Topic 1. The essence of Java: the language FJ.

We shift our attention to an Object Oriented (OO) language model, called Featherweight Java (FJ). FJ is intended to capture the core features of the OO character of Java. FJ as defined here (with minor variations) has been used in the research literature to develop Java generic types. It is so true to Java that while the specification here is purely symbolic, and we do not develop an implementation, any JVM serves as an implementation. In short, FJ is just a restricted featureset of Java. However, the idea is that this small set of features captures essential components of the Java OO model. In particular, the language model captures:

- Class definitions, including constructors, fields, and methods
- Inheritance
- Override and dynamic dispatch
- Object casting and run-time cast checks

Syntax

The syntax of FJ is defined below. We let A, B, C, D range over class names, x range over variables, f range over field names, and m range over method names. Values, denoted v or u, are objects, i.e. expressions of the form new C(v_1,..., v_n). We assume given an Object value that has no fields or methods.

\[
\begin{align*}
L & ::= \text{class } C \text{ extends } C \{ \bar{f} ; K \} \quad \text{class definitions} \\
K & ::= C(\bar{f})\{\text{super}(\bar{f}); \text{this}.\bar{f} = \bar{f}; \} \quad \text{constructors} \\
M & ::= C.m(\bar{x})\{\text{return } e; \} \quad \text{methods} \\
e & ::= x | e.f | e.m(\bar{e}) | \text{new } C(\bar{e}) | (C)e \quad \text{expressions}
\end{align*}
\]

For brevity in this syntax, we use vector notations. Specifically we write \(\bar{f}\) to denote the sequence \(f_1,...,f_n\), similarly for \(\bar{C}, \bar{f}, \bar{x}, \bar{e}\), etc., and we write \(\bar{R}\) as shorthand for \(M_1 \cdots M_n\). We write the empty sequence as \(\emptyset\), we use a comma as a sequence concatenation operator, and we write \(|\bar{x}|\) to denote the length of \(\bar{x}\). If and only if \(m\) is one of the names in \(\bar{m}\), we write \(m \in \bar{m}\). Vector notation is also used to abbreviate sequences of declarations; we let \(\bar{C}\) and \(\bar{f}\); denote \(C_1 f_1, \ldots, C_n f_n\) and \(C_1 f_1; \ldots; C_n f_n;\) respectively. The notation \(\text{this}.\bar{f} = \bar{f}\); abbreviates \(\text{this}.f_1 = f_1; \ldots; \text{this}.f_n = f_n;\). Sequences of names and declarations are assumed to contain no duplicate names. Example:

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

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

    Object meth(Object x) { return x; }
}
```

class Pair extends Object {
    Object fst;
    Object snd;
    Pair(Object fst, Object snd) {
        super(); this.fst = fst; this.snd = snd;
    }
    Pair setfst(Object newfst) {
        return new Pair(newfst, this.snd);
    }
}

Semantics
The semantic definition has several components, in addition to evaluation rules. However, the semantics is ultimately based on substitution, like the semantics of D.

Inheritance/subtyping relation
At the core of the OO model is the idea of inheritance: one class C inherits the behavior of another class D if C is a subclass of D, written $C <: D$. Operationally, this means that all Cs contain the fields and methods contained in D.

$$
\begin{align*}
C <: C & \quad B <: C & \quad C <: D \\
& \Rightarrow B <: D \\
CT(C) = \text{class } C \text{ extends } D \{ \ldots \}
\end{align*}
$$

The class table and field and method body lookup
The class table $CT$ maintains class definitions. The manner in which we look up field and method definitions implements inheritance and override, which allows fields and methods to be redefined in subclasses. A program is a pair $(CT, e)$ of a class table and an expression $e$ to be evaluated.

$$
\begin{align*}
\text{fields}(&\text{Object}) = \emptyset & \quad \text{class } C \text{ extends } D \{ \overline{C} \overline{f}; \overline{K} \overline{R} \} & \quad \text{fields}(&D) = \emptyset, \overline{C} \overline{f} \\
CT(C) = \text{class } C \text{ extends } D \{ \overline{C} \overline{f}; \overline{K} \overline{R} \} & \quad B \text{m}(\overline{B} \overline{x})\{\text{return } e;\} \in \overline{R} \\
& \quad mbody(m, C) = \overline{x}, e \\
CT(C) = \text{class } C \text{ extends } D \{ \overline{C} \overline{f}; \overline{K} \overline{R} \} & \quad m \not\in \overline{R} \\
& \quad mbody(m, C) = mbody(m, D)
\end{align*}
$$

Operational semantics
Now, we can define the operational semantics of FJ. We define the scope of a variable as the method body it is a parameter of, so substitution for free variables is as defined previously. Note the similarity between the $\text{INVOKE}$ rule and the $\text{APPLFIX}$ rule of D. Also, note especially how the treatment of objects as (higher-order!) values allows the semantics to capture dynamic dispatch. When a method is invoked, the version of the method associated with the particular object at hand during evaluation is activated, even if it is “known” by another type statically.

$$
\begin{align*}
\text{NEW} & \quad \forall e_i \in \overline{e}. e_i \Rightarrow v_i & \quad \text{FIELD} & \quad e \Rightarrow \text{new } C(\overline{v}) & \quad \text{fields}(C) = \overline{C} \overline{f} & \quad f_i \in \overline{f} \\
\text{new } C(\overline{e}) \Rightarrow \text{new } C(\overline{v}) & \quad e.f_i \Rightarrow v_i \\
\text{INVOC} & \quad d \Rightarrow \text{new } C(\overline{v}) & \quad \forall e_i \in \overline{e}. e_i \Rightarrow u_i & \quad mbody(m, C) = \overline{x}, e & \quad e[\text{new } C(\overline{v}) / \text{this}] [\overline{u}/\overline{x}] \Rightarrow v \\
& \quad d.m(\overline{e}) \Rightarrow v
\end{align*}
$$
Casting  We still have to give a semantics for casting— that is, changing the static view of an object. Due to inheritance behavior, it is always safe to upcast objects, but it is only safe to downcast if you’re actually looking at an object of the appropriate type at runtime. However, we need the capability to perform downcasts, otherwise our type system is overly restrictive. For example, observe that the fst and snd components are always useless Objects unless they are downcast to a more particular form. Hence, we define a semantics of casting that allows safe casts:

\[
\text{CAST} \quad e \Rightarrow \text{new } C(\overline{v}) \quad \text{C <: D} \\
(D)e \Rightarrow \text{new } C(\overline{v})
\]

and also catches unsafe casts and raises an exception badcast. Note that this is not the same thing as fail; in particular, we do not consider a badcast to be a failure condition, but rather the graceful exiting upon detection of a faulty cast. This is the price we pay for the flexibility to downcast— the cost of a run-time cast check.

\[
\text{BADCAST} \quad e \Rightarrow \text{new } C(\overline{v}) \quad \text{C !: D} \\
(D)e \Rightarrow \text{badcast}
\]

The semantics must also include contextual rules for aborting computation:

\[
\text{NEWRAISEBADCAST} \quad \exists e_i \in \overline{e}. \quad e_i \Rightarrow \text{badcast} \\
\text{new } C(\overline{e}) \Rightarrow \text{badcast}
\]

and so on. Example:

class D extends Object {
    D() { super(); }
    // castexample method assumes given a pair of Bs and an A
    Object castexample(Pair p, A a) {
        return ((B)(p.fst)).meth(a);
    }
}

// evaluates to (new A())
(new D()).castexample(new Pair(new C(), new B()), new A())

// evaluates to badcast
(new D()).castexample(new Pair(new A(), new B()), new A())