

Featherweight Defenders

Brian Goetz

January 19, 2011

1 Introduction

As a means of modeling the semantics of *virtual extension methods* (also known as *defender methods*) in Java, we describe a lightweight model of Java in the style of *Featherweight Java* (Pierce et al.), called *Featherweight Defenders* (or FD).

In this model, there are classes and interfaces, with single inheritance of classes and multiple inheritance of interfaces. Each class or interface may or may not specify a single method $m()$, which has no arguments but has a specified return type. Interface methods may have specified defaults or not, methods can be covariantly overridden, class methods may be abstract (indicated by the absence of a method body), and concrete class methods may be reabstracted.

I believe that this includes all the inheritance features that are relevant to resolution of extension methods.

2 Syntax

The metavariables A , B , C , and D (and their derivatives) range over class names and the metavariables I , and J range over interface names. The metavariables T , U , and V range over all types. The metavariable k ranges over a set of nominal identifiers, typed in the static typing context Γ . The metavariable S ranges over sets of functions. The metavariable e ranges over expressions. Figure 1 shows the syntactic forms for FD.

```
 $T ::= \text{Object} \mid C \mid I$   
 $K ::= \text{class } C \text{ extends } D \text{ implements } I_1, \dots, I_n \{ [ T m() [ k ] ] \}$   
 $L ::= \text{interface } I \text{ extends } I_1, \dots, I_n \{ [ T m() [ \text{default } k ] ] \}$   
 $e ::= x \mid \text{e.m}() \mid \text{new } C()$ 
```

Figure 1: FD language syntax

For simplicity of modeling, we have distorted the syntax somewhat to make the declaration of a method body in a class and a method default in an interface more similar. The absence of a method body k in a class definition indicates that the method is abstract.

The identifiers k represent names; set-theoretic operations on sets of identifiers treat identically named identifiers as being the same element. (So, for example, combining $\{k\} \cup \{k\}$ simply yields the set $\{k\}$.) For set-theoretic operations, nil is treated as the empty set.

3 Ancillary functions

As in Featherweight Java, we use an ancillary function $mtype(T)$ to indicate the type of $m()$ in type T . If $m()$ is not a member of T , then $mtype(T)$ will be nil . We introduce an ancillary function $mdef(I)$ to indicate the identity of the default method, if any, for $m()$ in interface I . Similarly, we introduce $mbody(C)$ to indicate the identity of a method body for $m()$ in class C (which may have been declared in C or in a superclass.) Finally, we use $mres(C)$ to indicate the resolution of $m()$ in C , which may have come from a body declared in C or a superclass, or may have come from a default in an interface.

We introduce the function $interfaces(T)$ to record the superinterfaces of a class or interface, which will be used when we compute the set of candidate defaults for a method in a class.

We also define a function lb_{inc} for computing an *inclusive lower bound* for a set of types. The inclusive lower bound for a set of types T_1, \dots, T_n is the lower bound of T_1, \dots, T_n if T_1, \dots, T_n contains its lower bound, and nil otherwise. This is used to determine whether a set of types can contribute a consistent return type for the method $m()$, and if so $lb_{inc}(T_1, \dots, T_n)$ evaluates to that most specific return type. We define $lb_{inc}(T_1, \dots, T_n)$ as follows:

$$lb_{inc}(T_1, \dots, T_n) = \begin{cases} mtype(T_i) & \text{if } \exists_i \text{ such that } mtype(T_i) \neq nil, \text{ and} \\ & \forall_{j \neq i} [mtype(T_j) = nil \vee mtype(T_i) <: mtype(T_j)] \\ nil & \text{otherwise} \end{cases}$$

Judgements may include conditions of the form $T = lb_{inc}(T_1, \dots, T_n)$. If lb_{inc} evaluates to nil , then these conditions are presumed to *not* hold.

Finally, we define $prune(I_1, \dots, I_n)$ as follows:

$$\begin{aligned} occludes(I, J) &= I <: J \wedge I \neq J \wedge mdef(I) \neq nil \\ prune(I_1, \dots, I_n) &= \{ I_i : \forall_{j \neq i} \neg occludes(I_j, I_i) \} \end{aligned}$$

4 Preliminaries

Figure 2 shows some general typing judgements needed by FD, and the subtyping judgements for classes and interfaces, as well as the base rules for the class

Object.

$$\begin{array}{c}
\text{S-REFL} \frac{}{T <: T} \\
\text{S-TRANS} \frac{T <: U \quad U <: S}{T <: S} \\
\text{T-SUB} \frac{\Gamma \vdash k : S \quad S <: T}{\Gamma \vdash k : T} \\
\text{T-OBJECT} \frac{}{\text{Object OK} \quad \text{mtype}(\text{Object}) = \text{nil}} \\
\text{S-CLASSDEF} \frac{\text{class } C \text{ extends } D \text{ implements } I_1, \dots, I_n \{ \dots \}}{C <: D \quad \forall_i C <: I_i \\ \text{interfaces}(C) = \text{interfaces}(D) \cup \bigcup_i \text{interfaces}(I_i)} \\
\text{S-INTDEF} \frac{\text{interface } I \text{ extends } I_1, \dots, I_n \{ \dots \}}{\forall_i I <: I_i \\ \text{interfaces}(I) = \{ I \} \cup \bigcup_i \text{interfaces}(I_i)}
\end{array}$$

Figure 2: Basic subtyping rules

5 Method typing

For each class or interface, the ancillary function $\text{mtype}(T)$ identifies the type of the function $\mathbf{m}()$ in T . The rule T-INVOKE shows how $\text{mtype}(C)$ is used in typing of method invocation expressions.

$$\begin{array}{c}
\text{T-VAR} \frac{}{\Gamma \vdash x : \Gamma(x)} \\
\text{T-NEW} \frac{C \text{ OK}}{\Gamma \vdash \text{new } C() : C} \\
\text{T-INVOKE} \frac{\Gamma \vdash e : C \quad C \text{ OK} \quad T = \text{mtype}(C)}{\Gamma \vdash e.\mathbf{m}() : T}
\end{array}$$

Figure 3: Expression typing

Figure 4 illustrates the rules for computing $\text{mtype}(I)$ and $\text{mdef}(I)$ for interfaces. The predicate $I \text{ OK}$ indicates that interface I is properly typed and

provides at most one appropriate default for $m()$.

$$\text{T-INTDEF} \frac{\begin{array}{l} \text{interface } I \text{ extends } I_1, \dots, I_n \{ T \ m() \ \text{default } k \} \\ \forall_i I_i \ \text{OK} \quad \Gamma \vdash k : T \\ \forall_i [\text{mtype}(I_i) = \text{nil} \vee T <: \text{mtype}(I_i)] \end{array}}{\text{mtype}(I) = T \quad \text{mdef}(I) = k \quad I \ \text{OK}}$$

$$\text{T-INTNODEF} \frac{\begin{array}{l} \text{interface } I \text{ extends } I_1, \dots, I_n \{ T \ m() \} \\ \forall_i I_i \ \text{OK} \\ \forall_i [\text{mtype}(I_i) = \text{nil} \vee T <: \text{mtype}(I_i)] \\ \left| \bigcup_{J \in \text{prune}(\text{interfaces}(I))} \{ \text{mdef}(J) \} \right| \leq 1 \end{array}}{\text{mtype}(I) = T \quad I \ \text{OK}}$$

$$\text{T-INTINH} \frac{\begin{array}{l} \text{interface } I \text{ extends } I_1, \dots, I_n \{ \} \\ \forall_i I_i \ \text{OK} \quad T = \text{lb}_{\text{inc}}(I_1, \dots, I_n) \\ \left| \bigcup_{J \in \text{prune}(\text{interfaces}(I))} \{ \text{mdef}(J) \} \right| \leq 1 \end{array}}{\text{mtype}(I) = T \quad I \ \text{OK}}$$

$$\text{T-INTNONE} \frac{\begin{array}{l} \text{interface } I \text{ extends } I_1, \dots, I_n \{ \} \\ \forall_i I_i \ \text{OK} \quad \forall_i \text{mtype}(I_i) = \text{nil} \end{array}}{\text{mtype}(I) = \text{nil} \quad I \ \text{OK}}$$

Figure 4: Interface method typing and defaults

6 Interface pruning

In FD, as in Java, it is allowable for a class or interface to extend an interface both directly and indirectly, as in the following example:

```
interface Collection { void m() default k }
interface Set extends Collection { void m() default l }
class MySet implements Set, Collection { }
```

Here, `MySet` implements `Collection` both directly and indirectly. This idiom is common as a documentation device, but in Java 7 and earlier the additional declaration of `Collection` has no effect, because it is already implicit in the extension of `Set`. This behavior should continue to hold true in the presence of extension methods.

The design of extension methods calls for “redundant” inheritance from less-specific interfaces (such as `Collection` in the example above) to not be considered further in the inheritance decision, except inasmuch as the less-specific interface has already contributed to its subinterface. If a class extends interfaces

I and J , where $I <: J$, J , and I contributes a default method for $m()$, then J is pruned from consideration in contributing a default. This is handled by the $prune(I_1, \dots, I_n)$ function in the method resolution rules for classes.

7 Class method typing

Figure 5 shows the rules for defining $mtype(C)$, $mbody(C)$, and $mres(C)$. Intuitively, these rules say that method bodies defined in a class take precedence over methods defined in superclasses or interfaces, that a method inherited from a superclass takes precedence over a default inherited from an interface, that methods can be covariantly overridden, that if multiple interfaces contribute a default, they must be identical to be considered, and that there must always be a consistent, most-specific return type for methods.

The predicate $C \text{ OK}$ indicates that C has *typed* correctly; it does not by itself mean that method resolution for $m()$ in C succeeds. For method resolution in C to succeed, we need both $C \text{ OK}$ and $mres(C) \neq nil$.

$$\begin{array}{c}
\text{T-CLASSCONC} \frac{\text{class } C \text{ extends } D \text{ implements } I_1, \dots, I_n \{ T \ m() \ k \}}{\Gamma \vdash k : T \quad \forall_i I_i \text{ OK} \quad D \text{ OK} \quad \forall_{U \in \{D, I_1, \dots, I_n\}} [\text{mtype}(U) = \text{nil} \vee T <: \text{mtype}(U)]} {\text{mtype}(C) = T \quad C \text{ OK} \quad \text{mbody}(C) = k} \\
\\
\text{T-CLASSABS} \frac{\text{class } C \text{ extends } D \text{ implements } I_1, \dots, I_n \{ T \ m() \}}{\forall_i I_i \text{ OK} \quad D \text{ OK} \quad \forall_{U \in \{D, I_1, \dots, I_n\}} [\text{mtype}(U) = \text{nil} \vee T <: \text{mtype}(U)]} {\text{mtype}(C) = T \quad C \text{ OK} \quad \text{mbody}(C) = \text{nil}} \\
\\
\text{T-CLASSINH} \frac{\text{class } C \text{ extends } D \text{ implements } I_1, \dots, I_n \{ \}}{\forall_i I_i \text{ OK} \quad D \text{ OK} \quad T = \text{lb}_{inc}(D, I_1, \dots, I_n)} {\text{mtype}(C) = T \quad C \text{ OK} \quad \text{mbody}(C) = \text{mbody}(D)} \\
\\
\text{T-CLASSNONE} \frac{\text{class } C \text{ extends } D \text{ implements } I_1, \dots, I_n \{ \}}{\forall_i I_i \text{ OK} \quad D \text{ OK} \quad \forall_i \text{mtype}(I_i) = \text{nil} \quad \text{mtype}(D) = \text{nil}} {\text{mtype}(C) = \text{nil} \quad C \text{ OK}} \\
\\
\text{R-BODY} \frac{\text{class } C \text{ extends } D \text{ implements } I_1, \dots, I_n \{ \dots \}}{C \text{ OK} \quad \text{mbody}(C) \neq \text{nil}} {\text{mres}(C) = \text{mbody}(C)} \\
\\
\text{R-DEFENDER} \frac{\text{class } C \text{ extends } D \text{ implements } I_1, \dots, I_n \{ \dots \}}{C \text{ OK} \quad \text{mbody}(C) = \text{nil} \quad S = \bigcup_{I \in \text{prune}(\text{interfaces}(C))} \{ \text{mdef}(I) \} \quad |S| = 1 \quad \exists_k k \in S \quad \Gamma \vdash k : T} {\text{mres}(C) = k}
\end{array}$$

Figure 5: Class method typing and resolution