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
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 nat. We use the finite maps defined in Map.thy to represent typing contexts and the static class table. This section defines the representations of each syntactic category (expressions, methods, constructors, classes, class tables) and defines several constants (Object and this).

1.1.1 Type definitions

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

type-synonym varCtx = varName -> className

1.1.2 Constants

definition
  Object :: className where
  Object = 0

definition
  this :: varName where
  this == 0

1.1.3 Expressions

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

1.1.4 Methods

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

1.1.5 Constructors

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

1.1.6 Classes

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

\[ \text{type-synonym classTable = className → classDef} \]

1.2 Sub-expression Relation

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

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

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

1.3 Values

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

\[ \begin{align*}
\text{inductive} & \quad \text{vals} :: [\text{exp list}] \Rightarrow \text{bool} \quad (\cdot \in \text{vals}'(\cdot) [80] 80) \\
& \quad \text{and} \quad \text{val} :: [\text{exp}] \Rightarrow \text{bool} \quad (\cdot \in \text{val}'(\cdot) [80] 80)
\end{align*} \]

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

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.

\[ \begin{align*}
\text{primrec} & \quad \text{substs} :: (\text{varName} → \text{exp}) → \text{exp} → \text{exp} \\
& \quad \text{and} \quad \text{subst-list1} :: (\text{varName} → \text{exp}) → \text{exp list} → \text{exp list} \\
& \quad \text{and} \quad \text{subst-list2} :: (\text{varName} → \text{exp}) → \text{exp list} → \text{exp list} \text{ where}
\end{align*} \]

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

(\text{where})

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

abbreviation

subst-list-syn :: [exp list] ⇒ [varName list] ⇒ [exp list] ⇒ exp list

(\text{where})

d/s/xes \equiv \text{map (substs (map-upds empty xs ds)) es}

1.5 Lookup

The function lookup 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

(\text{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

(\text{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**

\[
\text{inductive subtyping :: [classTable, className, className] } \Rightarrow \text{bool} \quad (-\vdash -<: -[80,80,80] 80)
\]

**Where**

\[
s-refl : \quad CT \vdash C <: C
\]

\[
s-trans : \quad [\quad CT \vdash C <: D; CT \vdash D <: E \quad ] \implies CT \vdash C <: E
\]

\[
s-super : \quad [\quad CT(C) = \text{Some}(CDef); cSuper CDef = D \quad ] \implies CT \vdash C <: D
\]

**Abstraction**

\[
\text{abbreviation neg-subtyping :: [classTable, className, className] } \Rightarrow \text{bool} \quad (-\vdash -\neg<: -[80,80,80] 80)
\]

**Where**

\[
\text{where CT} \vdash S \neg<: T \equiv \neg CT \vdash S <: T
\]

**Inductive**

\[
\text{inductive subtypings :: [classTable, className list, className list] } \Rightarrow \text{bool} \quad (-\vdash +- <: -[80,80,80] 80)
\]

**Where**

\[
s-nil : \quad CT \vdash [] <: []
\]

\[
s-cons : \quad [\quad CT \vdash C0 <: D0; CT \vdash +Cs <: Ds \quad ] \implies CT \vdash +(C0 \# Cs) <: (D0 \# Ds)
\]

### 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**

\[
\text{inductive fields :: [classTable, className, varDef list] } \Rightarrow \text{bool} \quad (fields'(\cdot,\cdot') = -[80,80,80] 80)
\]

**Where**

\[
\text{where f-obj:}
\]

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

\[
\mid f-class:
\]

\[
[\quad CT(C) = \text{Some}(CDef); cSuper CDef = D; cFields CDef = Cf; fields(CT,D) = Dg; DgCf = Dg \circ Cf \quad ]
\]

\[
\implies \text{fields}(CT,C) = DgCf
\]

### 1.9 mtype Relation

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

**Inductive**


mtype :: [classTable, methodName, className, className list, className] ⇒ bool
(mtype'(\_,\_,\_') = - \rightarrow [80,80,80,80] 80)
where
  mt-class:
  \[
  CT(C) = Some(CDef);
  \text{lookup} (cMethods CDef) \left(\lambda \text{md.(mName md = m)}\right) = Some(mDef);
  \text{varDefs-types} (mParams mDef) = Bs;
  \text{mReturn mDef} = B \]
  \implies mtype(CT,m,C) = Bs \rightarrow B

| mt-super:
  \[
  CT(C) = Some(CDef);
  \text{lookup} (cMethods CDef) \left(\lambda \text{md.(mName md = m)}\right) = None;
  cSuper CDef = D;
  \text{mtype}(CT,m,D) = Bs \rightarrow B \]
  \implies mtype(CT,m,C) = Bs \rightarrow B

1.10 mbody Relation

The mtype relation, written \(\text{mbody}(CT,m,C) = xs.e_0\) 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.

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

| mb-super:
  \[
  CT(C) = Some(CDef);
  \text{lookup} (cMethods CDef) \left(\lambda \text{md.(mName md = m)}\right) = None;
  cSuper CDef = D;
  \text{mbody}(CT,m,D) = xs . e \]
  \implies 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 \(\Gamma\). The multi-typing relation, written \(CT;\Gamma \vdash +es : Cs\) relates lists of expressions to lists of types.

\(\text{inductive}\)
typings :: [classTable, varCtx, exp list, className list] ⇒ bool (\$\vdash \vdash \cdot : \cdot [80,80,80,80] 80)
and typing :: [classTable, varCtx, exp, className] ⇒ bool (\$\vdash \vdash \cdot : \cdot [80,80,80,80] 80)

where
ts-nil : CT;Γ \vdash [] : []

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

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

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

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

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

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

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

| t-scast :
  [ CT;Γ \vdash e0 : D;]
We occasionally find the following induction principle, which only mentions the typing of a single expression, more useful than the mutual induction principle generated by Isabelle, which mentions the typings of single expressions and of lists of expressions.

**Lemma**: **typing-induct**:

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

**And** $\forall C \backslash C \vdash \Gamma \vdash \overline{x} : Some C \implies P C \Gamma (\overline{x} e) C$

$I C \vdash e0 : C0; P C \Gamma e0 C0; fields(CT, C0) = C0; lookup C0 (Cfd, vdName fd = fi) = Some fDef; vdType fDef = Ci;\implies P C \Gamma (FieldProj e0 fi)$

$\forall C \vdash C0 : C; Ds \vdash e0 m e m. [\Gamma (\overline{x} e0 : C0; P C \Gamma e0 C0; mtype(CT, mC0)) = Ds \implies C; CT: \Gamma \vdash e : Cs; \forall i . [i < length es] \implies P C \Gamma (es!i) (Csi)]; CT \vdash Cs \vdash Cs \vdash Ds; length es = length Ds \implies P C \Gamma (MethodInvk e0 m es) C$

$\forall C \vdash Cs : Ds \vdash e0 e0 Cs. fields(CT, C) = Ds; length es = length Ds$;

$\forall varDefs-types Df = Ds; CT: \Gamma \vdash e : Cs; \forall i . [i < length es] \implies P C \Gamma (es!i) (Csi)i; CT \vdash Cs \vdash Cs \vdash Ds \implies P C \Gamma (New C es) C$

$\forall C \vdash C0 : C; D \vdash e0 : D; P C \Gamma e0 D; CT \vdash D <; C \implies P C \Gamma (\overline{x} e0 C0) C$

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

**Proof**

### 1.12 Method Typing Relation

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

**Inductive**

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

**Where**

**m-typing**:

$[CT(C) = Some(CDef) ;$

cName CDef = C ;

cSuper CDef = D ;

mName mDef = m ;

lookup (cMethods CDef) (Amd.(mName md = m)) = Some(mDef);$

mReturn mDef = C0 ;
mParams mDef = Cxs ; mBody mDef = e0 ;

varDefs-types Cxs = Cxs ;

varDefs-names Cxs = Cxs ;

$\Gamma = (\text{map-upds empty xs Cs})(this \mapsto C)$


\[ \Gamma \vdash e_0 : E_0; \\
\vdash E_0 <: C_0; \\
\forall D_0.\ (\text{mtype}(CT, m, D) = D_0 \rightarrow (C_0 = D_0 \land C_0 = D_0)) \]
\]

\[ \Rightarrow \Gamma \vdash \text{mDef OK IN C} \]

**inductive**

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

**where**

\[ ms\text{-nil} : \\
\vdash + [] \text{ OK IN C} \]

\[ \mid ms\text{-cons} : \\
\vdash m \text{ OK IN C}; \\
\vdash ms \text{ OK IN C} \]
\]
\[ \Rightarrow \Gamma \vdash + (m \# ms) \text{ OK IN C} \]

**1.13 Class Typing Relation**

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

\[ \text{inductive} \]

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

**where**

\[ t\text{-class} : \mid \text{cName CDef} = C; \\
\text{cSuper CDef} = D; \\
\text{cMethods CDef} = M; \\
\text{kName KDef} = C; \\
\text{kParams KDef} = (D_0 @ C_0); \\
\text{kSuper KDef} = \text{varDefs-names D_0}; \\
\text{kInits KDef} = \text{varDefs-names C_0}; \\
\text{fields}(CT, D) = D_0; \\
\vdash M \text{ OK IN C} \]
\]
\[ \Rightarrow \Gamma \vdash C \text{Def OK} \]

**1.14 Class Table Typing Relation**

A class table is well-typed, written \( \Gamma \vdash \text{OK} \) if for every class name \( C \), the class definition mapped to by \( \Gamma \) is is well-typed and has name \( C \).

\[ \text{inductive} \]

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

**where**

\[ ct\text{-all-ok} : \mid \text{Object } \notin \text{dom}(CT); \\
\forall C \text{ CDef}. \ CT(C) = \text{Some}(C\text{Def}) \rightarrow (\Gamma \vdash \text{CDef OK}) \land (\text{cName CDef} = C) \]
\]
\[ \Rightarrow \Gamma \text{ OK} \]
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**

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

where

- **r-field**:

  \[ \text{fields}(CT,C) = Cf; \]
  \[ \text{lookup2 Cf es} (\lambda fd.(vdName fd = fi)) = Some(ei) \]
  \[ \Rightarrow CT \vdash \text{FieldProj} (\text{New C es}) \ f_i \rightarrow ei \]

- **r-invk**:

  \[ \text{mbody}(CT,m,C) = xs . e0; \]
  \[ \text{substs} ((\text{map-upds empty xs ds})(\text{this} \mapsto (\text{New C es}))) e0 = e0' \]
  \[ \Rightarrow CT \vdash \text{MethodInvk} (\text{New C es}) \ m \ ds \rightarrow e0' \]

- **r-cast**:

  \[ \Rightarrow CT \vdash \text{Cast D} (\text{New C es}) \rightarrow \text{New C es} \]

- **re-field**:

  \[ \Rightarrow CT \vdash \text{FieldProj} e0 f \rightarrow \text{FieldProj} e0' f \]

- **re-invk-recv**:

  \[ \Rightarrow CT \vdash \text{MethodInvk} e0 m \ es \rightarrow \text{MethodInvk} e0' m \ es \]

- **re-invk-arg**:

  \[ \Rightarrow CT \vdash \text{MethodInvk} e0 m (el@[ei]#er) \rightarrow \text{MethodInvk} e0 m (el@[ei']#er) \]

- **re-new-arg**:

  \[ \Rightarrow CT \vdash \text{New C} (el@[ei]#er) \rightarrow \text{New C} (el@[ei']#er) \]

- **re-cast**:

  \[ \Rightarrow CT \vdash \text{Cast C} e0 \rightarrow \text{Cast C} e0' \]

**inductive**

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

where

- **rs-refl**:

  $CT \vdash e \rightarrow^* e$

- **rs-trans**:

  \[ CT \vdash e \rightarrow e'; CT \vdash e' \rightarrow^* e'' \Rightarrow CT \vdash e \rightarrow^* e'' \]

end
2 FJ Aux: Auxiliary Lemmas

theory FJ Aux imports FJDefs begin

2.1 Non-FJ Lemmas

2.1.1 Lists

lemma mem-ith:
  assumes \( e_i \in \text{set es} \)
  shows \( \exists e_l e_r. \, \text{es} = e_l @ e_i # e_r \)

lemma ith-mem:
  \( \forall i. \, \,[ \, i < \text{length es} \, ] \implies \, \text{es}! i \in \text{set es} \)

2.1.2 Maps

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

lemma map-upds-index:
  assumes \( \text{length xs} = \text{length As} \)
  and \( \text{[xs].\mapsto As[x] = Some Ai} \)
  shows \( \exists i. \, (\text{As}! i = Ai) \land (i < \text{length As}) \land (\forall (Bs::'c list). ((\text{length Bs} = \text{length As}) \implies (\text{[xs].\mapsto} Bxs x = Some (Bs ! i)))) \)

2.2 FJ Lemmas

2.2.1 Substitution

lemma subst-list1-eq-map-substs :
  \( \forall \sigma. \, \text{subst-list1} \sigma \text{l} = \text{map} (\text{substs} \sigma) \text{l} \)

lemma subst-list2-eq-map-substs :
  \( \forall \sigma. \, \text{subst-list2} \sigma \text{l} = \text{map} (\text{substs} \sigma) \text{l} \)

2.2.2 Lookup

lemma lookup-functional:
  assumes \( \text{lookup l f} = a1 \)
and $\text{lookup } l f = o2$

shows $o1 = o2$

(proof)

\textbf{lemma lookup-true:}

$\text{lookup } l f = \text{Some } r \implies f r$

(proof)

\textbf{lemma lookup-hd:}

$[\text{length } l > 0; f (l!0)] \implies \text{lookup } l f = \text{Some } (l!0)$

(proof)

\textbf{lemma lookup-split:}$\text{lookup } l f = \text{None } \lor (\exists h. \text{lookup } l f = \text{Some } h)$

(proof)

\textbf{lemma lookup-index:}

\begin{itemize}
  \item \textbf{assumes}$\text{lookup } l1 f = \text{Some } e$
  \item \textbf{shows}$\forall l2. \exists i < (\text{length } l1). e = l1!i \land \left(\left(\text{length } l1 = \text{length } l2\right) \implies \text{lookup2 } l1 l2 f = \text{Some } (l2!i)\right)$
\end{itemize}

(proof)

\textbf{lemma lookup2-index:}

$\forall l2. \left(\text{lookup2 } l1 l2 f = \text{Some } e; \text{length } l1 = \text{length } l2\right) \implies \exists i < (\text{length } l2). e = (l2!i) \land \text{lookup } l1 f = \text{Some } (l1!i)$

(proof)

\textbf{lemma lookup-append:}

\begin{itemize}
  \item \textbf{assumes}$\text{lookup } l f = \text{Some } r$
  \item \textbf{shows}$\text{lookup } (l@l') f = \text{Some } r$
\end{itemize}

(proof)

\textbf{lemma method-typings-lookup:}

\begin{itemize}
  \item \textbf{assumes}$\text{lookup-eq-Some: } \text{lookup } M f = \text{Some } m\text{Def}$
  \item \text{and } $M\text{-ok: } CT \vdash M \text{ OK } IN C$
  \item \textbf{shows}$CT \vdash m\text{Def } \text{OK } IN C$
\end{itemize}

(proof)

\subsection{Functional}

These lemmas prove that several relations are actually functions

\textbf{lemma mtype-functional:}

\begin{itemize}
  \item \textbf{assumes}$\text{mtype}(CT,m,C) = Cs \rightarrow C0$
  \item \text{and } $\text{mtype}(CT,m,C) = Ds \rightarrow D0$
  \item \textbf{shows}$Ds=Cs \land D0=C0$
\end{itemize}

(proof)

\textbf{lemma mbbody-functional:}

\begin{itemize}
  \item \textbf{assumes}$\text{mbbody}(CT,m,C) = xs . e0$
\end{itemize}
and \( mb2 : \text{mbody}(CT, m, C) = ys . d0 \)

shows \( xs = ys \land e0 = d0 \)

\( \langle \text{proof} \rangle \)

**Lemma fields-functional:**
- **Assumes:** \( \text{fields}(CT, C) = Cf \)
- **And:** \( CT \text{ OK} \)
- **Shows:** \( \bigwedge Cf'. [ \text{fields}(CT, C) = Cf ] \implies Cf = Cf' \)

\( \langle \text{proof} \rangle \)

### 2.2.4 Subtyping and Typing

**Lemma typings-lengths:** **Assumes** \( CT; \Gamma \vdash es: Cs \) **Shows** \( \text{length } es = \text{length } Cs \)

\( \langle \text{proof} \rangle \)

**Lemma typings-index:**
- **Assumes:** \( CT; \Gamma \vdash es: Cs \)
- **Shows:** \( \bigwedge i. [ i < \text{length } Cs ] \implies CT; \Gamma \vdash (es!i) : (Cs!i) \)

\( \langle \text{proof} \rangle \)

**Lemma subtypings-index:**
- **Assumes:** \( CT \vdash+ Cs <: Ds \)
- **Shows:** \( \bigwedge i. [ i < \text{length } Cs ] \implies CT \vdash (Cs!i) <: (Ds!i) \)

\( \langle \text{proof} \rangle \)

**Lemma subtyping-append:**
- **Assumes:** \( CT \vdash+ Cs <: Ds \)
- **And:** \( CT \vdash C <: D \)
- **Shows:** \( CT \vdash+ (Cs@[C]) <: (Ds@[D]) \)

\( \langle \text{proof} \rangle \)

**Lemma typings-append:**
- **Assumes:** \( CT; \Gamma \vdash+ es : Cs \)
- **And:** \( CT; \Gamma \vdash e : C \)
- **Shows:** \( CT; \Gamma \vdash+ (es@[e]) : (Cs@[C]) \)

\( \langle \text{proof} \rangle \)

**Lemma ith-typing:** \( \bigwedge Cs. [ CT; \Gamma \vdash+ (es@[h#t]) : Cs ] \implies CT; \Gamma \vdash h : (Cs![(\text{length } es)]) \)

\( \langle \text{proof} \rangle \)

**Lemma ith-subtyping:** \( \bigwedge Ds. [ CT \vdash+ (Cs@[h#t]) <: Ds ] \implies CT \vdash h <: (Ds![(\text{length } Cs)]) \)

\( \langle \text{proof} \rangle \)

**Lemma subtypings-refl:** \( CT \vdash+ Cs <: Cs \)

\( \langle \text{proof} \rangle \)
lemma subtypings-trans: \( \forall Ds \ Es. [ CT \vdash+ Cs <: Ds; \ CT \vdash+ Ds <: Es ] \Rightarrow \) 
\( CT \vdash+ Cs <: Es \) 
〈proof〉

lemma ith-typing-sub: 
\( \forall Cs. [ CT; \Gamma \vdash+ (es@h@t) : Cs; \) 
\( CT; \Gamma \vdash h' : Ci'; \) 
\( CT \vdash Ci' <: (Cs!(length es)) ] \Rightarrow \exists Cs'. (CT; \Gamma \vdash+ (es@h'@t) : Cs' \land CT \vdash+ Cs' <: Cs) \) 
〈proof〉

lemma mem-typings: 
\( \forall Cs. [ CT; \Gamma \vdash+ es:Cs; ei \in set es ] \Rightarrow \exists Ci. CT; \Gamma \vdash ei:Ci \) 
〈proof〉

lemma typings-proj: 
assumes \( CT; \Gamma \vdash+ ds : As \) 
and \( CT \vdash+ As <: Bs \) 
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: 
\( CT \vdash+ As <: Bs \Rightarrow length As = length Bs \) 
〈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 \( \forall D. [ CT \vdash D \neg<: C; \ CT \vdash C \neg<: D ] \Rightarrow CT \vdash A \neg<: D \) 
〈proof〉

2.2.5 Sub-Expressions

lemma isubexpr-typing: 
assumes \( e1 \in isubexprs(e0) \) 
shows \( \forall C. [ CT; \emptyset \vdash e0 : C ] \Rightarrow \exists D. CT; \emptyset \vdash e1 : D \) 
〈proof〉

lemma subexpr-typing:

15
assumes $e_1 \in \text{subexprs}(e_0)$
shows $\forall C. [CT;\text{empty} \vdash e_0 : C] \implies \exists D. CT;\text{empty} \vdash e_1 : D$

\text{lemma isubexpr-reduct:}
assumes $d_1 \in \text{isubexprs}(e_1)$
shows $\forall d_2. [CT \vdash d_1 \rightarrow d_2] \implies \exists e_2. CT \vdash e_1 \rightarrow e_2$

\text{lemma subexpr-reduct:}
assumes $d_1 \in \text{subexprs}(e_1)$
shows $\forall d_2. [CT \vdash d_1 \rightarrow d_2] \implies \exists e_2. CT \vdash e_1 \rightarrow e_2$

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

\text{lemma mtype-mbody:}
fixes $Cs :: \text{nat list}$
assumes $mtype(CT,m,C) = Cs \rightarrow C0$
shows $\exists e. mbody(CT,m,C) = xs . e \land \text{length } xs = \text{length } Cs$

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

3.2 Method Types and Field Declarations of Subtypes

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

\text{lemma sub-fields:}
assumes $CT \vdash C <: D$
\[ \forall Dg. \text{fields}(CT, D) = Dg \implies \exists Cf. \text{fields}(CT, C) = (Dg \oplus Cf) \]

(\text{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 \( \exists As. CT;\Gamma_1 \vdash ds : As \land CT \vdash As < Bs \)
- shows \( CT;\Gamma \vdash es : Ds \implies \exists Cs. (CT;\Gamma_1 \vdash [(ds/xs)es] : Cs \land CT \vdash Cs < Ds) \) (is ?TYPIINGS \( \implies ?P1 \))
- and \( CT;\Gamma \vdash e : D \implies \exists C. (CT;\Gamma_1 \vdash ((ds/xs)e) : C \land CT \vdash C < D) \) (is ?TYPING \( \implies ?P2 \))

(\text{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 \( \implies ?Q2 \))

(\text{proof})

### 3.5 Method Body Typing Lemma

**Lemma A-1-4:**

- assumes \( ct-ok: CT \) OK
- and \( mb=\text{mbody}(CT,m,C) = xs \cdot e \)
- and \( mt=\text{mtype}(CT,m,C) = Ds \rightarrow D \)
- shows \( \exists D0 C0. (CT \vdash C < D0) \land (CT \vdash C0 < D) \land (CT;[xs\mapsto Ds] \text{this} \rightarrow D0) \vdash e : C0) \)

(\text{proof})

### 3.6 Subject Reduction Theorem

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

- assumes \( CT \vdash e \rightarrow e' \)
- and \( CT \) OK
- shows \( \forall C. [ CT;\Gamma \vdash e : C ] \implies \exists C'. (CT;\Gamma \vdash e' : C' \land CT \vdash C' < C) \)

(\text{proof})
3.7 Multi-Step Subject Reduction Theorem

corollary Cor-2-4-1-multi:
assumes $CT \vdash e \rightarrow^* e'$
and $CT$ OK
shows $\forall C. \left[ CT; \Gamma \vdash e : C \right] \implies \exists C'. (CT; \Gamma \vdash e' : C' \land CT \vdash C' <: C)$
(proof)

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 $CT;\emptyset \vdash e : C$
and $\text{FieldProj}(\text{New } C_0 es) \; \text{fi} \in \text{subexprs}(e)$
shows $\exists Cf \; \text{fDef}. \; \text{fields}(CT, C_0) = Cf \land \text{lookup} Cf (\lambda fd. (\text{vdName fd} = \text{fi})) = \text{Some fDef}$
(proof)

lemma Thm-2-4-2-2:
fixes $es \; ds :: \text{exp list}$
assumes $CT;\emptyset \vdash e : C$
and $\text{MethodInvk}(\text{New } C_0 es) \; m \; ds \in \text{subexprs}(e)$
shows $\exists xs \; e_0. \; \text{mbody}(CT, m, C_0) = xs \cdot e_0 \land \text{length} xs = \text{length} ds$
(proof)

lemma closed-subterm-split:
assumes $CT;\Gamma \vdash e : C$ and $\Gamma = \emptyset$
shows $((\exists C_0 es \; \text{fi}. \; (\text{FieldProj}(\text{New } C_0 es) \; \text{fi}) \in \text{subexprs}(e))$
$\lor (\exists C_0 es \; m \; ds. \; (\text{MethodInvk}(\text{New } C_0 es) \; m \; ds) \in \text{subexprs}(e))$
$\lor (\exists C_0 D es. \; (\text{Cast } D(\text{New } C_0 es)) \in \text{subexprs}(e))$
$\lor \text{val}(e) \; (\text{is} \; ?F e \lor \text{is} \; ?M e \lor \text{is} \; ?C e \lor \text{is} \; ?V e \; \text{is} \; ?IH e)$
(proof)

3.9 Type Soundness Theorem

theorem Thm-2-4-3:
assumes e-typ: $CT;\emptyset \vdash e : C$
and ct-ok: $CT$ OK
and multisteps: $CT \vdash e \rightarrow^* e_1$
and no-step: $\neg(\exists e_2. \; CT \vdash e_1 \rightarrow e_2)$
shows $\forall (\exists D. \; CT;\emptyset \vdash e_1 : D \land CT \vdash D <: C)\land\forall (\exists D C es. \; (\text{Cast } D(\text{New } C es) \in \text{subexprs}(e_1) \land CT \vdash C \not<: D))$
(proof)
4 Executing FeatherweightJava programs

We execute FeatherweightJava programs using the predicate compiler.

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

\textbf{thm} subtyping\textunderscore equation

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

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

\textbf{lemma} \([\text{code-pred-intro}]: \) append \( [] \) \( xs \) \( xs \)
\( \langle \text{proof} \rangle \)

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

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

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

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

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

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

\textbf{thm} reduction\textunderscore equation

\textbf{code-pred reductions} \( \langle \text{proof} \rangle \)
We also make the class typing executable: this requires that we derive rules for method-typing.

**definition** method-typing-aux

where

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

**lemma** method-typing-aux:

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

(proof)

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

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

(proof)

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

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

(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 :: className

where A == Suc 0

**abbreviation** B :: className

where B == 2

**abbreviation** cPair :: className

where cPair == 3

**definition** classA-Def :: classDef

where

\[
\text{classA-Def} = \{ \text{cName} = A, \ cSuper = \text{Object}, \ cFields = [], \ cConstructor = \\
(\{\text{kName} = A, \ kParams = [], \ kSuper = [], \ kInits = [], \ cMethods = []\})
\]
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 ⊢ classA-Def OK
value exampleProg ⊢ classB-Def OK
value exampleProg ⊢ classPair-Def OK

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

end
theory Featherweight-Java
imports FJSound Execute
begin
end

References
