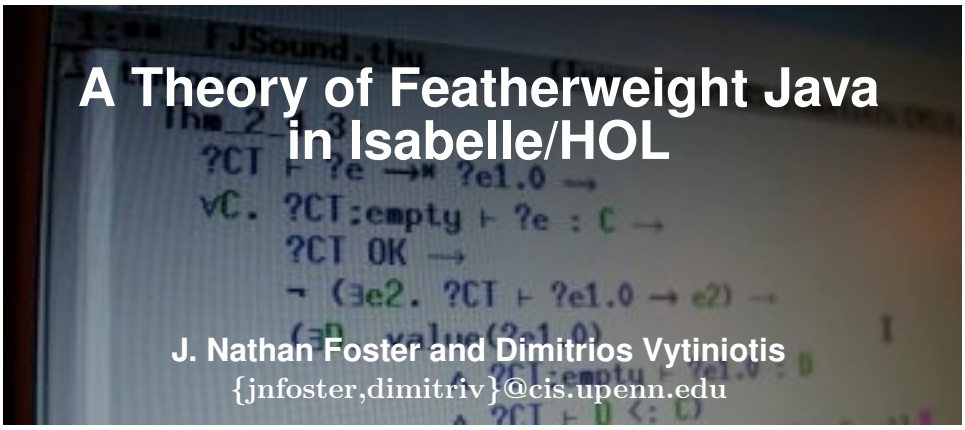

A Theory of Featherweight Java in Isabelle/HOL

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

theory *FJDefs*
imports *Main*
begin

1.1 Syntax

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

1.1.1 Type definitions

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

record *varDef* =
  vdName :: *varName*
  vdType :: *className*
type-synonym *varCtx* = *varName* ⇀ *className*

1.1.2 Constants

definition
  Object :: *className* **where**
  Object = *0*

definition
  this :: *varName* **where**
  this == *0*

1.1.3 Expressions

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

1.1.4 Methods

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

1.1.5 Constructors

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

1.1.6 Classes

record *classDef* =
  cName :: *className*
  cSuper :: *className*
  cFields :: *varDef list*
  cConstructor :: *constructorDef*
  cMethods :: *methodDef list*

1.1.7 Class Tables

type-synonym *classTable* = *className* ⇀ *classDef*

1.2 Sub-expression Relation

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

inductive-set
  isubexprs :: (*exp* ∗ *exp*) *set*
  and *isubexprs′* :: [*exp*,*exp*] ⇒ *bool* (- ∈ *isubexprs′*(-′) [*80*,*80*] *80*)
where
  e′ ∈ *isubexprs*(*e*) ≡ (*e′*,*e*) ∈ *isubexprs*
| *se-field* : *e* ∈ *isubexprs*(*FieldProj e fi*)
| *se-invkrecv* : *e* ∈ *isubexprs*(*MethodInvk e m es*)
| *se-invkarg* : ⟦ *ei* ∈ *set es* ⟧ ⟹ *ei* ∈ *isubexprs*(*MethodInvk e m es*)
| *se-newarg* : ⟦ *ei* ∈ *set es* ⟧ ⟹ *ei* ∈ *isubexprs*(*New C es*)
| *se-cast* : *e* ∈ *isubexprs*(*Cast C e*)

abbreviation
  subexprs :: [*exp*,*exp*] ⇒ *bool* (- ∈ *subexprs′*(-′) [*80*,*80*] *80*) **where**
  e′ ∈ *subexprs*(*e*) ≡ (*e′*,*e*) ∈ *isubexprs*^∗

1.3 Values

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

inductive
  vals :: [*exp list*] ⇒ *bool* (*vals′*(-′) [*80*] *80*)
  and *val* :: [*exp*] ⇒ *bool* (*val′*(-′) [*80*] *80*)
where
  vals-nil : *vals*([])
| *vals-cons* : ⟦ *val*(*vh*); *vals*(*vt*) ⟧ ⟹ *vals*((*vh* # *vt*))
| *val* : ⟦ *vals*(*vs*) ⟧ ⟹ *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* respecitvely.

primrec *substs* :: (*varName* ⇀ *exp*) ⇒ *exp* ⇒ *exp*
  and *subst-list1* :: (*varName* ⇀ *exp*) ⇒ *exp list* ⇒ *exp list*
  and *subst-list2* :: (*varName* ⇀ *exp*) ⇒ *exp list* ⇒ *exp list* **where**
  substs σ (*Var x*) = (*case* (*σ*(*x*)) *of None* ⇒ (*Var x*) | *Some p* ⇒ *p*)
| *substs σ* (*FieldProj e f*) = *FieldProj* (*substs σ e*) *f*
| *substs σ* (*MethodInvk e m es*) = *MethodInvk* (*substs σ e*) *m* (*subst-list1 σ es*)
| *substs σ* (*New C es*) = *New C* (*subst-list2 σ es*)

| *substs* σ (*Cast C e*) = *Cast C* (*substs* σ *e*)
| *subst-list1* σ [] = []
| *subst-list1* σ (*h* # *t*) = (*substs* σ *h*) # (*subst-list1* σ *t*)
| *subst-list2* σ [] = []
| *subst-list2* σ (*h* # *t*) = (*substs* σ *h*) # (*subst-list2* σ *t*)

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

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

1.5 Lookup

The fuction *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*] ⇒ *bool* (- ⊢ - <: - [*80*,*80*,*80*] *80*)
where
s-refl : $CT \vdash C <: C$
| *s-trans* : $[\![CT \vdash C <: D; CT \vdash D <: E]\!] \Longrightarrow CT \vdash C <: E$
| *s-super* : $[\![CT(C) = Some(CDef); cSuper\ CDef = D]\!] \Longrightarrow CT \vdash C <: D$

abbreviation
neg-subtyping :: [*classTable*, *className*, *className*] ⇒ *bool* (- ⊢ - ¬<: - [*80*,*80*,*80*] *80*)
where $CT \vdash S \neg<: T \equiv \neg\ CT \vdash S <: T$

inductive
subtypings :: [*classTable*, *className list*, *className list*] ⇒ *bool* (- ⊢+ - <: - [*80*,*80*,*80*] *80*)
where
ss-nil : $CT \vdash+ [] <: []$
| *ss-cons* : $[\![CT \vdash C0 <: D0; CT \vdash+ Cs <: Ds]\!] \Longrightarrow CT \vdash+ (C0 \# Cs) <: (D0 \# Ds)$

1.8 `fields` Relation

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

inductive
fields :: [*classTable*, *className*, *varDef list*] ⇒ *bool* (*fields*′(-,-′) = - [*80*,*80*,*80*] *80*)
where
f-obj:
$fields(CT,Object) = []$
| *f-class*:
$[\![CT(C) = Some(CDef); cSuper\ CDef = D; cFields\ CDef = Cf; fields(CT,D) = Dg; DgCf = Dg\ @\ Cf]\!]$
$\Longrightarrow fields(CT,C) = DgCf$

1.9 `mtype` Relation

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

inductive

mtype :: [*classTable*, *methodName*, *className*, *className list*, *className*] ⇒ *bool* (*mtype′*(-,-,-′) = - → - [*80*,*80*,*80*,*80*] *80*)
where
mt-class:
⟦ *CT*(*C*) = *Some*(*CDef*);
lookup (*cMethods CDef*) (λ*md*.(*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*) (λ*md*.(*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 $\texttt{mbody}(CT, m, C) = xs.e_0$ 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*] ⇒ *bool* (*mbody′*(-,-,-′) = - . - [*80*,*80*,*80*,*80*] *80*)
where
mb-class:
⟦ *CT*(*C*) = *Some*(*CDef*);
lookup (*cMethods CDef*) (λ*md*.(*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*) (λ*md*.(*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 $CT; \Gamma \vdash e : C$ 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*] ⇒ *bool* (-;- ⊢+ - : - [*80*,*80*,*80*,*80*] *80*)
and *typing* :: [*classTable*, *varCtx*, *exp*, *className*] ⇒ *bool* (-;- ⊢ - : - [*80*,*80*,*80*,*80*] *80*)
where
ts-nil : $CT;\Gamma \vdash+ [] : []$

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

| *t-var* :
$[\![\Gamma(x) = Some\ C]\!] \Longrightarrow CT;\Gamma \vdash (Var\ x) : C$

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

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

| *t-new* :
$[\![fields(CT,C) = Df;$
$length\ es = length\ Df;$
$varDefs\text{-}types\ Df = Ds;$
$CT;\Gamma \vdash+ es : Cs;$
$CT \vdash+ Cs <: Ds]\!]$
$\Longrightarrow CT;\Gamma \vdash New\ C\ es : C$

| *t-ucast* :
$[\![CT;\Gamma \vdash e0 : D;$
$CT \vdash D <: C]\!]$
$\Longrightarrow CT;\Gamma \vdash Cast\ C\ e0 : C$

| *t-dcast* :
$[\![CT;\Gamma \vdash e0 : D;$
$CT \vdash C <: D;\ C \neq D]\!]$
$\Longrightarrow CT;\Gamma \vdash Cast\ C\ e0 : C$

| *t-scast* :
$[\![CT;\Gamma \vdash e0 : D;$

$CT \vdash C \neg<: D$;
$CT \vdash D \neg<: C$ ⟧
$\Longrightarrow CT;\Gamma \vdash Cast\ C\ e0 : C$

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 $\bigwedge C\ CT\ \Gamma\ x.\ \Gamma\ x = Some\ C \Longrightarrow P\ CT\ \Gamma\ (Var\ x)\ C$
and $\bigwedge C0\ CT\ Cf\ Ci\ \Gamma\ e0\ fDef\ fi.$ ⟦$CT;\Gamma \vdash e0 : C0$; $P\ CT\ \Gamma\ e0\ C0$; $fields(CT,C0) = Cf$; $lookup\ Cf\ (\lambda fd.\ vdName\ fd = fi) = Some\ fDef$; $vdType\ fDef = Ci$⟧ $\Longrightarrow P\ CT\ \Gamma\ (FieldProj\ e0\ fi)\ Ci$
and $\bigwedge C\ C0\ CT\ Cs\ Ds\ \Gamma\ e0\ es\ m.$ ⟦$CT;\Gamma \vdash e0 : C0$; $P\ CT\ \Gamma\ e0\ C0$; $mtype(CT,m,C0) = Ds \to C$; $CT;\Gamma \vdash+ es : Cs$; $\bigwedge i\ .$ ⟦ $i < length\ es$ ⟧ $\Longrightarrow$ $P\ CT\ \Gamma\ (es!i)\ (Cs!i)$; $CT \vdash+ Cs <: Ds$; $length\ es = length\ Ds$⟧ $\Longrightarrow P\ CT\ \Gamma\ (MethodInvk\ e0\ m\ es)\ C$
and $\bigwedge C\ CT\ Cs\ Df\ Ds\ \Gamma\ es.$ ⟦$fields(CT,C) = Df$; $length\ es = length\ Df$; $varDefs\text{-}types\ Df = Ds$; $CT;\Gamma \vdash+ es : Cs$; $\bigwedge i.$ ⟦ $i < length\ es$ ⟧ $\Longrightarrow P\ CT\ \Gamma\ (es!i)\ (Cs!i)$; $CT \vdash+ Cs <: Ds$⟧ $\Longrightarrow P\ CT\ \Gamma\ (New\ C\ es)\ C$
and $\bigwedge C\ CT\ D\ \Gamma\ e0.$ ⟦$CT;\Gamma \vdash e0 : D$; $P\ CT\ \Gamma\ e0\ D$; $CT \vdash D <: C$⟧ $\Longrightarrow P\ CT\ \Gamma\ (Cast\ C\ e0)\ C$
and $\bigwedge C\ CT\ D\ \Gamma\ e0.$ ⟦$CT;\Gamma \vdash e0 : D$; $P\ CT\ \Gamma\ e0\ D$; $CT \vdash C <: D$; $C \neq D$⟧ $\Longrightarrow P\ CT\ \Gamma\ (Cast\ C\ e0)\ C$
and $\bigwedge C\ CT\ D\ \Gamma\ e0.$ ⟦$CT;\Gamma \vdash e0 : D$; $P\ CT\ \Gamma\ e0\ D$; $CT \vdash C \neg<: D$; $CT \vdash D \neg<: C$⟧ $\Longrightarrow P\ CT\ \Gamma\ (Cast\ C\ e0)\ C$
shows $P\ CT\ \Gamma\ e\ C$ (**is** *?P*)
⟨*proof*⟩

1.12 Method Typing Relation

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

inductive
method-typing :: [*classTable*, *methodDef*, *className*] ⇒ *bool* (- ⊢ - *OK IN* - [*80*,*80*,*80*] *80*)
where
m-typing:
⟦ $CT(C) = Some(CDef)$;
$cName\ CDef = C$;
$cSuper\ CDef = D$;
$mName\ mDef = m$;
$lookup\ (cMethods\ CDef)\ (\lambda md.(mName\ md = m)) = Some(mDef)$;
$mReturn\ mDef = C0$; $mParams\ mDef = Cxs$; $mBody\ mDef = e0$;
$varDefs\text{-}types\ Cxs = Cs$;
$varDefs\text{-}names\ Cxs = xs$;
$\Gamma = (map\text{-}upds\ Map.empty\ xs\ Cs)(this \mapsto C)$;

CT;$\Gamma \vdash e0 : E0$;
$CT \vdash E0 <: C0$;
$\forall Ds\ D0.\ (mtype(CT,m,D) = Ds \to D0) \longrightarrow (Cs{=}Ds \wedge C0{=}D0)\ ]\!]$
$\Longrightarrow CT \vdash mDef\ OK\ IN\ C$

inductive
method-typings :: [*classTable*, *methodDef list*, *className*] ⇒ *bool* (- ⊢+ - *OK IN* - [*80*,*80*,*80*] *80*)
where
ms-nil :
$CT \vdash+ []\ OK\ IN\ C$

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

1.13 Class Typing Relation

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

inductive
class-typing :: [*classTable*, *classDef*] ⇒ *bool* (- ⊢ - *OK* [*80*,*80*] *80*)
where
t-class: $[\![\ cName\ CDef = C;$
$cSuper\ CDef = D;$
$cConstructor\ CDef = KDef;$
$cMethods\ CDef = M;$
$kName\ KDef = C;$
$kParams\ KDef = (Dg@Cf);$
$kSuper\ KDef = varDefs\text{-}names\ Dg;$
$kInits\ KDef = varDefs\text{-}names\ Cf;$
$fields(CT,D) = Dg;$
$CT \vdash+ M\ OK\ IN\ C\ ]\!]$
$\Longrightarrow CT \vdash CDef\ OK$

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 is well-typed and has name C.

inductive
ct-typing :: *classTable* ⇒ *bool* (- *OK* *80*)
where
ct-all-ok:
$[\![\ Object \notin dom(CT);$
$\forall C\ CDef.\ CT(C) = Some(CDef) \longrightarrow (CT \vdash CDef\ OK) \wedge (cName\ CDef = C)\ ]\!]$
$\Longrightarrow CT\ OK$

1.15 Evaluation Relation

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

inductive
reduction :: [*classTable*, *exp*, *exp*] ⇒ *bool* (- ⊢ - → - [80,80,80] 80)
where

r-field:
⟦ *fields*(*CT*,*C*) = *Cf*;
lookup2 Cf es (λ*fd*.(*vdName fd* = *fi*)) = *Some*(*ei*) ⟧
⟹ *CT* ⊢ *FieldProj* (*New C es*) *fi* → *ei*

| *r-invk*:
⟦ *mbody*(*CT*,*m*,*C*) = *xs* . *e0*;
substs ((*map-upds Map.empty xs ds*)(*this* ↦ (*New C es*))) *e0* = *e0′* ⟧
⟹ *CT* ⊢ *MethodInvk* (*New C es*) *m ds* → *e0′*

| *r-cast*:
⟦ *CT* ⊢ *C* <: *D* ⟧
⟹ *CT* ⊢ *Cast D* (*New C es*) → *New C es*

| *rc-field*:
⟦ *CT* ⊢ *e0* → *e0′* ⟧
⟹ *CT* ⊢ *FieldProj e0 f* → *FieldProj e0′ f*

| *rc-invk-recv*:
⟦ *CT* ⊢ *e0* → *e0′* ⟧
⟹ *CT* ⊢ *MethodInvk e0 m es* → *MethodInvk e0′ m es*

| *rc-invk-arg*:
⟦ *CT* ⊢ *ei* → *ei′* ⟧
⟹ *CT* ⊢ *MethodInvk e0 m* (*el*@*ei*#*er*) → *MethodInvk e0 m* (*el*@*ei′*#*er*)

| *rc-new-arg*:
⟦ *CT* ⊢ *ei* → *ei′* ⟧
⟹ *CT* ⊢ *New C* (*el*@*ei*#*er*) → *New C* (*el*@*ei′*#*er*)

| *rc-cast*:
⟦ *CT* ⊢ *e0* → *e0′* ⟧
⟹ *CT* ⊢ *Cast C e0* → *Cast C e0′*

inductive
reductions :: [*classTable*, *exp*, *exp*] ⇒ *bool* (- ⊢ - →* - [80,80,80] 80)
where
rs-refl: *CT* ⊢ *e* →* *e*
| *rs-trans*: ⟦ *CT* ⊢ *e* → *e′*; *CT* ⊢ *e′* →* *e″* ⟧ ⟹ *CT* ⊢ *e* →* *e″*

end

2 FJAux: Auxiliary Lemmas

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

2.1 Non-FJ Lemmas

2.1.1 Lists

lemma *mem-ith*:
assumes $ei \in set\ es$
shows $\exists\ el\ er.\ es = el@ei\#er$
⟨*proof*⟩

lemma *ith-mem*: $\bigwedge i.\ [\![\ i < length\ es\ ]\!] \Longrightarrow es!i \in set\ es$
⟨*proof*⟩

2.1.2 Maps

lemma *map-shuffle*:
assumes $length\ xs = length\ ys$
shows $[xs[\mapsto]ys,x\mapsto y] = [(xs@[x])[\mapsto](ys@[y])]$
⟨*proof*⟩

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

2.2 FJ Lemmas

2.2.1 Substitution

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

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

2.2.2 Lookup

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

and *lookup l f = o2*
shows *o1 = o2*
⟨*proof*⟩

lemma *lookup-true*:
lookup l f = Some r ⟹ f r
⟨*proof*⟩

lemma *lookup-hd*:
⟦ *length l > 0*; *f (l!0)* ⟧ *⟹ lookup l f = Some (l!0)*
⟨*proof*⟩

lemma *lookup-split*: *lookup l f = None ∨ (∃ h. lookup l f = Some h)*
⟨*proof*⟩

lemma *lookup-index*:
assumes *lookup l1 f = Some e*
shows ⋀*l2*. ∃ *i* < (*length l1*). *e = l1!i ∧* ((*length l1 = length l2*) ⟶ *lookup2 l1 l2 f = Some (l2!i)*)
⟨*proof*⟩

lemma *lookup2-index*:
⋀*l2*. ⟦ *lookup2 l1 l2 f = Some e*;
length l1 = length l2 ⟧ *⟹* ∃ *i* < (*length l2*). *e = (l2!i) ∧ lookup l1 f = Some (l1!i)*
⟨*proof*⟩

lemma *lookup-append*:
assumes *lookup l f = Some r*
shows *lookup (l@l′) f = Some r*
⟨*proof*⟩

lemma *method-typings-lookup*:
assumes *lookup-eq-Some*: *lookup M f = Some mDef*
and *M-ok*: *CT ⊢+ M OK IN C*
shows *CT ⊢ mDef OK IN C*
⟨*proof*⟩

2.2.3 Functional

These lemmas prove that several relations are actually functions

lemma *mtype-functional*:
assumes *mtype*(*CT*,*m*,*C*) = *Cs* → *C0*
and *mtype*(*CT*,*m*,*C*) = *Ds* → *D0*
shows *Ds*=*Cs ∧ D0*=*C0*
⟨*proof*⟩

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

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

lemma *fields-functional*:
assumes $fields(CT,C) = Cf$
and $CT\ OK$
shows $\bigwedge Cf'.\ [\![\ fields(CT,C) = Cf'\ ]\!] \Longrightarrow Cf = Cf'$
⟨*proof*⟩

2.2.4 Subtyping and Typing

lemma *typings-lengths*: **assumes** $CT;\Gamma \vdash+\ es{:}Cs$ **shows** $length\ es = length\ Cs$
⟨*proof*⟩

lemma *typings-index*:
assumes $CT;\Gamma \vdash+\ es{:}Cs$
shows $\bigwedge i.\ [\![\ i < length\ es\ ]\!] \Longrightarrow CT;\Gamma \vdash (es!i) : (Cs!i)$
⟨*proof*⟩

lemma *subtypings-index*:
assumes $CT \vdash+\ Cs <: Ds$
shows $\bigwedge i.\ [\![\ i < length\ Cs\ ]\!] \Longrightarrow CT \vdash (Cs!i) <: (Ds!i)$
⟨*proof*⟩

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

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

lemma *ith-typing*: $\bigwedge Cs.\ [\![\ CT;\Gamma \vdash+\ (es@(h\#t)) : Cs\ ]\!] \Longrightarrow CT;\Gamma \vdash h : (Cs!(length\ es))$
⟨*proof*⟩

lemma *ith-subtyping*: $\bigwedge Ds.\ [\![\ CT \vdash+\ (Cs@(h\#t)) <: Ds\ ]\!] \Longrightarrow CT \vdash h <: (Ds!(length\ Cs))$
⟨*proof*⟩

lemma *subtypings-refl*: $CT \vdash+\ Cs <: Cs$
⟨*proof*⟩

lemma *subtypings-trans*: $\bigwedge Ds\ Es.\ [\![\ CT \vdash+\ Cs <:\ Ds;\ \ CT \vdash+\ Ds <:\ Es\ ]\!] \Longrightarrow$
$CT \vdash+\ Cs <:\ Es$
⟨*proof*⟩

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

lemma *mem-typings*:
$\bigwedge Cs.\ [\![\ CT;\Gamma \vdash+\ es{:}Cs;\ ei \in set\ es]\!] \Longrightarrow \exists\, Ci.\ CT;\Gamma \vdash ei{:}Ci$
⟨*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*:
$CT \vdash+\ As <:\ Bs \Longrightarrow length\ As = length\ Bs$
⟨*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 $\bigwedge D.\ [\![\ CT \vdash D\ \neg<:\ C;\ \ CT \vdash C\ \neg<:\ D]\!] \Longrightarrow CT \vdash A\ \neg<:\ D$
⟨*proof*⟩

2.2.5 Sub-Expressions

lemma *isubexpr-typing*:
assumes $e1 \in isubexprs(e0)$
shows $\bigwedge C.\ [\![\ CT;Map.empty \vdash e0 : C\ ]\!] \Longrightarrow \exists\, D.\ CT;Map.empty \vdash e1 : D$
⟨*proof*⟩

lemma *subexpr-typing*:

assumes *e1* ∈ *subexprs*(*e0*)
shows ⋀*C*. ⟦ *CT*;*Map.empty* ⊢ *e0* : *C* ⟧ ⟹ ∃ *D*. *CT*;*Map.empty* ⊢ *e1* : *D*
⟨*proof*⟩

lemma *isubexpr-reduct*:
assumes *d1* ∈ *isubexprs*(*e1*)
shows ⋀*d2*. ⟦ *CT* ⊢ *d1* → *d2* ⟧ ⟹ ∃ *e2*. *CT* ⊢ *e1* → *e2*
⟨*proof*⟩

lemma *subexpr-reduct*:
assumes *d1* ∈ *subexprs*(*e1*)
shows ⋀*d2*. ⟦ *CT* ⊢ *d1* → *d2* ⟧ ⟹ ∃ *e2*. *CT* ⊢ *e1* → *e2*
⟨*proof*⟩

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* :: *nat list*
assumes *mtype*(*CT*,*m*,*C*) = *Cs* → *C0*
shows ∃ *xs e*. *mbody*(*CT*,*m*,*C*) = *xs* . *e* ∧ *length xs* = *length Cs*
⟨*proof*⟩

lemma *mtype-mbody-length*:
assumes *mt*:*mtype*(*CT*,*m*,*C*) = *Cs* → *C0*
and *mb*:*mbody*(*CT*,*m*,*C*) = *xs* . *e*
shows *length xs* = *length Cs*
⟨*proof*⟩

3.2 Method Types and Field Declarations of Subtypes

lemma *A-1-1*:
assumes *CT* ⊢ *C* <: *D* **and** *CT OK*
shows (*mtype*(*CT*,*m*,*D*) = *Cs* → *C0*) ⟹ (*mtype*(*CT*,*m*,*C*) = *Cs* → *C0*)
⟨*proof*⟩

lemma *sub-fields*:
assumes *CT* ⊢ *C* <: *D*

shows $\bigwedge Dg.\ \mathit{fields}(CT,D) = Dg \Longrightarrow \exists\, Cf.\ \mathit{fields}(CT,C) = (Dg@Cf)$
⟨*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 $\mathit{length}\ xs = \mathit{length}\ ds$
and $\mathit{length}\ Bs = \mathit{length}\ ds$
and $\exists\, As.\ CT;\Gamma 1 \vdash+\ ds : As \wedge CT \vdash+\ As <: Bs$
shows $CT;\Gamma \vdash+\ es{:}Ds \Longrightarrow \exists\, Cs.\ (CT;\Gamma 1 \vdash+\ ([ds/xs]es){:}Cs \wedge CT \vdash+\ Cs <: Ds)$ (**is** *?TYPINGS* $\Longrightarrow$ *?P1*)
and $CT;\Gamma \vdash e{:}D \Longrightarrow \exists\, C.\ (CT;\Gamma 1 \vdash ((ds/xs)e){:}C \wedge CT \vdash C <: D)$ (**is** *?TYPING* $\Longrightarrow$ *?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 $(CT;\Gamma 2 \vdash+\ es : Cs) \Longrightarrow (CT;\Gamma 1 +\!\!+ \Gamma 2 \vdash+\ es : Cs)$ (**is** *?P1* $\Longrightarrow$ *?P2*)
and $CT;\Gamma 2 \vdash e : C \Longrightarrow CT;\Gamma 1 +\!\!+ \Gamma 2 \vdash e : C$ (**is** *?Q1* $\Longrightarrow$ *?Q2*)
⟨*proof*⟩

3.5 Method Body Typing Lemma

lemma *A-1-4*:
assumes *ct-ok*: $CT\ OK$
and *mb*:$\mathit{mbody}(CT,m,C) = xs\ .\ e$
and *mt*:$\mathit{mtype}(CT,m,C) = Ds \to D$
shows $\exists\, D0\ C0.\ (CT \vdash C <: D0) \wedge$
$(CT \vdash C0 <: D) \wedge$
$(CT;[xs[\mapsto]Ds](\mathit{this} \mapsto D0) \vdash e : C0)$
⟨*proof*⟩

3.6 Subject Reduction Theorem

theorem *Thm-2-4-1*:
assumes $CT \vdash e \to e'$
and $CT\ OK$
shows $\bigwedge C.\ [\![\ CT;\Gamma \vdash e : C\ ]\!]$
$\Longrightarrow \exists\, C'.\ (CT;\Gamma \vdash e' : C' \wedge CT \vdash C' <: C)$
⟨*proof*⟩

3.7 Multi-Step Subject Reduction Theorem

corollary *Cor-2-4-1-multi*:
assumes $CT \vdash e \rightarrow* e'$
and $CT\ OK$
shows $\bigwedge C.\ [\![\ CT;\Gamma \vdash e : C\ ]\!] \Longrightarrow \exists\, C'.\ (CT;\Gamma \vdash e' : C' \wedge CT \vdash C' <: C)$
⟨*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 $CT;Map.empty \vdash e : C$
and $FieldProj\ (New\ C0\ es)\ fi \in subexprs(e)$
shows $\exists\, Cf\ fDef.\ fields(CT,\ C0) = Cf \wedge lookup\ Cf\ (\lambda fd.\ (vdName\ fd = fi)) = Some\ fDef$
⟨*proof*⟩

lemma *Thm-2-4-2-2*:
fixes *es ds* :: *exp list*
assumes $CT;Map.empty \vdash e : C$
and $MethodInvk\ (New\ C0\ es)\ m\ ds \in subexprs(e)$
shows $\exists\, xs\ e0.\ mbody(CT,m,C0) = xs\ .\ e0 \wedge length\ xs = length\ ds$
⟨*proof*⟩

lemma *closed-subterm-split*:
assumes $CT;\Gamma \vdash e : C$ **and** $\Gamma = Map.empty$
shows
$((\exists\, C0\ es\ fi.\ (FieldProj\ (New\ C0\ es)\ fi) \in subexprs(e))$
$\vee\ (\exists\, C0\ es\ m\ ds.\ (MethodInvk\ (New\ C0\ es)\ m\ ds) \in subexprs(e))$
$\vee\ (\exists\, C0\ D\ es.\ (Cast\ D\ (New\ C0\ es)) \in subexprs(e))$
$\vee\ val(e))$ (**is** *?F e* $\vee$ *?M e* $\vee$ *?C e* $\vee$ *?V e* **is** *?IH e*)
⟨*proof*⟩

3.9 Type Soundness Theorem

theorem *Thm-2-4-3*:
assumes *e-typ*: $CT;Map.empty \vdash e : C$
and *ct-ok*: $CT\ OK$
and *multisteps*: $CT \vdash e \rightarrow* e1$
and *no-step*: $\neg(\exists\, e2.\ CT \vdash e1 \rightarrow e2)$
shows $(val(e1) \wedge (\exists\, D.\ CT;Map.empty \vdash e1 : D \wedge CT \vdash D <: C))$
$\vee\ (\exists\, D\ C\ es.\ (Cast\ D\ (New\ C\ es) \in subexprs(e1) \wedge CT \vdash C\ \neg<:\ D))$
⟨*proof*⟩

end

theory *Execute*
imports *FJSound*
begin

4 Executing FeatherweightJava programs

We execute FeatherweightJava programs using the predicate compiler.

code-pred (*modes*: *i* => *i* => *i* => *bool*,
i => *i* => *o* => *bool as supertypes-of*) *subtyping* ⟨*proof*⟩

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*
⟨*proof*⟩

lemma [*code-pred-intro*]: *append xs ys zs* ⟹ *append* (*x*#*xs*) *ys* (*x*#*zs*)
⟨*proof*⟩

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

lemma *rc-invk-arg'*: *CT* ⊢ *ei* → *ei'* ⟹ *append el* (*ei* # *er*) *e'* ⟹ *append el* (*ei'* # *er*) *e''* ⟹
CT ⊢ *MethodInvk e m e'* → *MethodInvk e m e''*
⟨*proof*⟩

lemma *rc-new-arg'*: *CT* ⊢ *ei* → *ei'* ⟹ *append el* (*ei* # *er*) *e* ⟹ *append el* (*ei'* # *er*) *e'*
==> *CT* ⊢ *New C e* → *New C e'*
⟨*proof*⟩

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

code-pred (*modes*: *i* => *i* => *i* => *bool*, *i* => *i* => *o* => *bool as reduce*) *reduction*
⟨*proof*⟩

thm *reduction.equation*

code-pred *reductions* ⟨*proof*⟩

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 = (¬ (∀ *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*)
⟨*proof*⟩

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

lemma [*code-pred-intro*]:
mtype(*CT*,*m*,*D*) = *Ds* → *D0* ⟹ *C* ≠ *D0* ⟹ *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
classA-Def = (| *cName* = *A*, *cSuper* = *Object*, *cFields* = [], *cConstructor* = (|*kName* = *A*, *kParams* = [], *kSuper* = [], *kInits* = []|), *cMethods* = []|)

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

abbreviation *ffst* :: *varName*
where
ffst == *4*

abbreviation *fsnd* :: *varName*
where
fsnd == *5*

abbreviation *setfst* :: *methodName*
where
setfst == *6*

abbreviation *newfst* :: *varName*
where
newfst == *7*

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

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

[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/.