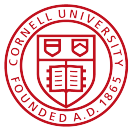
CS 4110 – Programming Languages and Logics
Lecture #29: Featherweight Java



One way to model the features of an object-oriented languages is to encode it using standard type structure. This leads to so-called object encodings. A different (and arguably simpler) way is to model these features directly. This lecture considers a core calculus for Java developed by Igarashi, Pierce, and Wadler called *Featherweight Java*.

Featherweight Java is small by design. It reduces Java to its essential features including classes, inheritance, constructors, fields, methods, and casts, and omits everything else. In particular, the language does not include interfaces, assignment, concurrency, overloading, exceptions, or the `public`, `private`, and `protected` modifiers. Because the language is so simple, its proof of type soundness is short. It is also easy to extend—indeed, in the original paper on Featherweight Java, the authors present an extension with parametric polymorphism (i.e., generics).

1 Featherweight Java

The syntax of Featherweight Java is given by the following grammar.

$$
\begin{array}{rcll}
P & ::= & \overline{CL}\ e & \textit{programs} \\
CL & ::= & \texttt{class}\ C\ \texttt{extends}\ C\ \{\overline{C\ f};\ K\ \overline{M}\} & \textit{classes} \\
K & ::= & C(\overline{C\ f})\ \{\texttt{super}(\overline{f}); \overline{\texttt{this}.f = f;}\} & \textit{constructors} \\
M & ::= & C\ m(\overline{C\ x})\{\texttt{return}\ e\} & \textit{methods} \\
e & ::= & x & \textit{expressions} \\
 & | & e.f & \\
 & | & e.m(\overline{e}) & \\
 & | & \texttt{new}\ C(\overline{e}) & \\
 & | & (C)\ e & \\
v & ::= & \texttt{new}\ C(\overline{v}) & \textit{values} \\
E & ::= & [\cdot] & \textit{evaluation contexts} \\
 & | & E.f & \\
 & | & E.m(\overline{e}) & \\
 & | & v.m(\overline{v}, E, \overline{e}) & \\
 & | & \texttt{new}\ C(\overline{v}, E, \overline{e}) & \\
 & | & (C)\ E &
\end{array}
$$

We will use the notation $\overline{e}$ to denote sequences of the form $e_1, \ldots, e_k$ (and $\overline{C\ f}$ for $C_1\ f_1, \ldots, C_k\ f_k$). By convention, metavariables B, C, and D range over class names, m ranges over method names, and f and g range over field names. As usual, x ranges over variables. Note that the syntax of Featherweight Java is a strict subset of Java. This means that every Featherweight Java program can be executed using a stock Java compiler and virtual machine.

At the top level, programs consist of a list of classes and a distinguished "main" expression. We will use the notation $P(C)$ to denote the definition of the class C in the program. A class has a name, a superclass, a list of fields (instance variables), a constructor, and a list of methods. A constructor takes a list of arguments, invokes the `super(...)` constructor, and then initializes its fields. A method takes a list of arguments and returns a single expression—a variable, field projection, method invocation, constructor call, or cast.

Although this language is simple, we can still write many useful programs (in fact, all useful programs—the language is Turing complete). Here is a simple example that illustrates how we can represent pairs in Featherweight Java:

```
class A extends Object {
  A() { super(); }
}

class B extends Object {
  B() { super(); }
}

class Pair extends Object  {
  Object fst;
  Object snd;

  Pair(Object fst, Object snd) {
    super();
    this.fst = fst;
    this.snd = snd;
  }

  Pair swap Object() {
    return new Pair(this.snd, this.fst);
  }
}
```

Using the small-step operational semantics described later in this lecture, it will be possible to evaluate the expression

```
new Pair(new A(), new B()).swap()
```

to the following:

```
new Pair(new B(), new A())
```

Note that because the language does not include assignment (except in constructors), Featherweight Java programs must be written in a functional style, constructing new objects instead of mutating old ones.

As another example, consider what happens when we evaluate the following expression:

```
(A) new B()
```

Because `B` is not declared to be a subtype of `A`, the cast fails. In the full Java language, the virtual machine would raise an exception. In Featherweight Java, we model this instead as a stuck term.

2 Subtype Relation

The subtype relation is the reflexive and transitive closure of the binary relation between classes and superclasses. Formally it is defined using the following axioms and inference rules:

$$\text{S-REFL}\ \frac{}{C \leq C} \qquad\qquad \text{S-TRANS}\ \frac{C \leq D \quad D \leq E}{C \leq E}$$

$$\text{S-CLASS}\ \frac{P(C) = \texttt{class}\ C\ \texttt{extends}\ D\ \{\overline{C\ f};\ K\ \overline{M}\}}{C \leq D}$$

Note that Featherweight Java subtyping is *nominal*, just like Java—the objects generated by a class are a subtype of the objects generated by its superclass.

3 Auxiliary Functions

Before we present the operational semantics for Featherweight Java, let us define a few auxiliary functions for looking up the methods and fields of classes.

Field Lookup The set of fields defined in a class is simply the list of all fields in the definition of the class in the program, as well as the fields of its superclass.

$$\text{F-OBJECT}\ \frac{}{\mathit{fields}(\texttt{Object}) = []}$$

$$\text{F-CLASS}\ \frac{P(C) = \texttt{class}\ C\ \texttt{extends}\ D\ \{\overline{C\ f};\ K\ \overline{M}\} \qquad \mathit{fields}(D) = \overline{D\ g}}{\mathit{fields}(C) = \overline{D\ g} \mathbin{@} \overline{C\ f}}$$

Method Body Lookup Similarly, to lookup the body of a method we either read it off from the class definition, or take the method body of the superclass. Note that the structure returned includes both the arguments $\overline{x}$ and the body of the method e.

$$\text{MB-CLASS}\ \frac{\begin{array}{c}P(C) = \texttt{class}\ C\ \texttt{extends}\ D\ \{\overline{C\ f};\ K\ \overline{M}\} \\ B\ m\ (\overline{B\ x})\ \{\texttt{return}\ e\} \in \overline{M}\end{array}}{\mathit{mbody}(m, C) = (\overline{x}, e)}$$

$$\text{MB-SUPER}\ \frac{\begin{array}{c}P(C) = \texttt{class}\ C\ \texttt{extends}\ D\ \{\overline{C\ f};\ K\ \overline{M}\} \\ B\ m\ (\overline{B\ x})\ \{\texttt{return}\ e\} \notin \overline{M}\end{array}}{\mathit{mbody}(m, C) = \mathit{mbody}(m, D)}$$

4 Operational Semantics

The operational semantics for Featherweight Java is defined in the usual way, using small-step operational semantics rules and evaluation contexts. It uses a call-by-value evaluation strategy.

$$\text{E-CONTEXT}\ \frac{e \to e'}{E[e] \to E[e']}$$

$$\text{E-PROJ}\ \frac{\mathit{fields}(C) = \overline{C\ f}}{\texttt{new}\ C(\overline{v}).f_i \to v_i}$$

$$\text{E-INVK}\ \frac{\mathit{mbody}(m, C) = (\overline{x}, e)}{\texttt{new}\ C(\overline{v}).m(\overline{u}) \to [\overline{x} \mapsto \overline{u}, \texttt{this} \mapsto \texttt{new}\ C(\overline{v})]e}$$

$$\text{E-CAST}\ \frac{C \leq D}{(D)\ \texttt{new}\ C(\overline{v}) \to \texttt{new}\ C(\overline{v})}$$

Note that the rule for method invocation steps to the body of the method with the actual arguments substituted for the formal parameters and the object that the method is being invoked on, `new` C $(\overline{v})$, substituted for `this`. Also note that in the cast rule, the target type of the cast D must be a supertype of the object `new` $C(\overline{v})$ being cast—i.e., an upcast simply strips off the cast while downcasts and casts between unrelated classes get stuck.

5 Type System

The type system for Featherweight Java has three main pieces (and several auxiliary definitions). The first is the typing relation for expressions, which is a three-place relation $\Gamma \vdash e : C$ between a context Γ that maps variables to their types, an expression e, and a type C.

Method Type Lookup The typing relation for expressions relies on an auxiliary definition that calculates method types. To calculate the type of a method `m` in a class `C` we either look it up from the class definition, or take the type of the method in its superclass.

$$\text{MT-CLASS}\ \frac{\begin{array}{c}P(C) = \texttt{class}\ C\ \texttt{extends}\ D\ \{\overline{C\ f};\ K\ \overline{M}\} \\ B\ m\ (\overline{B\ x})\ \{\texttt{return}\ e\} \in \overline{M}\end{array}}{\mathit{mtype}(m, C) = \overline{B} \to B}$$

$$\text{MT-SUPER}\ \frac{\begin{array}{c}P(C) = \texttt{class}\ C\ \texttt{extends}\ D\ \{\overline{C\ f};\ K\ \overline{M}\} \\ B\ m\ (\overline{B\ x})\ \{\texttt{return}\ e\} \notin \overline{M}\end{array}}{\mathit{mtype}(m, C) = \mathit{mtype}(m, D)}$$

Expression Typing With this auxiliary definition in hand, we are ready to define the typing relation for expressions:

$$\text{T-VAR}\ \frac{\Gamma(x) = C}{\Gamma \vdash x : C}$$

$$\text{T-FIELD}\ \frac{\Gamma \vdash e : C \qquad \mathit{fields}(C) = \overline{C\ f}}{\Gamma \vdash e.f_i : C_i}$$

$$\text{T-INVK}\ \frac{\Gamma \vdash e : C \qquad \mathit{mtype}(m, C) = \overline{B} \to B \qquad \Gamma \vdash \overline{e} : \overline{A} \qquad \overline{A} \leq \overline{B}}{\Gamma \vdash e.m(\overline{e}) : B}$$

$$\text{T-NEW}\ \frac{\mathit{fields}(C) = \overline{C\ f} \qquad \Gamma \vdash \overline{e} : \overline{B} \qquad \overline{B} \leq \overline{C}}{\Gamma \vdash \texttt{new}\ C(\overline{e}) : C}$$

$$\text{T-UCAST}\ \frac{\Gamma \vdash e : D \qquad D \leq C}{\Gamma \vdash (C)\ e : C}$$

$$\text{T-DCAST}\ \frac{\Gamma \vdash e : D \qquad C \leq D \qquad C \neq D}{\Gamma \vdash (C)\ e : C}$$

$$\text{T-SCAST}\ \frac{\Gamma \vdash e : D \qquad C \not\leq D \qquad D \not\leq C \qquad \textit{stupid warning}}{\Gamma \vdash (C)\ e : C}$$

Note that it includes three typing rules for casts—one for upcasts, another for downcasts, and another for "stupid" casts between unrelated types. Stupid casts are not allowed by the standard Java typechecker but are needed to prove preservation.

Method Typing The next definition is a two place relation that checks that a method `m` is "okay" in a class `C`. It uses an auxiliary definition *override* that checks that a method validly overrides any methods with the same name defined by its superclass.

$$\text{OVERRIDE}\ \frac{\mathit{mtype}(m, D) = \overline{A} \to A \textit{ implies } \overline{A} = \overline{B} \text{ and } A = B}{\mathit{override}(m, D, \overline{B} \to B)}$$

$$\text{METHOD-OK}\ \frac{\overline{x : B}, \texttt{this} : C \vdash e : A \qquad A \leq B \qquad P(C) = \texttt{class}\ C\ \texttt{extends}\ D\ \{\overline{C\ f};\ K\ \overline{M}\} \qquad \mathit{override}(m, D, \overline{B} \to B)}{B\ m(\overline{B\ x})\{\texttt{return}\ e\}\ \mathit{OK\ in}\ C}$$

Class Typing The final piece of the type system checks that a class is "okay".

$$\text{CLASS-OK}\ \frac{K = C(\overline{D\ g}, \overline{C\ f})\ \{\texttt{super}(\overline{g}); \overline{\texttt{this}.f = f;}\} \qquad \mathit{fields}(D) = \overline{D\ g} \qquad \overline{M}\ \mathit{OK\ in}\ C}{\texttt{class}\ C\ \texttt{extends}\ D\ \{\overline{C\ f};\ K\ \overline{M}\}\ \mathit{OK}}$$

To typecheck a whole program, we check that every class is okay, and that the main expression is well-typed under the empty context.