Featherweight Defenders

Brian Goetz

January 31, 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 syntatic forms for FD.

$$
\begin{array}{rcl}
T & ::= & \texttt{Object} \mid C \mid I \\
K & ::= & \texttt{class}\ C\ \texttt{extends}\ D\ \texttt{implements}\ I_1, ..., I_n\ \{\ [\ T\ m()\ [\ k\ ]\ ]\ \} \\
L & ::= & \texttt{interface}\ I\ \texttt{extends}\ I_1, ..., I_n\ \{\ [\ T\ m()\ [\ \texttt{default}\ k\ ]\ ]\ \} \\
e & ::= & x \mid \texttt{e.m()} \mid \texttt{new C()}
\end{array}
$$

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 $T_1, ..., T_n$ is the lower bound of $T_1, ..., T_n$ if $T_1, ..., T_n$ 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 $lb_{inc}(T_1, ..., T_n)$ evaluates to that most specific return type. We define $lb_{inc}(T_1, ..., T_n)$ as follows:

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

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

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

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

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`.

$$\text{S-Refl}\ \frac{}{T <: T}$$

$$\text{S-Trans}\ \frac{T <: U \qquad U <: S}{T <: S}$$

$$\text{T-Sub}\ \frac{\Gamma \vdash k : S \qquad S <: T}{\Gamma \vdash k : T}$$

$$\text{T-Object}\ \frac{}{\texttt{Object OK} \qquad mtype(\texttt{Object}) = nil}$$

$$\text{S-ClassDef}\ \frac{\texttt{class}\ C\ \texttt{extends}\ D\ \texttt{implements}\ I_1, ..., I_n\{...\}}{\begin{array}{c} C <: D \qquad \forall_i\ C <: I_i \\ interfaces(C) = interfaces(D)\ \cup\ \bigcup_i interfaces(I_i) \end{array}}$$

$$\text{S-IntDef}\ \frac{\texttt{interface}\ I\ \texttt{extends}\ I_1, ..., I_n\{...\}}{\begin{array}{c} \forall_i\ I <: I_i \\ interfaces(I) = \{\ I\ \}\ \cup\ \bigcup_i interfaces(I_i) \end{array}}$$

Figure 2: Basic subtyping rules

5 Method typing

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

$$\text{T-Var}\ \frac{}{\Gamma \vdash x : \Gamma(x)}$$

$$\text{T-New}\ \frac{C\ \texttt{OK}}{\Gamma \vdash \texttt{new C()} : C}$$

$$\text{T-Invoke}\ \frac{\Gamma \vdash e : C \qquad C\ \texttt{OK} \qquad T = mtype(C)}{\Gamma \vdash e.m() : T}$$

Figure 3: Expression typing

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

provides at most one appropriate default for `m()`. For each syntactic form (declare method with a default, declare method without a default, don't declare the method), the interface rules are mutually exclusive; at most one will apply to any given class.

Many of the preconditions in these rules deal with proper covariant overriding, which can happen explicitly (where an interface declares a signature for a method when the method also appears in superinterface) or implicitly (where an interface extends multiple interfaces which provide the same method name but not exactly the same return type.) Further, in the presence of covariant overrides, the available default(s) must return a value compatible with the most specific return type.[1]

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 $prune(I_1, ..., I_n)$ 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 overriden, that if multiple interfaces contribute a default, they must be identical to be considered, and that that there must always be a consistent, most-specific return type for methods.

[1] We could choose instead to simply require an interface to declare a default when covariantly overriding a method; in this case we would simply remove rules T-IntDefOvr and T-IntDefInh.

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

$$\text{T-IntSimpleOvr}\ \frac{\begin{array}{c}\texttt{interface}\ I\ \texttt{extends}\ I_1,...,I_n\{\ T\ m()\ \} \\ \forall_i\ I_i\ \ \texttt{OK} \\ \forall_i\ [\ mtype(I_i) = nil\ \vee\ T <: mtype(I_i)\ ] \\ \left|\ \bigcup_{J \in prune(interfaces(I))}\{\ mdef(J)\ \}\ \right| = 0\end{array}}{mtype(I) = T \qquad I\ \ \texttt{OK}}$$

$$\text{T-IntDefOvr}\ \frac{\begin{array}{c}\texttt{interface}\ I\ \texttt{extends}\ I_1,...,I_n\{\ T\ m()\ \} \\ \forall_i\ I_i\ \ \texttt{OK} \\ \forall_i\ [\ mtype(I_i) = nil\ \vee\ T <: mtype(I_i)\ ] \\ S = \bigcup_{J \in prune(interfaces(I))}\{\ mdef(J)\ \} \\ |\ S\ | = 1 \qquad \exists_k k \in S \qquad \Gamma \vdash k : T\end{array}}{mtype(I) = T \qquad I\ \ \texttt{OK}}$$

$$\text{T-IntSimpleInh}\ \frac{\begin{array}{c}\texttt{interface}\ I\ \texttt{extends}\ I_1,...,I_n\{\ \} \\ \forall_i\ I_i\ \ \texttt{OK} \qquad T = lb_{inc}(I_1,...,I_n) \\ \left|\ \bigcup_{J \in prune(interfaces(I))}\{\ mdef(J)\ \}\ \right| = 0\end{array}}{mtype(I) = T \qquad I\ \ \texttt{OK}}$$

$$\text{T-IntDefInh}\ \frac{\begin{array}{c}\texttt{interface}\ I\ \texttt{extends}\ I_1,...,I_n\{\ \} \\ \forall_i\ I_i\ \ \texttt{OK} \qquad T = lb_{inc}(I_1,...,I_n) \\ S = \bigcup_{J \in prune(interfaces(I))}\{\ mdef(J)\ \} \\ |\ S\ | = 1 \qquad \exists_k k \in S \qquad \Gamma \vdash k : T\end{array}}{mtype(I) = T \qquad I\ \ \texttt{OK}}$$

$$\text{T-IntNone}\ \frac{\begin{array}{c}\texttt{interface}\ I\ \texttt{extends}\ I_1,...,I_n\{\ \} \\ \forall_i\ I_i\ \ \texttt{OK} \qquad \forall_i\ mtype(I) = nil\end{array}}{mtype(I) = nil \qquad I\ \ \texttt{OK}}$$

Figure 4: Interface method typing and defaults

The predicate C `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 `OK` and $mres(C) \neq nil$.

$$\text{T-ClassConc}\ \frac{\begin{array}{c}\texttt{class}\ C\ \texttt{extends}\ D\ \texttt{implements}\ I_1,...,I_n\{\ T\ m()\ k\ \} \\ \Gamma \vdash k : T \qquad \forall_i\ I_i\ \texttt{OK} \qquad D\ \texttt{OK} \\ \forall_{U \in \{D,I_1,...,I_n\}}\ [\ mtype(U) = nil\ \vee\ T <: mtype(U)\ ]\end{array}}{mtype(C) = T \qquad C\ \texttt{OK} \qquad mbody(C) = k}$$

$$\text{T-ClassAbs}\ \frac{\begin{array}{c}\texttt{class}\ C\ \texttt{extends}\ D\ \texttt{implements}\ I_1,...,I_n\{\ T\ m()\ \} \\ \forall_i\ I_i\ \texttt{OK} \qquad D\ \texttt{OK} \\ \forall_{U \in \{D,I_1,...,I_n\}}\ [\ mtype(U) = nil \vee T <: mtype(U)\ ]\end{array}}{mtype(C) = T \qquad C\ \texttt{OK} \qquad mbody(C) = nil}$$

$$\text{T-ClassInh}\ \frac{\begin{array}{c}\texttt{class}\ C\ \texttt{extends}\ D\ \texttt{implements}\ I_1,...,I_n\{\ \} \\ \forall_i\ I_i\ \texttt{OK} \qquad D\ \texttt{OK} \qquad T = lb_{inc}(D, I_1, ..., I_n)\end{array}}{mtype(C) = T \qquad C\ \texttt{OK} \qquad mbody(C) = mbody(D)}$$

$$\text{T-ClassNone}\ \frac{\begin{array}{c}\texttt{class}\ C\ \texttt{extends}\ D\ \texttt{implements}\ I_1,...,I_n\{\ \} \\ \forall_i\ I_i\ \texttt{OK} \qquad D\ \texttt{OK} \qquad \forall_i\ mtype(I_i) = nil \qquad mtype(D) = nil\end{array}}{mtype(C) = nil \qquad C\ \texttt{OK}}$$

$$\text{R-Body}\ \frac{\begin{array}{c}\texttt{class}\ C\ \texttt{extends}\ D\ \texttt{implements}\ I_1,...,I_n\{\ ...\ \} \\ C\ \texttt{OK} \qquad mbody(C) \neq nil\end{array}}{mres(C) = mbody(C)}$$

$$\text{R-Defender}\ \frac{\begin{array}{c}\texttt{class}\ C\ \texttt{extends}\ D\ \texttt{implements}\ I_1,...,I_n\{\ ...\ \} \\ C\ \texttt{OK} \qquad mbody(C) = nil \qquad T = mtype(C) \\ S = \bigcup_{I \in prune(interfaces(C))}\{\ mdef(I)\ \} \\ \mid S \mid = 1 \qquad \exists_k k \in S \qquad \Gamma \vdash k : T\end{array}}{mres(C) = k}$$

Figure 5: Class method typing and resolution