

A Theory of Featherweight Java in Isabelle/HOL

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

Abstract

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

Contents

1	FJDefs: Basic Definitions	2
1.1	Syntax	2
1.1.1	Type definitions	2
1.1.2	Constants	3
1.1.3	Expressions	3
1.1.4	Methods	3
1.1.5	Constructors	3
1.1.6	Classes	3
1.1.7	Class Tables	4
1.2	Sub-expression Relation	4
1.3	Values	4
1.4	Substitution	4
1.5	Lookup	5
1.6	Variable Definition Accessors	5
1.7	Subtyping Relation	5
1.8	fields Relation	6
1.9	mtype Relation	6
1.10	mbody Relation	7
1.11	Typing Relation	7
1.12	Method Typing Relation	9
1.13	Class Typing Relation	10
1.14	Class Table Typing Relation	10
1.15	Evaluation Relation	11

2	FJAux: Auxiliary Lemmas	12
2.1	Non-FJ Lemmas	12
2.1.1	Lists	12
2.1.2	Maps	12
2.2	FJ Lemmas	12
2.2.1	Substitution	12
2.2.2	Lookup	12
2.2.3	Functional	13
2.2.4	Subtyping and Typing	14
2.2.5	Sub-Expressions	15
3	FJSound: Type Soundness	16
3.1	Method Type and Body Connection	16
3.2	Method Types and Field Declarations of Subtypes	16
3.3	Substitution Lemma	17
3.4	Weakening Lemma	17
3.5	Method Body Typing Lemma	17
3.6	Subject Reduction Theorem	17
3.7	Multi-Step Subject Reduction Theorem	18
3.8	Progress	18
3.9	Type Soundness Theorem	18
4	Executing FeatherweightJava programs	19
4.1	A simple example	20

1 FJDefs: Basic Definitions

```
theory FJDefs
imports Main
begin
```

1.1 Syntax

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

1.1.1 Type definitions

```
type-synonym varName = nat
type-synonym methodName = nat
type-synonym className = nat
```

```
record varDef =  
  vdName :: varName  
  vdType :: className  
type-synonym varCtx = varName  $\rightarrow$  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 $classTable = className \rightarrow classDef$

1.2 Sub-expression Relation

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

inductive-set

$isubexprs :: (exp * exp) set$
and $isubexprs' :: [exp,exp] \Rightarrow bool \ (- \in isubexprs'(-) [80,80] 80)$
where
 $e' \in isubexprs(e) \equiv (e',e) \in isubexprs$
 $| se-field \quad : e \in isubexprs(FieldProj\ e\ fi)$
 $| se-invkrecurv : e \in isubexprs(MethodInvk\ e\ m\ es)$
 $| se-invkarq \quad : \llbracket ei \in set\ es \rrbracket \Longrightarrow ei \in isubexprs(MethodInvk\ e\ m\ es)$
 $| se-newarg \quad : \llbracket ei \in set\ es \rrbracket \Longrightarrow ei \in isubexprs(New\ C\ es)$
 $| se-cast \quad : e \in isubexprs(Cast\ C\ e)$

abbreviation

$subexprs :: [exp,exp] \Rightarrow bool \ (- \in subexprs'(-) [80,80] 80)$ **where**
 $e' \in subexprs(e) \equiv (e',e) \in isubexprs^*$

1.3 Values

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

inductive

$vals :: [exp\ list] \Rightarrow bool \ (vals'(-) [80] 80)$
and $val :: [exp] \Rightarrow bool \ (val'(-) [80] 80)$
where
 $vals-nil : vals(\llbracket \rrbracket)$
 $| vals-cons : \llbracket val(vh); vals(vt) \rrbracket \Longrightarrow vals((vh \# vt))$
 $| val : \llbracket vals(vs) \rrbracket \Longrightarrow val(New\ C\ vs)$

1.4 Substitution

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

primrec $subst :: (varName \rightarrow exp) \Rightarrow exp \Rightarrow exp$
and $subst-list1 :: (varName \rightarrow exp) \Rightarrow exp\ list \Rightarrow exp\ list$
and $subst-list2 :: (varName \rightarrow exp) \Rightarrow exp\ list \Rightarrow exp\ list$ **where**
 $subst\ \sigma\ (Var\ x) = \text{case } (\sigma(x)) \text{ of } None \Rightarrow (Var\ x) \mid Some\ p \Rightarrow p$
 $| subst\ \sigma\ (FieldProj\ e\ f) = FieldProj\ (subst\ \sigma\ e)\ f$
 $| subst\ \sigma\ (MethodInvk\ e\ m\ es) = MethodInvk\ (subst\ \sigma\ e)\ m\ (subst-list1\ \sigma\ es)$
 $| subst\ \sigma\ (New\ C\ es) = New\ C\ (subst-list2\ \sigma\ es)$

```

| subst  $\sigma$  (Cast C e) =          Cast C (subst  $\sigma$  e)
| subst-list1  $\sigma$  [] = []
| subst-list1  $\sigma$  (h # t) = (subst  $\sigma$  h) # (subst-list1  $\sigma$  t)
| subst-list2  $\sigma$  [] = []
| subst-list2  $\sigma$  (h # t) = (subst  $\sigma$  h) # (subst-list2  $\sigma$  t)

```

abbreviation

```

subst-syn :: [exp list] => [varName list] => [exp] => exp
  ('['-/']- [80,80,80] 80) where
  (ds/xs)e  $\equiv$  subst (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  $\equiv$  map (subst (map-upds Map.empty xs ds)) es

```

1.5 Lookup

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

```

primrec lookup :: 'a list => ('a => bool) => 'a option
where

```

```

  lookup [] P = None
| lookup (h#t) P = (if P h then Some h else lookup t P)

```

```

primrec lookup2 :: 'a list => 'b list => ('a => bool) => 'b option
where

```

```

  lookup2 [] l2 P = None
| lookup2 (h1#t1) l2 P = (if P h1 then Some(hd l2) else lookup2 t1 (tl l2) P)

```

1.6 Variable Definition Accessors

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

definition

```

varDefs-names :: varDef list => varName list where
varDefs-names = map vdName

```

definition

```

varDefs-types :: varDef list => className list where
varDefs-types = map vdType

```

1.7 Subtyping Relation

The subtyping relation, written $CT \vdash C <: D$ is just the reflexive and transitive closure of the immediate subclass relation. (For the sake of simplicity,

we define subtyping directly instead of using the reflexive and transitive closure operator.) The subtyping relation is extended to lists of classes, written $CT \vdash + Cs <: Ds$.

inductive

$subtyping :: [classTable, className, className] \Rightarrow bool$ ($- \vdash - <: -$ [80,80,80] 80)

where

$s-refl : CT \vdash C <: C$
 $| s-trans : \llbracket CT \vdash C <: D; CT \vdash D <: E \rrbracket \Longrightarrow CT \vdash C <: E$
 $| s-super : \llbracket CT(C) = Some(CDef); cSuper CDef = D \rrbracket \Longrightarrow CT \vdash C <: D$

abbreviation

$neg-subtyping :: [classTable, className, className] \Rightarrow bool$ ($- \vdash - \neg <: -$ [80,80,80] 80)

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

inductive

$subtypings :: [classTable, className list, className list] \Rightarrow bool$ ($- \vdash + - <: -$ [80,80,80] 80)

where

$ss-nil : CT \vdash + [] <: []$
 $| ss-cons : \llbracket CT \vdash C0 <: D0; CT \vdash + Cs <: Ds \rrbracket \Longrightarrow CT \vdash + (C0 \# Cs) <: (D0 \# Ds)$

1.8 fields Relation

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

inductive

$fields :: [classTable, className, varDef list] \Rightarrow bool$ ($fields'(-,-) = -$ [80,80,80] 80)

where

$f-obj$:
 $fields(CT, Object) = []$
 $| f-class$:
 $\llbracket CT(C) = Some(CDef); cSuper CDef = D; cFields CDef = Cf; fields(CT, D) = Dg; DgCf = Dg @ Cf \rrbracket$
 $\Longrightarrow fields(CT, C) = DgCf$

1.9 mtype Relation

The `mtype` relation, written $mtype(CT, m, C) = Cs \rightarrow C_0$ relates a class C , method name m , and the arrow type $Cs \rightarrow C_0$. It either returns the type of the declaration of m in C , if any such declaration exists, and otherwise returning the type of m from C 's superclass.

inductive

$mtype :: [classTable, methodName, className, className list, className] \Rightarrow bool$
 $(mtype'(-,-,-) = - \rightarrow - [80,80,80,80] 80)$

where

mt-class:
 $\llbracket CT(C) = Some(CDef);$
 $lookup (cMethods CDef) (\lambda md.(mName md = m)) = Some(mDef);$
 $varDefs-types (mParams mDef) = Bs;$
 $mReturn mDef = B \rrbracket$
 $\Longrightarrow mtype(CT,m,C) = Bs \rightarrow B$

| *mt-super:*
 $\llbracket CT(C) = Some (CDef);$
 $lookup (cMethods CDef) (\lambda md.(mName md = m)) = None;$
 $cSuper CDef = D;$
 $mtype(CT,m,D) = Bs \rightarrow B \rrbracket$
 $\Longrightarrow mtype(CT,m,C) = Bs \rightarrow B$

1.10 mbody Relation

The **mtype** relation, written $mbody(CT, m, C) = xs.e_0$ relates a class C , method name m , and the names of the parameters xs and the body of the method e_0 . 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

$mbody :: [classTable, methodName, className, varName list, exp] \Rightarrow bool$ ($mbody'(-,-,-)$
 $= - . - [80,80,80,80] 80)$

where

mb-class:
 $\llbracket CT(C) = Some(CDef);$
 $lookup (cMethods CDef) (\lambda md.(mName md = m)) = Some(mDef);$
 $varDefs-names (mParams mDef) = xs;$
 $mBody mDef = e \rrbracket$
 $\Longrightarrow mbody(CT,m,C) = xs . e$

| *mb-super:*
 $\llbracket CT(C) = Some(CDef);$
 $lookup (cMethods CDef) (\lambda md.(mName md = m)) = None;$
 $cSuper CDef = D;$
 $mbody(CT,m,D) = xs . e \rrbracket$
 $\Longrightarrow mbody(CT,m,C) = xs . e$

1.11 Typing Relation

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

inductive

$typings :: [classTable, varCtx, exp list, className list] \Rightarrow bool$ (-;- \vdash - - -
 $[80,80,80,80] 80)$
and $typing :: [classTable, varCtx, exp, className] \Rightarrow bool$ (-;- \vdash - - - $[80,80,80,80]$
 $80)$
where
 $ts-nil : CT; \Gamma \vdash [] : []$

| $ts-cons$:
 $\llbracket CT; \Gamma \vdash e0 : C0; CT; \Gamma \vdash es : Cs \rrbracket$
 $\Longrightarrow CT; \Gamma \vdash (e0 \# es) : (C0 \# Cs)$

| $t-var$:
 $\llbracket \Gamma(x) = Some\ C \rrbracket \Longrightarrow CT; \Gamma \vdash (Var\ x) : C$

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

| $t-invok$:
 $\llbracket CT; \Gamma \vdash e0 : C0;$
 $mtype(CT, m, C0) = Ds \rightarrow C;$
 $CT; \Gamma \vdash es : Cs;$
 $CT \vdash Cs <: Ds;$
 $length\ es = length\ Ds \rrbracket$
 $\Longrightarrow CT; \Gamma \vdash MethodInvk\ e0\ m\ es : C$

| $t-new$:
 $\llbracket fields(CT, C) = Df;$
 $length\ es = length\ Df;$
 $varDefs-types\ Df = Ds;$
 $CT; \Gamma \vdash es : Cs;$
 $CT \vdash Cs <: Ds \rrbracket$
 $\Longrightarrow CT; \Gamma \vdash New\ C\ es : C$

| $t-ucast$:
 $\llbracket CT; \Gamma \vdash e0 : D;$
 $CT \vdash D <: C \rrbracket$
 $\Longrightarrow CT; \Gamma \vdash Cast\ C\ e0 : C$

| $t-dcast$:
 $\llbracket CT; \Gamma \vdash e0 : D;$
 $CT \vdash C <: D; C \neq D \rrbracket$
 $\Longrightarrow CT; \Gamma \vdash Cast\ C\ e0 : C$

| $t-scast$:
 $\llbracket CT; \Gamma \vdash e0 : D;$

$$\begin{aligned}
& CT \vdash C \neg<: D; \\
& CT \vdash D \neg<: C \] \\
\implies & CT; \Gamma \vdash \text{Cast } C \ e0 : C
\end{aligned}$$

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*:

$$\begin{aligned}
& \text{assumes } CT; \Gamma \vdash e : C \text{ (is ?T)} \\
& \text{and } \bigwedge C \ CT \ \Gamma \ x. \ \Gamma \ x = \text{Some } C \implies P \ CT \ \Gamma \ (\text{Var } x) \ C \\
& \text{and } \bigwedge C0 \ CT \ Cf \ Ci \ \Gamma \ e0 \ fDef \ fi. \ \llbracket CT; \Gamma \vdash e0 : C0; P \ CT \ \Gamma \ e0 \ C0; \text{fields}(CT, C0) \\
& = Cf; \text{lookup } Cf \ (\lambda fd. \text{vdName } fd = fi) = \text{Some } fDef; \text{vdType } fDef = Ci \rrbracket \implies P \\
& CT \ \Gamma \ (\text{FieldProj } e0 \ fi) \ Ci \\
& \text{and } \bigwedge C \ C0 \ CT \ Cs \ Ds \ \Gamma \ e0 \ es \ m. \ \llbracket CT; \Gamma \vdash e0 : C0; P \ CT \ \Gamma \ e0 \ C0; \text{mtype}(CT, m, C0) \\
& = Ds \rightarrow C; CT; \Gamma \vdash + es : Cs; \bigwedge i. \ \llbracket i < \text{length } es \rrbracket \implies P \ CT \ \Gamma \ (\text{es!}i) \ (Cs!i); \\
& CT \vdash + Cs <: Ds; \text{length } es = \text{length } Ds \rrbracket \implies P \ CT \ \Gamma \ (\text{MethodInvk } e0 \ m \ es) \ C \\
& \text{and } \bigwedge C \ CT \ Cs \ Df \ Ds \ \Gamma \ es. \ \llbracket \text{fields}(CT, C) = Df; \text{length } es = \text{length } Df; \\
& \text{varDefs-types } Df = Ds; CT; \Gamma \vdash + es : Cs; \bigwedge i. \ \llbracket i < \text{length } es \rrbracket \implies P \ CT \ \Gamma \\
& (\text{es!}i) \ (Cs!i); CT \vdash + Cs <: Ds \rrbracket \implies P \ CT \ \Gamma \ (\text{New } C \ es) \ C \\
& \text{and } \bigwedge C \ CT \ D \ \Gamma \ e0. \ \llbracket CT; \Gamma \vdash e0 : D; P \ CT \ \Gamma \ e0 \ D; CT \vdash D <: C \rrbracket \implies P \ CT \\
& \Gamma \ (\text{Cast } C \ e0) \ C \\
& \text{and } \bigwedge C \ CT \ D \ \Gamma \ e0. \ \llbracket CT; \Gamma \vdash e0 : D; P \ CT \ \Gamma \ e0 \ D; CT \vdash C <: D; C \neq D \rrbracket \\
& \implies P \ CT \ \Gamma \ (\text{Cast } C \ e0) \ C \\
& \text{and } \bigwedge C \ CT \ D \ \Gamma \ e0. \ \llbracket CT; \Gamma \vdash e0 : D; P \ CT \ \Gamma \ e0 \ D; CT \vdash C \neg<: D; CT \vdash D \\
& \neg<: C \rrbracket \implies P \ CT \ \Gamma \ (\text{Cast } C \ e0) \ C \\
& \text{shows } P \ CT \ \Gamma \ e \ C \text{ (is ?P)} \\
& \langle \text{proof} \rangle
\end{aligned}$$

1.12 Method Typing Relation

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

inductive

$$\text{method-typing} :: [\text{classTable}, \text{methodDef}, \text{className}] \Rightarrow \text{bool} \ (- \vdash - \text{OK IN } - \text{ [80,80,80] } 80)$$

where

m-typing:

$$\begin{aligned}
& \llbracket CT(C) = \text{Some}(CDef); \\
& \quad cName \ CDef = C; \\
& \quad cSuper \ CDef = D; \\
& \quad mName \ mDef = m; \\
& \quad \text{lookup } (cMethods \ CDef) \ (\lambda md. (mName \ md = m)) = \text{Some}(mDef); \\
& \quad mReturn \ mDef = C0; mParams \ mDef = Cxs; mBody \ mDef = e0; \\
& \quad \text{varDefs-types } Cxs = Cs; \\
& \quad \text{varDefs-names } Cxs = xs; \\
& \quad \Gamma = (\text{map-upds } \text{Map.empty } xs \ Cs)(\text{this} \mapsto C);
\end{aligned}$$

$$\begin{aligned}
& CT; \Gamma \vdash e0 : E0; \\
& CT \vdash E0 <: C0; \\
& \forall Ds D0. (mtype(CT, m, D) = Ds \rightarrow D0) \longrightarrow (Cs=Ds \wedge C0=D0) \] \\
\implies & CT \vdash mDef OK IN C
\end{aligned}$$

inductive

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

where

ms-nil :
 $CT \vdash \] OK IN C$

| *ms-cons* :
 $\ [CT \vdash m OK IN C;$
 $CT \vdash ms OK IN C \]$
 $\implies CT \vdash (m \# ms) OK IN C$

1.13 Class Typing Relation

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

inductive

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

where

t-class: $\ [cName CDef = C;$
 $cSuper CDef = D;$
 $cConstructor CDef = KDef;$
 $cMethods CDef = M;$
 $kName KDef = C;$
 $kParams KDef = (Dg@Cf);$
 $kSuper KDef = varDefs-names Dg;$
 $kInits KDef = varDefs-names Cf;$
 $fields(CT, D) = Dg;$
 $CT \vdash M OK IN C \]$
 $\implies CT \vdash CDef OK$

1.14 Class Table Typing Relation

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

inductive

ct-typing :: classTable \Rightarrow bool (- OK 80)

where

ct-all-ok:
 $\ [Object \notin dom(CT);$
 $\ \forall C CDef. CT(C) = Some(CDef) \longrightarrow (CT \vdash CDef OK) \wedge (cName CDef = C) \]$
 $\implies CT OK$

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

reduction :: [classTable, exp, exp] \Rightarrow bool (- \vdash - \rightarrow - [80,80,80] 80)

where

r-field:

\llbracket fields(CT, C) = Cf ;
 lookup2 Cf es ($\lambda fd. (vdName\ fd = fi)$) = $Some(ei)$ \rrbracket
 $\Rightarrow CT \vdash FieldProj$ ($New\ C\ es$) $fi \rightarrow ei$

| *r-invok*:

\llbracket mbody(CT, m, C) = $xs . e0$;
 substs ((map-upds $Map.empty\ xs\ ds$)($this \mapsto (New\ C\ es)$)) $e0 = e0'$ \rrbracket
 $\Rightarrow CT \vdash MethodInvk$ ($New\ C\ es$) $m\ ds \rightarrow e0'$

| *r-cast*:

$\llbracket CT \vdash C <: D \rrbracket$
 $\Rightarrow CT \vdash Cast\ D$ ($New\ C\ es$) $\rightarrow New\ C\ es$

| *rc-field*:

$\llbracket CT \vdash e0 \rightarrow e0' \rrbracket$
 $\Rightarrow CT \vdash FieldProj\ e0\ f \rightarrow FieldProj\ e0'\ f$

| *rc-invok-recv*:

$\llbracket CT \vdash e0 \rightarrow e0' \rrbracket$
 $\Rightarrow CT \vdash MethodInvk\ e0\ m\ es \rightarrow MethodInvk\ e0'\ m\ es$

| *rc-invok-arg*:

$\llbracket CT \vdash ei \rightarrow ei' \rrbracket$
 $\Rightarrow CT \vdash MethodInvk\ e0\ m\ (el@ei\#er) \rightarrow MethodInvk\ e0\ m\ (el@ei'\#er)$

| *rc-new-arg*:

$\llbracket CT \vdash ei \rightarrow ei' \rrbracket$
 $\Rightarrow CT \vdash New\ C\ (el@ei\#er) \rightarrow New\ C\ (el@ei'\#er)$

| *rc-cast*:

$\llbracket CT \vdash e0 \rightarrow e0' \rrbracket$
 $\Rightarrow CT \vdash Cast\ C\ e0 \rightarrow Cast\ C\ e0'$

inductive

reductions :: [classTable, exp, exp] \Rightarrow bool (- \vdash - \rightarrow^* - [80,80,80] 80)

where

rs-refl: $CT \vdash e \rightarrow^* e$

| *rs-trans*: $\llbracket CT \vdash e \rightarrow e'; CT \vdash e' \rightarrow^* e'' \rrbracket \Rightarrow CT \vdash e \rightarrow^* e''$

end

2 FJAux: Auxiliary Lemmas

theory *FJAux* **imports** *FJDefs*
begin

2.1 Non-FJ Lemmas

2.1.1 Lists

lemma *mem-ith*:
 assumes $ei \in \text{set } es$
 shows $\exists el\ er. es = el@ei\#er$
 $\langle \text{proof} \rangle$

lemma *ith-mem*: $\bigwedge i. \llbracket i < \text{length } es \rrbracket \implies es!i \in \text{set } es$
 $\langle \text{proof} \rangle$

2.1.2 Maps

lemma *map-shuffle*:
 assumes $\text{length } xs = \text{length } ys$
 shows $[xs[\mapsto]ys, x \mapsto y] = [(xs@[x])[\mapsto](ys@[y])]$
 $\langle \text{proof} \rangle$

lemma *map-upds-index*:
 assumes $\text{length } xs = \text{length } As$
 and $[xs[\mapsto]As]x = \text{Some } Ai$
 shows $\exists i. (As!i = Ai)$
 $\wedge (i < \text{length } As)$
 $\wedge (\forall (Bs::'c \text{ list}). ((\text{length } Bs = \text{length } As)$
 $\longrightarrow ([xs[\mapsto]Bs] x = \text{Some } (Bs !i))))$
 $(\text{is } \exists i. ?P\ i\ xs\ As$
 is $\exists i. (?P1\ i\ As) \wedge (?P2\ i\ As) \wedge (\forall Bs::('c \text{ list}). (?P3\ i\ xs\ As\ Bs)))$
 $\langle \text{proof} \rangle$

2.2 FJ Lemmas

2.2.1 Substitution

lemma *subst-list1-eq-map-substs* :
 $\forall \sigma. \text{subst-list1 } \sigma\ l = \text{map } (\text{substs } \sigma)\ l$
 $\langle \text{proof} \rangle$

lemma *subst-list2-eq-map-substs* :
 $\forall \sigma. \text{subst-list2 } \sigma\ l = \text{map } (\text{substs } \sigma)\ l$
 $\langle \text{proof} \rangle$

2.2.2 Lookup

lemma *lookup-functional*:
 assumes $\text{lookup } l\ f = o1$

and $lookup\ l\ f = o2$
shows $o1 = o2$
 $\langle proof \rangle$

lemma *lookup-true*:
 $lookup\ l\ f = Some\ r \implies f\ r$
 $\langle proof \rangle$

lemma *lookup-hd*:
 $\llbracket length\ l > 0; f\ (l!0) \rrbracket \implies lookup\ l\ f = Some\ (l!0)$
 $\langle proof \rangle$

lemma *lookup-split*: $lookup\ l\ f = None \vee (\exists h. lookup\ l\ f = Some\ h)$
 $\langle proof \rangle$

lemma *lookup-index*:
assumes $lookup\ l1\ f = Some\ e$
shows $\bigwedge l2. \exists i < (length\ l1). e = l1!i \wedge ((length\ l1 = length\ l2) \longrightarrow lookup2\ l1\ l2\ f = Some\ (l2!i))$
 $\langle proof \rangle$

lemma *lookup2-index*:
 $\bigwedge l2. \llbracket lookup2\ l1\ l2\ f = Some\ e; length\ l1 = length\ l2 \rrbracket \implies \exists i < (length\ l2). e = (l2!i) \wedge lookup\ l1\ f = Some\ (l1!i)$
 $\langle proof \rangle$

lemma *lookup-append*:
assumes $lookup\ l\ f = Some\ r$
shows $lookup\ (l@l')\ f = Some\ r$
 $\langle proof \rangle$

lemma *method-typings-lookup*:
assumes *lookup-eq-Some*: $lookup\ M\ f = Some\ mDef$
and *M-ok*: $CT \vdash+ M\ OK\ IN\ C$
shows $CT \vdash mDef\ OK\ IN\ C$
 $\langle proof \rangle$

2.2.3 Functional

These lemmas prove that several relations are actually functions

lemma *mtype-functional*:
assumes $mtype(CT, m, C) = Cs \rightarrow C0$
and $mtype(CT, m, C) = Ds \rightarrow D0$
shows $Ds = Cs \wedge D0 = C0$
 $\langle proof \rangle$

lemma *mbody-functional*:
assumes *mb1*: $mbody(CT, m, C) = xs . e0$

and $mb2: mbody(CT, m, C) = ys . d0$
shows $xs = ys \wedge e0 = d0$
 $\langle proof \rangle$

lemma *fields-functional*:
assumes $fields(CT, C) = Cf$
and $CT OK$
shows $\bigwedge Cf'. \llbracket fields(CT, C) = Cf \rrbracket \implies Cf = Cf'$
 $\langle proof \rangle$

2.2.4 Subtyping and Typing

lemma *typings-lengths*: **assumes** $CT; \Gamma \vdash+ es:Cs$ **shows** $length\ es = length\ Cs$
 $\langle proof \rangle$

lemma *typings-index*:
assumes $CT; \Gamma \vdash+ es:Cs$
shows $\bigwedge i. \llbracket i < length\ es \rrbracket \implies CT; \Gamma \vdash (es!i) : (Cs!i)$
 $\langle proof \rangle$

lemma *subtypings-index*:
assumes $CT \vdash+ Cs <: Ds$
shows $\bigwedge i. \llbracket i < length\ Cs \rrbracket \implies CT \vdash (Cs!i) <: (Ds!i)$
 $\langle proof \rangle$

lemma *subtyping-append*:
assumes $CT \vdash+ Cs <: Ds$
and $CT \vdash C <: D$
shows $CT \vdash+ (Cs@[C]) <: (Ds@[D])$
 $\langle proof \rangle$

lemma *typings-append*:
assumes $CT; \Gamma \vdash+ es : Cs$
and $CT; \Gamma \vdash e : C$
shows $CT; \Gamma \vdash+ (es@[e]) : (Cs@[C])$
 $\langle proof \rangle$

lemma *ith-typing*: $\bigwedge Cs. \llbracket CT; \Gamma \vdash+ (es@(h\#t)) : Cs \rrbracket \implies CT; \Gamma \vdash h : (Cs!(length\ es))$
 $\langle proof \rangle$

lemma *ith-subtyping*: $\bigwedge Ds. \llbracket CT \vdash+ (Cs@(h\#t)) <: Ds \rrbracket \implies CT \vdash h <: (Ds!(length\ Cs))$
 $\langle proof \rangle$

lemma *subtypings-refl*: $CT \vdash+ Cs <: Cs$
 $\langle proof \rangle$

lemma *subtypings-trans*: $\bigwedge Ds\ Es. \llbracket CT \vdash\vdash Cs <: Ds; CT \vdash\vdash Ds <: Es \rrbracket \implies CT \vdash\vdash Cs <: Es$
 ⟨*proof*⟩

lemma *ith-typing-sub*:
 $\bigwedge Cs. \llbracket CT; \Gamma \vdash\vdash (es @ (h \# t)) : Cs;$
 $CT; \Gamma \vdash h' : Ci';$
 $CT \vdash Ci' <: (Cs!(length\ es)) \rrbracket$
 $\implies \exists Cs'. (CT; \Gamma \vdash\vdash (es @ (h' \# t)) : Cs' \wedge CT \vdash\vdash Cs' <: Cs)$
 ⟨*proof*⟩

lemma *mem-typings*:
 $\bigwedge Cs. \llbracket CT; \Gamma \vdash\vdash es : Cs; ei \in set\ es \rrbracket \implies \exists Ci. CT; \Gamma \vdash ei : Ci$
 ⟨*proof*⟩

lemma *typings-proj*:
 assumes $CT; \Gamma \vdash\vdash ds : As$
 and $CT \vdash\vdash As <: Bs$
 and $length\ ds = length\ As$
 and $length\ ds = length\ Bs$
 and $i < length\ ds$
 shows $CT; \Gamma \vdash ds!i : As!i$ and $CT \vdash As!i <: Bs!i$
 ⟨*proof*⟩

lemma *subtypings-length*:
 $CT \vdash\vdash As <: Bs \implies length\ As = length\ Bs$
 ⟨*proof*⟩

lemma *not-subtypes-aux*:
 assumes $CT \vdash C <: Da$
 and $C \neq Da$
 and $CT\ C = Some\ CDef$
 and $cSuper\ CDef = D$
 shows $CT \vdash D <: Da$
 ⟨*proof*⟩

lemma *not-subtypes*:
 assumes $CT \vdash A <: C$
 shows $\bigwedge D. \llbracket CT \vdash D \neg <: C; CT \vdash C \neg <: D \rrbracket \implies CT \vdash A \neg <: D$
 ⟨*proof*⟩

2.2.5 Sub-Expressions

lemma *isubexpr-typing*:
 assumes $e1 \in isubexprs(e0)$
 shows $\bigwedge C. \llbracket CT; Map.empty \vdash e0 : C \rrbracket \implies \exists D. CT; Map.empty \vdash e1 : D$
 ⟨*proof*⟩

lemma *subexpr-typing*:

assumes $e1 \in \text{subexprs}(e0)$
shows $\bigwedge C. \llbracket CT; \text{Map.empty} \vdash e0 : C \rrbracket \implies \exists D. CT; \text{Map.empty} \vdash e1 : D$
 $\langle \text{proof} \rangle$

lemma *isubexpr-reduct*:
assumes $d1 \in \text{isubexprs}(e1)$
shows $\bigwedge d2. \llbracket CT \vdash d1 \rightarrow d2 \rrbracket \implies \exists e2. CT \vdash e1 \rightarrow e2$
 $\langle \text{proof} \rangle$

lemma *subexpr-reduct*:
assumes $d1 \in \text{subexprs}(e1)$
shows $\bigwedge d2. \llbracket CT \vdash d1 \rightarrow d2 \rrbracket \implies \exists e2. CT \vdash e1 \rightarrow e2$
 $\langle \text{proof} \rangle$

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 $\exists xs \ e. \text{mbody}(CT, m, C) = xs . e \wedge \text{length } xs = \text{length } Cs$
 $\langle \text{proof} \rangle$

lemma *mtype-mbody-length*:
assumes $\text{mt}:\text{mtype}(CT, m, C) = Cs \rightarrow C0$
and $\text{mb}:\text{mbody}(CT, m, C) = xs . e$
shows $\text{length } xs = \text{length } Cs$
 $\langle \text{proof} \rangle$

3.2 Method Types and Field Declarations of Subtypes

lemma *A-1-1*:
assumes $CT \vdash C <: D$ **and** $CT \text{ OK}$
shows $(\text{mtype}(CT, m, D) = Cs \rightarrow C0) \implies (\text{mtype}(CT, m, C) = Cs \rightarrow C0)$
 $\langle \text{proof} \rangle$

lemma *sub-fields*:
assumes $CT \vdash C <: D$

shows $\bigwedge Dg. \text{fields}(CT, D) = Dg \implies \exists Cf. \text{fields}(CT, C) = (Dg @ Cf)$
 ⟨*proof*⟩

3.3 Substitution Lemma

lemma *A-1-2*:

assumes *CT OK*

and $\Gamma = \Gamma 1 ++ \Gamma 2$

and $\Gamma 2 = [xs \mapsto] Bs$

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

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

and $\exists As. CT; \Gamma 1 \vdash ds : As \wedge CT \vdash As <: Bs$

shows $CT; \Gamma \vdash es : Ds \implies \exists Cs. (CT; \Gamma 1 \vdash ([ds/xs]es) : Cs \wedge CT \vdash Cs <: Ds)$ (**is** *?TYPINGS* \implies *?P1*)

and $CT; \Gamma \vdash e : D \implies \exists C. (CT; \Gamma 1 \vdash ((ds/xs)e) : C \wedge CT \vdash C <: D)$ (**is** *?TYPING* \implies *?P2*)

⟨*proof*⟩

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 $(CT; \Gamma 2 \vdash es : Cs) \implies (CT; \Gamma 1 ++ \Gamma 2 \vdash es : Cs)$ (**is** *?P1* \implies *?P2*)

and $CT; \Gamma 2 \vdash e : C \implies CT; \Gamma 1 ++ \Gamma 2 \vdash e : C$ (**is** *?Q1* \implies *?Q2*)

⟨*proof*⟩

3.5 Method Body Typing Lemma

lemma *A-1-4*:

assumes *ct-ok: CT OK*

and $mb : mbody(CT, m, C) = xs . e$

and $mt : mtype(CT, m, C) = Ds \rightarrow D$

shows $\exists D0 \ C0. (CT \vdash C <: D0) \wedge$

$(CT \vdash C0 <: D) \wedge$

$(CT; [xs \mapsto] Ds)(this \mapsto D0) \vdash e : C0)$

⟨*proof*⟩

3.6 Subject Reduction Theorem

theorem *Thm-2-4-1*:

assumes $CT \vdash e \rightarrow e'$

and *CT OK*

shows $\bigwedge C. \llbracket CT; \Gamma \vdash e : C \rrbracket$

$\implies \exists C'. (CT; \Gamma \vdash e' : C' \wedge CT \vdash C' <: C)$

⟨*proof*⟩

3.7 Multi-Step Subject Reduction Theorem

corollary *Cor-2-4-1-multi:*

assumes $CT \vdash e \rightarrow^* e'$

and $CT \text{ OK}$

shows $\bigwedge C. \llbracket CT; \Gamma \vdash e : C \rrbracket \implies \exists C'. (CT; \Gamma \vdash e' : C' \wedge CT \vdash C' <: C)$

\langle proof \rangle

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

shows $\exists Cf \ fDef. \text{fields}(CT, C0) = Cf \wedge \text{lookup } Cf \ (\lambda fd. (vdName \ fd = fi)) = \text{Some } fDef$

\langle proof \rangle

lemma *Thm-2-4-2-2:*

fixes $es \ ds :: \text{exp list}$

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

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

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

\langle proof \rangle

lemma *closed-subterm-split:*

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

shows

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

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

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

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

\langle proof \rangle

3.9 Type Soundness Theorem

theorem *Thm-2-4-3:*

assumes $e\text{-typ}: CT; \text{Map.empty} \vdash e : C$

and $ct\text{-ok}: CT \text{ OK}$

and $\text{multisteps}: CT \vdash e \rightarrow^* e1$

and $\text{no-step}: \neg(\exists e2. CT \vdash e1 \rightarrow e2)$

shows $(\text{val}(e1) \wedge (\exists D. CT; \text{Map.empty} \vdash e1 : D \wedge CT \vdash D <: C))$

$\vee (\exists D \ C \ es. (\text{Cast } D \ (New \ C \ es)) \in \text{subexprs}(e1) \wedge CT \vdash C \neg <: D)$

\langle proof \rangle

end

```

theory Execute
imports FJSound
begin

```

4 Executing FeatherweightJava programs

We execute FeatherweightJava programs using the predicate compiler.

```

code-pred (modes:  $i \Rightarrow i \Rightarrow i \Rightarrow \text{bool}$ ,
 $i \Rightarrow i \Rightarrow o \Rightarrow \text{bool}$  as supertypes-of) subtyping  $\langle \text{proof} \rangle$ 

```

```

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 []  $xs\ xs$ 
 $\langle \text{proof} \rangle$ 

```

```

lemma [code-pred-intro]: append  $xs\ ys\ zs \Longrightarrow \text{append } (x\#xs)\ ys\ (x\#zs)$ 
 $\langle \text{proof} \rangle$ 

```

With this at hand, we derive new introduction rules for the reduction relation:

```

lemma rc-invok-arg':  $CT \vdash ei \rightarrow ei' \Longrightarrow \text{append } el\ (ei\ \#\ er)\ e' \Longrightarrow \text{append } el\ (ei' \#\ er)\ e'' \Longrightarrow$ 
 $CT \vdash \text{MethodInvk } e\ m\ e' \rightarrow \text{MethodInvk } e\ m\ e''$ 
 $\langle \text{proof} \rangle$ 

```

```

lemma rc-new-arg':  $CT \vdash ei \rightarrow ei' \Longrightarrow \text{append } el\ (ei\ \#\ er)\ e \Longrightarrow \text{append } el\ (ei' \#\ er)\ e'$ 
 $\Longrightarrow CT \vdash \text{New } C\ e \rightarrow \text{New } C\ e'$ 
 $\langle \text{proof} \rangle$ 

```

```

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

```

```

code-pred (modes:  $i \Rightarrow i \Rightarrow i \Rightarrow \text{bool}$ ,  $i \Rightarrow i \Rightarrow o \Rightarrow \text{bool}$  as reduce)
reduction
 $\langle \text{proof} \rangle$ 

```

```

thm reduction.equation

```

```

code-pred reductions  $\langle \text{proof} \rangle$ 

```

thm *reductions.equation*

We also make the class typing executable: this requires that we derive rules for *method-typing*.

definition *method-typing-aux*

where

method-typing-aux $CT\ m\ D\ Cs\ C = (\neg (\forall Ds\ D0. mtype(CT,m,D) = Ds \rightarrow D0 \rightarrow Cs = Ds \wedge C = D0))$

lemma *method-typing-aux*:

$(\forall Ds\ D0. mtype(CT,m,D) = Ds \rightarrow D0 \rightarrow Cs = Ds \wedge C = D0) = (\neg \text{method-typing-aux } CT\ m\ D\ Cs\ C)$
<proof>

lemma [*code-pred-intro*]:

$mtype(CT,m,D) = Ds \rightarrow D0 \implies Cs \neq Ds \implies \text{method-typing-aux } CT\ m\ D\ Cs\ C$
<proof>

lemma [*code-pred-intro*]:

$mtype(CT,m,D) = Ds \rightarrow D0 \implies C \neq D0 \implies \text{method-typing-aux } CT\ m\ D\ Cs\ C$
<proof>

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>

4.1 A simple example

We now execute a simple FJ example program:

abbreviation $A :: \text{className}$

where $A == \text{Suc } 0$

abbreviation $B :: \text{className}$

where $B == 2$

abbreviation $cPair :: \text{className}$

where $cPair == 3$

definition *classA-Def* :: *classDef*

where

classA-Def = ($\{ cName = A, cSuper = \text{Object}, cFields = [], cConstructor = (kName = A, kParams = [], kSuper = [], kInits = []), cMethods = [] \}$)

definition

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

abbreviation $ffst :: varName$

where

$ffst == 4$

abbreviation $fsnd :: varName$

where

$fsnd == 5$

abbreviation $setfst :: methodName$

where

$setfst == 6$

abbreviation $newfst :: varName$

where

$newfst == 7$

definition $classPair-Def :: classDef$

where

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

definition $exampleProg :: classTable$

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

value $exampleProg \vdash classA-Def OK$

value $exampleProg \vdash classB-Def OK$

value $exampleProg \vdash classPair-Def OK$

values $\{x. exampleProg \vdash MethodInvk (New cPair [New A [], New B []]) setfst [New B []] \rightarrow^* x\}$

values $\{x. exampleProg \vdash FieldProj (FieldProj (New cPair [New cPair [New A [], New B []], New A []]) ffst) fsnd \rightarrow^* x\}$

end

```
theory Featherweight-Java
imports FJSound Execute
begin

end
```

References

- [1] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.
- [2] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283. 2002. <http://www.in.tum.de/~nipkow/LNCS2283/>.