Featherweight Defenders

Brian Goetz, Robert Field

April 22, 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, which may be covariantly overridden. Interface methods may simply be abstract definitions, have specified defaults or may explicitly cancel defaults inherited from supertypes. Class methods are abstract or concrete, and concrete methods may be reabstracted.

We believe that this includes the most significant inheritance features that are relevant to resolution of extension methods in Java. There are some additional features, such as bridge methods, that are beyond the scope of this model.

2 Syntax

The metavariables C, and D (and their derivatives) range over class names and the metavariable I and J range over interface names. The metavariables T, R, U, V, and W range over all types (classes and interfaces). The metavariable S ranges over sets of types. The metavariable k ranges over a set of nominal identifiers, typed in the static typing context Γ. The metavariable b ranges over a set of typed method bodies, typed in the static typing context Γ. Figure 1 shows the syntatic forms for FD.

Set operations on sets of types rely on the identity of the types (all classes in FD are nominal). So, for example, combining $\{T\} \cup \{T\}$ simply yields the set $\{T\}$. For set operations, *nil* is treated as the empty set.

$$
\begin{array}{rcl}
T & ::= & \texttt{Object} \mid C \mid I \\
K & ::= & \texttt{class}\ C\ \texttt{extends}\ D\ \texttt{implements}\ I_1, ..., I_n\ \{\ [\ R\ m()\ \langle\ b\ \mid\ \texttt{abstract}\ \rangle\ ]\ \} \\
L & ::= & \texttt{interface}\ I\ \texttt{extends}\ I_1, ..., I_n\ \{\ [\ R\ m()\ [\ \texttt{default}\ \langle\ k \mid \texttt{none}\ \rangle\ ]\ ]\ \}
\end{array}
$$

Figure 1: FD language syntax

3 Preliminaries

Figure 2 shows some general typing judgements needed by FD, and the subtyping judgements for classes and interfaces.

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

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

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

$$\text{S-Class}\ \frac{\texttt{class}\ C\ \texttt{extends}\ D\ \texttt{implements}\ I_1, ..., I_n\{...\}}{C <: D \qquad \forall_i\ C <: I_i}$$

$$\text{S-Intf}\ \frac{\texttt{interface}\ I\ \texttt{extends}\ I_1, ..., I_n\{...\}}{\forall_i\ I <: I_i}$$

Figure 2: Basic typing rules

As in Featherweight Java, we use "lookup functions" (such as $mtype(T)$) and "marker predicates" (such as T `OK`) in the inference rules to record information about types. These will be introduced as they are used by the inference rules.

4 Compile-time vs runtime

One of the goals of this model is to present a formal procedure for resolving references to a method `m()` in a class C among the many candidate choices contributed by superclasses and interfaces. In languages like Java, method resolution is performed both at compile time (to ensure that methods resolve uniquely, and that required methods are implemented, and to reject source files that do not meet these requirements) and at run time (to perform linkage dynamically.) It is important to ensure that resolution decisions made at compile time and run time are consistent, but issues like separate compilation and dynamic linking pose challenges to this goal, since class files compiled separately may not provide a consistent view of the type hierarchy at run time.

This model divides rules into two categories – typing-related (those beginning with T-) and resolution-related (those beginning with R-). A compiler would execute and enforce all the rules; the runtime would execute only the resolution rules. In a well-typed program, the compiler and runtime view of the world would be identical; in an inconsistently-typed program (say, due to separate compilation), it may still be possible to perform method resolution at runtime according to the R- rules.

5 Method typing

Figure 3 shows the typing rules for resolving the type of method `m()` in classes and interfaces.

Figure 3 defines the following lookup functions and marker predicates:

- $mtype(T)$, which indicates the type of `m()` in type T. If `m()` is not a member of T, then $mtype(T)$ will be nil.
- T `SigOK`, which indicates that the type T provides a consistent return type for `m()`. The compiler should reject types for which T `SigOK` does not hold, but T `SigOK` is not sufficient to declare that T is well-formed (For example, T could have conflicting defenders or problems with covariant overrides.)

We define the function lbi_f for computing an *inclusive lower bound* (under subtyping) for a projection under f of a set of types. (The inclusive lower bound for a set S is the lower bound of S if S contains its lower bound, and nil (undefined) otherwise.) We define $lbi_f(T_1, ..., T_n)$ as follows:

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

It is possible that there are multiple values of i that satisfy the conditions for $lbi_f(T_1, ..., T_n)$; in that case, each of the $f(T_i)$ will be subtypes of each other, and therefore equivalent. We use lbi_{mtype} to determine whether a set of types can contribute a consistent return type for the method `m()`.

We also define the following shorthand for optional subtyping on lookup functions:

$$T \tilde{<:} f(U) \quad \equiv \quad f(U) = nil \ \vee \ T <: f(U)$$

$$\text{T-ClassConc}\ \frac{\texttt{class}\ C\ \texttt{extends}\ D\ \texttt{implements}\ I_1, ..., I_n\ \{\ T\ m()\ b\ \} \quad \Gamma \vdash b : T \quad \forall_{U \in \{D, I_1, ..., I_n\}} [\ U\ \ \texttt{SigOK} \wedge T <: mtype(U)\ ]}{mtype(C) = T \qquad C\ \ \texttt{SigOK}}$$

$$\text{T-ClassAbs}\ \frac{\texttt{class}\ C\ \texttt{extends}\ D\ \texttt{implements}\ I_1, ..., I_n\ \{\ T\ m()\ \ \texttt{abstract}\ \} \quad \forall_{U \in \{D, I_1, ..., I_n\}} [\ U\ \ \texttt{SigOK} \wedge T <: mtype(U)\ ]}{mtype(C) = T \qquad C\ \ \texttt{SigOK}}$$

$$\text{T-ClassNone}\ \frac{\texttt{class}\ C\ \texttt{extends}\ D\ \texttt{implements}\ I_1, ..., I_n\ \{\ \} \quad \forall_{U \in \{D, I_1, ..., I_n\}}\ U\ \ \texttt{SigOK} \quad T = lbi_{mtype}(D, I_1, ..., I_n)}{mtype(C) = T \qquad C\ \ \texttt{SigOK}}$$

$$\text{T-IntDef}\ \frac{\texttt{interface}\ I\ \texttt{extends}\ I_1, ..., I_n\ \{\ T\ m()\ \ \texttt{default}\ k\ \} \quad \Gamma \vdash k : T \qquad \forall_i [\ I_i\ \ \texttt{SigOK} \wedge T <: mtype(I_i)\ ]}{mtype(I) = T \qquad I\ \ \texttt{SigOK}}$$

$$\text{T-IntNoDef}\ \frac{\texttt{interface}\ I\ \texttt{extends}\ I_1, ..., I_n\ \{\ T\ m()\ [\ \texttt{default none}\ ]\ \} \quad \forall_i [\ I_i\ \ \texttt{SigOK} \wedge T <: mtype(I_i)\ ]}{mtype(I) = T \qquad I\ \ \texttt{SigOK}}$$

$$\text{T-IntInh}\ \frac{\texttt{interface}\ I\ \texttt{extends}\ I_1, ..., I_n\ \{\ \} \quad \forall_i\ I_i\ \ \texttt{SigOK} \quad T = lbi_{mtype}(I_1, ..., I_n)}{mtype(I) = T \qquad I\ \ \texttt{SigOK}}$$

$$\text{T-IntNone}\ \frac{\texttt{interface}\ I\ \texttt{extends}\ I_1, ..., I_n\ \{\ \} \quad \forall_i [\ I_i\ \ \texttt{SigOK} \wedge mtype(I_i) = nil\ ]}{mtype(I) = nil \qquad I\ \ \texttt{SigOK}}$$

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

Figure 3: Method typing

6 Defender methods

The rules so far apply (mostly) to the existing semantics of Java as well as the Featherweight Defenders extension. We now explore the rules for declaring, inheriting, and pruning defender methods (interface methods with defaults).

A key aspect of inheritance for defender methods is *pruning* of less-specific default-providing interfaces from consideration in the resolution process. In Java, it is allowable for a class or interface to extend an interface both directly and indirectly, as in the following example:

```
interface A { Object m() default k }
interface B extends A { Object m() default l }
class C implements A, B { }
```

Here, `C` implements `A` both directly and indirectly. This idiom is common as a documentation device, but in Java 7 and earlier the additional declaration of `A` has no effect, because it is already implicit in the extension of `B`. 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 `A` 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 I and J each contribute a default method for `m()`, and I is a strict subtype of J, then W is pruned from consideration in contributing a default. The R-IntInh and R-ClassInh rules in figure 4 implement this pruning behavior.

Intuitively, the rules for method resolution behave as follows:

- A method defined in a type takes precedence over methods defined in supertypes.
- Abstract methods in interfaces do not influence resolution.
- A method (concrete or abstract) inherited from a superclass takes precedence over a default inherited from an interface.
- More specific default-providing interfaces take precedence over less specific ones.
- If we are to resolve `m()` to a default method, it must be from the unique most specific default-providing interface.

Figure 4 covers the rules for inheriting, overriding, and reabstracting defender methods in interfaces. It defines the following lookup function:

- $dcand(T)$, which indicates the set of interfaces which could provide a defender for `m()` in T.

$$\text{T-IntBody}\ \frac{\texttt{interface}\ I\ \texttt{extends}\ I_1, ..., I_n\ \{\ T\ m()\ \texttt{default}\ \langle\ k \mid \texttt{none}\ \rangle\ \}\quad I\ \texttt{SigOK}\quad \forall_i\ I_i\ \texttt{OK}}{I\ \texttt{OK}}$$

$$\text{T-IntNoBody}\ \frac{\texttt{interface}\ I\ \texttt{extends}\ I_1, ..., I_n\ \{\ [\ T\ m()\ ]\ \}\quad I\ \texttt{SigOK}\quad \forall_i\ I_i\ \texttt{OK}\quad \mid dcand(I) \mid = 0}{I\ \texttt{OK}}$$

$$\text{T-IntInhBody}\ \frac{\texttt{interface}\ I\ \texttt{extends}\ I_1, ..., I_n\ \{\ [\ T\ m()\ ]\ \}\quad I\ \texttt{SigOK}\quad \forall_i\ I_i\ \texttt{OK}\quad \mid dcand(I) \mid = 1\quad \exists_{V \in dcand(I)}\ mtype(I) = mtype(V)}{I\ \texttt{OK}}$$

$$\text{R-IntDef}\ \frac{\texttt{interface}\ I\ \texttt{extends}\ I_1, ..., I_n\ \{\ T\ m()\ \texttt{default}\ \langle\ k \mid \texttt{none}\ \rangle\ \}}{dcand(I) = \{\ I\ \}}$$

$$\text{R-IntInh}\ \frac{\texttt{interface}\ I\ \texttt{extends}\ I_1, ..., I_n\ \{\ [\ T\ m()\ ]\ \}\quad S = \bigcup_i\ dcand(I_i)}{dcand(I) = \{\ W \in S\ :\ \forall_{V \in S}\ V <: W\ \Rightarrow\ V = W\ \}}$$

$$\text{R-ClassInh}\ \frac{\texttt{class}\ C\ \texttt{extends}\ D\ \texttt{implements}\ I_1, ..., I_n\ \{\ \}\quad S = \bigcup_{U \in \{\ D, I_1, ..., I_n\ \}} dcand(U)}{dcand(C) = \{\ W \in S\ :\ \forall_{V \in S}\ V <: W\ \Rightarrow\ V = W\ \}}$$

Figure 4: Calculating defender candidates

7 Method inheritance in classes

All classes (except the root class `Object`) have a single superclass. We treat a concrete method body and a declaration that the method is abstract in the same way (collectively, we call these the method definition). A class inherits method definitions from its superclass, unless the method is explicitly given a new definition in the subclass (either a new body is provided or the method is explicitly reabstracted). Figure 5 shows the rules for method body inheritance, including the requirement that if a method's return type is covariantly overridden, any method definition from a superclass must be overridden at the same point as the covariant override. Figure 5 defines the following lookup functions and marker predicates:

- $mprov(C)$, which indicates the *provenance* of the (possibly inherited)

$$\text{T-ObjectBodyOK}\ \frac{}{\texttt{Object BodyOK}}$$

$$\text{T-ClassBody}\ \frac{\texttt{class}\ C\ \texttt{extends}\ D\ \texttt{implements}\ I_1, ..., I_n\ \{\ T\ m()\ \langle\ b \mid \texttt{abstract}\ \rangle\ \}}{C\ \texttt{SigOK} \qquad D\ \texttt{BodyOK} \qquad \forall_i\ I_i\ \texttt{OK}}{C\ \texttt{BodyOK}}$$

$$\text{T-ClassNoBody}\ \frac{\texttt{class}\ C\ \texttt{extends}\ D\ \texttt{implements}\ I_1, ..., I_n\ \{\ \}}{C\ \texttt{SigOK} \qquad D\ \texttt{BodyOK} \qquad \forall_i\ I_i\ \texttt{OK} \qquad \neg D\ \texttt{HasDefn}\ \vee\ [\ D\ \texttt{HasDefn}\ \wedge\ mtype(D) = mtype(C)\ ]}{C\ \texttt{BodyOK}}$$

$$\text{R-ClassBody}\ \frac{\texttt{class}\ C\ \texttt{extends}\ D\ \texttt{implements}\ I_1, ..., I_n\ \{\ T\ m()\ \langle\ b \mid \texttt{abstract}\ \rangle\ \}}{mprov(C) = C \qquad C\ \texttt{HasDefn}}$$

$$\text{R-ClassInhBody}\ \frac{\texttt{class}\ C\ \texttt{extends}\ D\ \texttt{implements}\ I_1, ..., I_n\ \{\ \}}{D\ \texttt{HasDefn}}{mprov(C) = mprov(D) \qquad C\ \texttt{HasDefn}}$$

Figure 5: Method body inheritance

method body for `m()` in C. It is the identity of the type providing the body (or abstract declaration) for C.

- C `HasDefn`, which indicates that C or one of its superclasses has either a method body or is explicitly declared abstract. If C `HasDefn` but not C `HasBody`, then (a well-formed) C must be an abstract class.
- C `BodyOK`, which indicates that C is well-formed with respect to a declared or inherited method body (or abstract declaration). A class is deemed to be `BodyOK` if it has a valid body or abstract declaration, it inherits a valid body or abstract declaration (and the return type for `m()` is not covariantly overriden, either explicitly or implicitly), or inherits no body or abstract declaration.

8 Method resolution

We are now able to define the resolution of `m()` in a class C which may inherit its implementation from a superclass or from a defended method in an interface.

Figure 6 defines the following lookup functions and marker predicates:

- $mres(C)$, which indicates the resolution or `m()` in class C. Its value is the identity of the type from which `m()` is inherited. The resolution would be to a concrete or default method.
- C `OK`, which indicates that C does not contain any conflicting definitions for `m()`. (Such as conflicting defaults, bad overrides, conflicts between the return type specified in a body and the return type broadened through covariant overrides, etc.) It does not mean that `m()` has been uniquely resolved within C (either to a concrete body or to a default); it is allowable for classes to be abstract.
- T `HasBody`, which indicates that T declares either a concrete method body or a method default, depending on whether it is a class or interface. (It is not inherited; it is strictly a property of the class declaration.)

$$\text{T-NoDefault}\ \frac{\texttt{class}\ C\ \texttt{extends}\ D\ \texttt{implements}\ I_1, ..., I_n\ \{\ ...\ \} \quad C\ \texttt{BodyOK} \quad D\ \texttt{OK} \quad C\ \texttt{HasDefn} \vee [\ \neg C\ \texttt{HasDefn} \wedge |\ dcand(C)\ | = 0\ ]}{C\ \texttt{OK}}$$

$$\text{T-FromDefault}\ \frac{\texttt{class}\ C\ \texttt{extends}\ D\ \texttt{implements}\ I_1, ..., I_n\ \{\ ...\ \} \quad C\ \texttt{BodyOK} \quad D\ \texttt{OK} \quad \neg C\ \texttt{HasDefn} \quad |\ dcand(C)\ | = 1 \quad \exists_{J \in dcand(C)}\ mtype(J) = mtype(C)}{C\ \texttt{OK}}$$

$$\text{R-HasBody}\ \frac{\texttt{class}\ C\ \texttt{extends}\ D\ \texttt{implements}\ I_1, ..., I_n \{\ T\ m()\ b\ \}}{C\ \texttt{HasBody}}$$

$$\text{R-HasDef}\ \frac{\texttt{interface}\ I\ \texttt{extends}\ I_1, ..., I_n \{\ T\ m()\ \ \texttt{default}\ k\ \}}{I\ \texttt{HasBody}}$$

$$\text{R-ResolveImpl}\ \frac{\texttt{class}\ C\ \texttt{extends}\ D\ \texttt{implements}\ I_1, ..., I_n\ \{\ ...\ \} \quad C\ \texttt{HasDefn} \quad T = mprov(C) \quad T\ \texttt{HasBody}}{mres(C) = T}$$

$$\text{R-ResolveDef}\ \frac{\texttt{class}\ C\ \texttt{extends}\ D\ \texttt{implements}\ I_1, ..., I_n\ \{\ ...\ \} \quad \neg C\ \texttt{HasDefn} \quad |\ dcand(C)\ | = 1 \quad \exists_{J \in dcand(C)} \quad J\ \texttt{HasBody}}{mres(C) = J}$$

Figure 6: Resolution