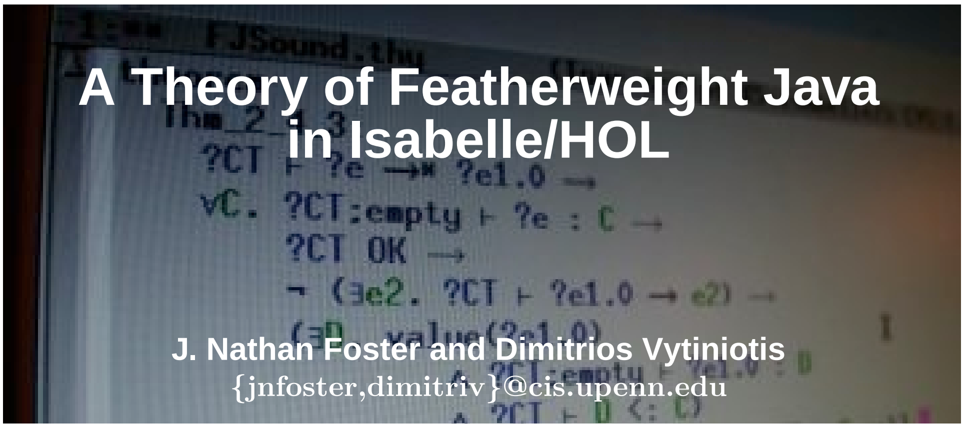

A Theory of Featherweight Java in Isabelle/HOL

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

lemmas *in-set-code*[*code unfold*] = *mem-iff*[*symmetric*, *THEN eq-reflection*]

1.1 Syntax

We use a named representation for terms: variables, method names, and class names, are all represented as `nat`s. 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

types *varName* = *nat*
types *methodName* = *nat*
types *className* = *nat*
record *varDef* =

vdName :: *varName*
vdType :: *className*
types *varCtx* = *varName* ⇀ *className*

1.1.2 Constants

consts
Object :: *className*
this :: *varName*
defs
Object : *Object* == *0*
this : *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

types *classTable* = *className* ⇀ *classDef*

1.2 Sub-expression Relation

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

consts
  $\mathit{isubexprs} :: (\mathit{exp} * \mathit{exp})\ \mathit{set}$
syntax
  $\text{-}\mathit{isubexprs} :: [\mathit{exp},\mathit{exp}] \Rightarrow \mathit{bool}\ \ (\text{-} \in \mathit{isubexprs}'(\text{-}')\ [80,80]\ 80)$
translations
  $e' \in \mathit{isubexprs}(e) \rightleftharpoons (e',e) \in \mathit{isubexprs}$
inductive *isubexprs*
intros
$\mathit{se\text{-}field}\ \ : e \in \mathit{isubexprs}(\mathit{FieldProj}\ e\ \mathit{fi})$
$\mathit{se\text{-}invkrecv} : e \in \mathit{isubexprs}(\mathit{MethodInvk}\ e\ m\ \mathit{es})$
$\mathit{se\text{-}invkarg}\ : [\![\ \mathit{ei} \in \mathit{set}\ \mathit{es}\ ]\!] \Longrightarrow \mathit{ei} \in \mathit{isubexprs}(\mathit{MethodInvk}\ e\ m\ \mathit{es})$
$\mathit{se\text{-}newarg}\ : [\![\ \mathit{ei} \in \mathit{set}\ \mathit{es}\ ]\!] \Longrightarrow \mathit{ei} \in \mathit{isubexprs}(\mathit{New}\ C\ \mathit{es})$
$\mathit{se\text{-}cast}\ \ : e \in \mathit{isubexprs}(\mathit{Cast}\ C\ e)$

consts
 $\mathit{subexprs} :: (\mathit{exp} * \mathit{exp})\ \mathit{set}$
syntax
  $\text{-}\mathit{subexprs} :: [\mathit{exp},\mathit{exp}] \Rightarrow \mathit{bool}\ \ (\text{-} \in \mathit{subexprs}'(\text{-}')\ [80,80]\ 80)$
translations
  $e' \in \mathit{subexprs}(e) \rightleftharpoons (e',e) \in \mathit{isubexprs}\hat{}*$

1.3 Values

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

consts
  $\mathit{vals} :: (\mathit{exp}\ \mathit{list})\ \mathit{set}$
  $\mathit{val} :: \mathit{exp}\ \mathit{set}$
syntax
  $\text{-}\mathit{vals} :: [\mathit{exp}\ \mathit{list}] \Rightarrow \mathit{bool}\ (\mathit{vals}'(\text{-}')\ [80]\ 80)$
  $\text{-}\mathit{val} :: [\mathit{exp}] \Rightarrow \mathit{bool}\ (\mathit{val}'(\text{-}')\ [80]\ 80)$
translations
  $\mathit{val}(v) \rightleftharpoons v \in \mathit{val}$
  $\mathit{vals}(\mathit{vl}) \rightleftharpoons \mathit{vl} \in \mathit{vals}$
inductive *vals val*
intros
   $\mathit{vals\text{-}nil} : \mathit{vals}([])$
   $\mathit{vals\text{-}cons} : [\![\ \mathit{val}(\mathit{vh});\ \mathit{vals}(\mathit{vt})\ ]\!] \Longrightarrow \mathit{vals}((\mathit{vh}\ \#\ \mathit{vt}))$
   $\mathit{val} : [\![\ \mathit{vals}(\mathit{vs})\ ]\!] \Longrightarrow \mathit{val}(\mathit{New}\ C\ \mathit{vs})$

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.

consts
substs :: $(varName \rightharpoonup exp) \Rightarrow exp \Rightarrow exp$
subst-list1 :: $(varName \rightharpoonup exp) \Rightarrow exp\ list \Rightarrow exp\ list$
subst-list2 :: $(varName \rightharpoonup exp) \Rightarrow exp\ list \Rightarrow exp\ list$
syntax
-substs :: $[varName\ list] \Rightarrow [exp\ list] \Rightarrow [exp] \Rightarrow exp$ ('(-/-')- [80,80,80] 80)
-subst-list :: $[varName\ list] \Rightarrow [exp\ list] \Rightarrow [exp\ list] \Rightarrow exp\ list$ ('[-/-']- [80,80,80] 80)
translations
$[ds/xs]es \rightleftharpoons map\ (substs\ (map\text{-}upds\ empty\ xs\ ds))\ es$
$(ds/xs)e \rightleftharpoons substs\ (map\text{-}upds\ empty\ xs\ ds)\ e$

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

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

consts *lookup* :: $'a\ list \Rightarrow ('a \Rightarrow bool) \Rightarrow 'a\ option$
primrec
$lookup\ []\ P = None$
$lookup\ (h\#t)\ P = (if\ P\ h\ then\ Some\ h\ else\ lookup\ t\ P)$

consts *lookup2* :: $'a\ list \Rightarrow 'b\ list \Rightarrow ('a \Rightarrow bool) \Rightarrow 'b\ option$
primrec
$lookup2\ []\ l2\ P = None$
$lookup2\ (h1\#t1)\ l2\ P = (if\ P\ h1\ then\ Some(hd\ l2)\ else\ lookup2\ t1\ (tl\ l2)\ P)$

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).

constdefs *varDefs-names* :: $varDef\ list \Rightarrow varName\ list$
$varDefs\text{-}names == map\ vdName$

constdefs *varDefs-types* :: $varDef\ list \Rightarrow className\ list$
$varDefs\text{-}types == map\ vdType$

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 $CT \vdash +Cs{<:}Ds$.

consts *subtyping* :: (*classTable* ∗ *className* ∗ *className*) *set*
subtypings :: (*classTable* ∗ *className list* ∗ *className list*) *set*
syntax
-subtyping :: [*classTable*, *className*, *className*] ⇒ *bool* (- ⊢ - <: - [*80*,*80*,*80*] *80*)
-subtypings :: [*classTable*, *className list*, *className list*] ⇒ *bool* (- ⊢+ - <: - [*80*,*80*,*80*] *80*)
-neg-subtyping :: [*classTable*, *className*, *className*] ⇒ *bool* (- ⊢ - ¬<: - [*80*,*80*,*80*] *80*)
translations
$CT \vdash S <: T \rightleftharpoons (CT,S,T) \in \mathit{subtyping}$
$CT \vdash\!+\ Ss <: Ts \rightleftharpoons (CT,Ss,Ts) \in \mathit{subtypings}$
$CT \vdash S \neg{<:}\ T \rightleftharpoons (CT,S,T) \notin \mathit{subtyping}$
inductive *subtyping*
intros
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 : $\llbracket\ CT(C) = \mathit{Some}(\mathit{CDef});\ \mathit{cSuper}\ \mathit{CDef} = D\ \rrbracket \Longrightarrow CT \vdash C <: D$

inductive *subtypings*
intros
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.

consts *fields* :: (*classTable* ∗ *className* ∗ *varDef list*) *set*
syntax
-fields :: [*classTable*, *className*, *varDef list*] ⇒ *bool* (*fields*′(-,-′) = - [*80*,*80*,*80*] *80*)
translations
$\mathit{fields}(CT,C) = Cf \rightleftharpoons (CT,C,Cf) \in \mathit{fields}$
inductive *fields*
intros
f-obj:
$\mathit{fields}(CT,\mathit{Object}) = []$
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 $\texttt{mtype}(CT, m, C) = Cs \rightarrow C_0$ 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.

consts *mtype* :: (*classTable* $*$ *methodName* $*$ *className* $*$ ((*className list*) $*$ *className*)) *set*
syntax
-mtype :: [*classTable*, *methodName*, *className*, *className list*, *className*] $\Rightarrow$ *bool* (*mtype'*(-,-,-') = - $\rightarrow$ - [*80*,*80*,*80*,*80*] *80*)
translations
$mtype(CT,m,C) = Cs \rightarrow C0 \rightleftharpoons (CT,m,C,(Cs,C0)) \in mtype$
inductive *mtype*
intros
mt-class:
$\llbracket$ $CT(C) = Some(CDef)$;
$lookup\ (cMethods\ CDef)\ (\lambda md.(mName\ md = m)) = Some(mDef)$;
$varDefs\text{-}types\ (mParams\ mDef) = Bs$;
$mReturn\ mDef = B$ $\rrbracket$
$\Longrightarrow$ $mtype(CT,m,C) = Bs \rightarrow B$

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 $\texttt{mbody}(CT, m, C) = xs.e_0$ 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.

consts *mbody* :: (*classTable* $*$ *methodName* $*$ *className* $*$ (*varName list* $*$ *exp*)) *set*
syntax
-mbody :: [*classTable*, *methodName*, *className*, *varName list*, *exp*] $\Rightarrow$ *bool* (*mbody'*(-,-,-') = - . - [*80*,*80*,*80*,*80*] *80*)
translations
$mbody(CT,m,C) = xs\ .\ e \rightleftharpoons (CT,m,C,(xs,e)) \in mbody$

inductive *mbody*
intros
mb-class:
$[\![$ $CT(C) = Some(CDef)$;
$lookup\ (cMethods\ CDef)\ (\lambda md.(mName\ md = m)) = Some(mDef)$;
$varDefs\text{-}names\ (mParams\ mDef) = xs$;
$mBody\ mDef = e$ $]\!]$
$\Longrightarrow mbody(CT,m,C) = xs\ .\ e$

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

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.

consts
$typing$ :: $(classTable * varCtx * exp * className)\ set$
$typings$ :: $(classTable * varCtx * exp\ list * className\ list)\ set$
syntax
-typing :: [*classTable*, *varCtx*, *exp list*, *className*] $\Rightarrow$ *bool* (-;- $\vdash$ - : - [*80*,*80*,*80*,*80*] *80*)
-typings :: [*classTable*, *varCtx*, *exp list*, *className*] $\Rightarrow$ *bool* (-;- $\vdash$+ - : - [*80*,*80*,*80*,*80*] *80*)
translations
$CT;\Gamma \vdash e : C \rightleftharpoons (CT,\Gamma,e,C) \in typing$
$CT;\Gamma \vdash + \ es : Cs \rightleftharpoons (CT,\Gamma,es,Cs) \in typings$

inductive *typings typing*
intros
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 :
$[\![$ $CT;\Gamma \vdash e0 : C0$;
$fields(CT,C0) = Cf$;
$lookup\ Cf\ (\lambda fd.(vdName\ fd = fi)) = Some(fDef)$;

$vdType\ fDef = Ci$ $\rrbracket$
$\Longrightarrow CT;\Gamma \vdash FieldProj\ e0\ fi : Ci$

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

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

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

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 $\bigwedge C0\ CT\ Cf\ Ci\ \Gamma\ e0\ fDef\ fi.\ \llbracket CT;\Gamma \vdash e0 : C0;\ P\ CT\ \Gamma\ e0\ C0;\ (CT,\ C0,\ Cf) \in FJDefs.fields;\ lookup\ Cf\ (\lambda fd.\ vdName\ fd = fi) = Some\ fDef;\ vdType\ fDef = Ci\rrbracket \Longrightarrow P\ CT\ \Gamma\ (FieldProj\ e0\ fi)\ Ci$
and $\bigwedge C\ C0\ CT\ Cs\ Ds\ \Gamma\ e0\ es\ m.\ \llbracket CT;\Gamma \vdash e0 : C0;\ P\ CT\ \Gamma\ e0\ C0;\ (CT,\ m,\ C0,\ Ds,\ C) \in mtype;\ CT;\Gamma \vdash+ es : Cs;\ \bigwedge i\ .\ \llbracket\ i < length\ es\ \rrbracket \Longrightarrow\ P\ CT\ \Gamma\ (es!i)\ (Cs!i);\ CT \vdash+ Cs <: Ds;\ length\ es = length\ Ds\rrbracket \Longrightarrow P\ CT\ \Gamma\ (MethodInvk\ e0\ m\ es)\ C$

and $\bigwedge C\ CT\ Cs\ Df\ Ds\ \Gamma\ es.\ [\![(CT,\ C,\ Df) \in FJDefs.fields;\ length\ es = length\ Df;\ varDefs\text{-}types\ Df = Ds;\ CT;\Gamma \vdash{+}\ es : Cs;\ \bigwedge i.\ [\![\ i < length\ es\ ]\!] \Longrightarrow P\ CT\ \Gamma\ (es!i)\ (Cs!i);\ CT \vdash{+}\ Cs <: Ds]\!] \Longrightarrow P\ CT\ \Gamma\ (New\ C\ es)\ C$
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.

consts *method-typing* :: (*classTable* ∗ *methodDef* ∗ *className*) *set*
method-typings :: (*classTable* ∗ *methodDef list* ∗ *className*) *set*
syntax
-method-typing :: [*classTable*, *methodDef*, *className*] ⇒ *bool* (- ⊢ - *OK IN* - [*80*,*80*,*80*] *80*)
-method-typings :: [*classTable*, *methodDef list*, *className*] ⇒ *bool* (- ⊢+ - *OK IN* - [*80*,*80*,*80*] *80*)
translations
$CT \vdash md\ OK\ IN\ C \rightleftharpoons (CT,md,C) \in method\text{-}typing$
$CT \vdash{+}\ mds\ OK\ IN\ C \rightleftharpoons (CT,mds,C) \in method\text{-}typings$

inductive *method-typing*
intros
m-typing:
$[\![\ CT(C) = Some(CDef);$
$cName\ CDef = C;$
$cSuper\ CDef = D;$
$mName\ mDef = m;$
$lookup\ (cMethods\ CDef)\ (\lambda md.(mName\ md = m)) = Some(mDef);$
$mReturn\ mDef = C0;\ mParams\ mDef = Cxs;\ mBody\ mDef = e0;$
$varDefs\text{-}types\ Cxs = Cs;$
$varDefs\text{-}names\ Cxs = xs;$
$\Gamma = (map\text{-}upds\ empty\ xs\ Cs)(this \mapsto C);$
$CT;\Gamma \vdash e0 : E0;$
$CT \vdash E0 <: C0;$
$\forall Ds\ D0.\ (mtype(CT,m,D) = Ds \rightarrow D0) \longrightarrow (Cs{=}Ds \wedge C0{=}D0)\ ]\!]$
$\Longrightarrow CT \vdash mDef\ OK\ IN\ C$

inductive *method-typings*
intros
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*`OK` if its constructor initializes each field, and all of its methods are well-typed.

consts *class-typing* :: (*classTable* $*$ *classDef*) *set*
syntax
-class-typing :: [*classTable*, *classDef*] $\Rightarrow$ *bool* (- $\vdash$ - *OK* [*80*,*80*] *80*)
translations
CT $\vdash$ *cd OK* $\rightleftharpoons$ (*CT*,*cd*) $\in$ *class-typing*

inductive *class-typing*
intros
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* $]\!]$
$\Longrightarrow$ *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*.

consts *ct-typing* :: *classTable set*
syntax
-ct-typing :: *classTable* $\Rightarrow$ *bool* (- *OK 80*)
translations
CT OK $\rightleftharpoons$ *CT* $\in$ *ct-typing*
inductive *ct-typing*
intros
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 $CT \vdash e \rightarrow e'$ and $CT \vdash e \rightarrow^* e'$ respectively.

consts *reduction* :: $(classTable * exp * exp)$ *set*
syntax
$\;\;$ *-reduction* :: $[classTable,\ exp,\ exp] \Rightarrow bool$ $(\text{-} \vdash \text{-} \rightarrow \text{-}\ [80,80,80]\ 80)$

translations
$\;\;CT \vdash e \rightarrow e' \rightleftharpoons (CT,e,e') \in reduction$
inductive *reduction*
intros

r-field:
$\;\;[\![\ fields(CT,C) = Cf;$
$\;\;\;\;lookup2\ Cf\ es\ (\lambda fd.(vdName\ fd = fi)) = Some(ei)\ ]\!]$
$\;\;\Longrightarrow CT \vdash FieldProj\ (New\ C\ es)\ fi \rightarrow ei$

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

r-cast:
$\;\;[\![\ CT \vdash C <: D\ ]\!]$
$\;\;\Longrightarrow CT \vdash Cast\ D\ (New\ C\ es) \rightarrow New\ C\ es$

rc-field:
$\;\;[\![\ CT \vdash e0 \rightarrow e0'\ ]\!]$
$\;\;\Longrightarrow CT \vdash FieldProj\ e0\ f \rightarrow FieldProj\ e0'\ f$

rc-invk-recv:
$\;\;[\![\ CT \vdash e0 \rightarrow e0'\ ]\!]$
$\;\;\Longrightarrow CT \vdash MethodInvk\ e0\ m\ es \rightarrow MethodInvk\ e0'\ m\ es$

rc-invk-arg:
$\;\;[\![\ CT \vdash ei \rightarrow ei'\ ]\!]$
$\;\;\Longrightarrow CT \vdash MethodInvk\ e0\ m\ (el@ei\#er) \rightarrow MethodInvk\ e0\ m\ (el@ei'\#er)$

rc-new-arg:
$\;\;[\![\ CT \vdash ei \rightarrow ei'\ ]\!]$
$\;\;\Longrightarrow CT \vdash New\ C\ (el@ei\#er) \rightarrow New\ C\ (el@ei'\#er)$

rc-cast:
$\;\;[\![\ CT \vdash e0 \rightarrow e0'\ ]\!]$
$\;\;\Longrightarrow CT \vdash Cast\ C\ e0 \rightarrow Cast\ C\ e0'$

consts *reductions* :: $(classTable * exp * exp)$ *set*
syntax

$-reductions$ :: [$classTable$, exp, exp] $\Rightarrow$ $bool$ (- $\vdash$ - $\rightarrow*$ - [$80,80,80$] 80)
translations
$CT \vdash e \rightarrow* e' \rightleftharpoons (CT,e,e') \in reductions$
inductive *reductions*
intros
rs-refl: $CT \vdash e \rightarrow* e$
rs-trans: $[\![\ CT \vdash e \rightarrow e';\ CT \vdash e' \rightarrow* e''\ ]\!] \Longrightarrow\ CT \vdash e \rightarrow* e''$

end

2 FJAux: Auxiliary Lemmas

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

2.1 Non-FJ Lemmas

2.1.1 Lists

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

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

2.1.2 Maps

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

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 $\exists\, i.(?P1\ i\ As) \wedge (?P2\ i\ As) \wedge (\forall\, Bs::('c\ list).(?P3\ i\ xs\ As\ Bs)))$
$\langle proof \rangle$

2.2 FJ Lemmas

2.2.1 Substitution

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

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

2.2.2 Lookup

lemma *lookup-functional*:
assumes $\mathit{lookup}\ l\ f = \mathit{o1}$
and $\mathit{lookup}\ l\ f = \mathit{o2}$
shows $\mathit{o1} = \mathit{o2}$
$\langle \mathit{proof} \rangle$

lemma *lookup-true*:
$\mathit{lookup}\ l\ f = \mathit{Some}\ r \Longrightarrow f\ r$
$\langle \mathit{proof} \rangle$

lemma *lookup-hd*:
$[\![\ \mathit{length}\ l > 0;\ f\ (l!0)\ ]\!] \Longrightarrow \mathit{lookup}\ l\ f = \mathit{Some}\ (l!0)$
$\langle \mathit{proof} \rangle$

lemma *lookup-split*: $\mathit{lookup}\ l\ f = \mathit{None} \lor (\exists\,h.\ \mathit{lookup}\ l\ f = \mathit{Some}\ h)$
$\langle \mathit{proof} \rangle$

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

lemma *lookup2-index*:
$\bigwedge \mathit{l2}.\ [\![\ \mathit{lookup2}\ \mathit{l1}\ \mathit{l2}\ f = \mathit{Some}\ e;$
$\mathit{length}\ \mathit{l1} = \mathit{length}\ \mathit{l2}\ ]\!] \Longrightarrow \exists\,i < (\mathit{length}\ \mathit{l2}).\ e = (\mathit{l2}!i) \land \mathit{lookup}\ \mathit{l1}\ f = \mathit{Some}\ (\mathit{l1}!i)$
$\langle \mathit{proof} \rangle$

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

lemma *method-typings-lookup*:
assumes *lookup-eq-Some*: $\mathit{lookup}\ M\ f = \mathit{Some}\ \mathit{mDef}$

and *M-ok*: $CT \vdash\!+ M\ OK\ IN\ C$
shows $CT \vdash mDef\ OK\ IN\ C$
⟨*proof*⟩

2.2.3 Functional

These lemmas prove that several relations are actually functions

lemma *mtype-functional*:
assumes $mtype(CT,m,C) = Cs \rightarrow C0$
and $mtype(CT,m,C) = Ds \rightarrow D0$
shows $Ds=Cs \wedge D0=C0$
⟨*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 $CT;\Gamma \vdash+ (es@[e]) : (Cs@[C])$
$\langle proof \rangle$

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

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

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

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

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

lemma *mem-typings*:
$\bigwedge Cs.\ [\![\ CT;\Gamma \vdash+ es{:}Cs;\ ei \in set\ es]\!] \Longrightarrow \exists\, Ci.\ CT;\Gamma \vdash ei{:}Ci$
$\langle proof \rangle$

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** $CT \vdash As!i <: Bs!i$
$\langle proof \rangle$

lemma *subtypings-length*:
$CT \vdash+ As <: Bs \Longrightarrow length\ As = length\ Bs$
$\langle proof \rangle$

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

$\langle proof \rangle$

lemma *not-subtypes*:
assumes $CT \vdash A <: C$
shows $\bigwedge D.\ [\![\ CT \vdash D \neg<: C;\ \ CT \vdash C \neg<: D]\!] \Longrightarrow CT \vdash A \neg<: D$
$\langle proof \rangle$ **pr**
$\langle proof \rangle$

2.2.5 Sub-Expressions

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

lemma *subexpr-typing*:
assumes $e1 \in subexprs(e0)$
shows $\bigwedge C.\ [\![\ CT;empty \vdash e0 : C\ ]\!] \Longrightarrow \exists D.\ CT;empty \vdash e1 : D$
$\langle proof \rangle$

lemma *isubexpr-reduct*:
assumes $d1 \in isubexprs(e1)$
shows $\bigwedge d2.\ [\![\ CT \vdash d1 \rightarrow d2\ ]\!] \Longrightarrow \exists e2.\ CT \vdash e1 \rightarrow e2$
$\langle proof \rangle$

lemma *subexpr-reduct*:
assumes $d1 \in subexprs(e1)$
shows $\bigwedge d2.\ [\![\ CT \vdash d1 \rightarrow d2\ ]\!] \Longrightarrow \exists e2.\ CT \vdash e1 \rightarrow e2$
$\langle proof \rangle$

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*:
assumes $mtype(CT,m,C) = Cs \rightarrow C0$
shows $\exists xs\ e.\ mbody(CT,m,C) = xs\ .\ e \wedge length\ xs = length\ Cs$
$\langle proof \rangle$

lemma *mtype-mbody-length*:
assumes $mt{:}mtype(CT,m,C) = Cs \rightarrow C0$
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 $CT \vdash C <: D$ **and** $CT\ OK$
shows $(mtype(CT,m,D) = Cs \rightarrow C0) \Longrightarrow (mtype(CT,m,C) = Cs \rightarrow C0)$
⟨*proof*⟩

lemma *sub-fields*:
assumes $CT \vdash C <: D$
shows $\bigwedge Dg.\ fields(CT,D) = Dg \Longrightarrow \exists\, Cf.\ fields(CT,C) = (Dg@Cf)$
⟨*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 $length\ xs = length\ ds$
and $length\ Bs = length\ ds$
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 $mb{:}mbody(CT,m,C) = xs\ .\ e$

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 $\bigwedge C.\ [\![\ CT;\Gamma \vdash e : C\ ]\!]$
$\Longrightarrow \exists\, C'.\ (CT;\Gamma \vdash e' : C' \wedge CT \vdash C' <: C)$
⟨*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 $CT;empty \vdash e : C$
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*:
assumes $CT;empty \vdash e : C$
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** $\Gamma = empty$
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*: $CT;empty \vdash e : C$
  and *ct-ok*: CT OK
  and *multisteps*: $CT \vdash e \rightarrow* e1$
  and *no-step*: $\neg(\exists e2.\ CT \vdash e1 \rightarrow e2)$
  shows $(val(e1) \wedge (\exists D.\ CT;empty \vdash e1 : D \wedge CT \vdash D <: C))$
    $\vee (\exists D\ C\ es.\ (Cast\ D\ (New\ C\ es) \in subexprs(e1) \wedge CT \vdash C \neg<: D))$
⟨*proof*⟩

end

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/.