $B == 2$

abbreviation $cPair :: className$
where $cPair == 3$

definition $classA-Def :: classDef$
where
$\begin{array}{l}
classA-Def = \langle cName = A, cSuper = Object, cFields = [], cConstructor = \\
\langle kName = A, kParams = [], kSuper = [], kInits = [], cMethods = [] \rangle \rangle
\end{array}$

definition $classB-Def = \langle cName = B, cSuper = Object, cFields = [], cConstructor = \\
\langle kName = B, kParams = [], kSuper = [], kInits = [], cMethods = [] \rangle \rangle$

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
$\begin{array}{l}
classPair-Def = \langle cName = cPair, cSuper = Object, \\
cFields = [\langle vdName = ffst, vdType = Object \rangle, \langle vdName = fsnd, vdType = Object \rangle], \\
cConstructor = \langle kName = cPair, kParams = [\langle vdName = ffst, vdType = Object \rangle, \langle vdName = fsnd, vdType = Object \rangle], kSuper = [], kInits = [ffst, fsnd] \rangle, \\
cMethods = [\langle mReturn = cPair, mName = setfst, mParams = [\langle vdName = newfst, vdType = Object \rangle], \\
mBody = New cPair [Var newfst, FieldProj (Var this) fsnd] \rangle] \rangle
\end{array}$
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
