Objects, continued

Lecture 22 Thursday, April 18, 2013

1 Middleweight Java

Recall that Java is a class-based object oriented language. Let's consider a Java-like calculus, that captures some of the key aspects of Java programs. The aim of this lecture is, in part, to see how some of the class-based object-oriented language features play out, and also to see what is involved with trying to come up with a simple, yet realistic, model of a real language.

We will use *Middleweight Java*[1] (MJ), an imperative calculus designed to help reason about Java-like programs. (It was introduced a couple of years after a pure calculus called "Featherweight Java"[2].)

In MJ, a program consists of a sequence of class declarations, followed by a sequence of statements $\overline{s}$. (In this caluclus, we sometimes write $\overline{X}$ to indicate a sequence of syntactic object X)

Program	$p ::= cd_1 \ldots cd_m; \overline{s}$
Class declaration	$cd ::= \mathsf{class}\ C\ \mathsf{extends}\ D\ \{\ \overline{fd}\ cnd\ \overline{md}\ \}$

A class declaration gives the name of the class being declared, C, the name of the superclass D, and then provides a sequence of field declarations, a constructor declaration, and a sequence of method declarations. We assume there is a distinguished class name Object, which means that every class declaration can give a superclass. A field declaration is just the name of the field, and the class of the field.

Field declaration	$fd ::= C\ f;$

A constructor declaration takes n arguments, invokes the constructor of the super class, and has a sequence of statements.

Constructor declaration	$cnd ::= C(C_1\ x_1, \ldots, C_n\ x_n)\{\ \mathsf{super}(e_1, \ldots, e_k);\ \overline{s}\ \}$

A method declaration takes n arguments, and has a sequence of statements. The return type is either void or a class name.

Method declaration	$md ::= \tau\ m(C_1\ x_1, \ldots, C_n\ x_n)\{\ \overline{s}\ \}$
Return type	$\tau ::= C \mid \mathsf{void}$

Expressions are either the use of a variable x, the special constant value null, a field access $e.f$, a cast $(C)e$, a method call $e.m(e_1, \ldots, e_k)$, or the creation of a new object $\mathsf{new}\ C(e_1, \ldots, e_n)$. Note that some of the expressions (method calls and object creation) are "promotable" to statements.

Expression	$e ::= x \mid \mathsf{null} \mid e.f \mid (C)e \mid pe$
Promotable expression	$pe ::= e.m(e_1, \ldots, e_k) \mid \mathsf{new}\ C(e_1, \ldots, e_n)$
Statement	$s ::= ; \mid pe; \mid \mathsf{if}\ (e_1 == e_2)\ s_1\ \mathsf{else}\ s_2 \mid e_1.f = e_2;$
	$\mid C\ x; \mid x = e; \mid \mathsf{return}\ e; \mid \{\ \overline{s}\ \}$

[1] G. M. Bierman, M. J. Parkinson, and A. M. Pitts. MJ: An imperative core calculus for Java and Java with effects. Technical Report UCAM-CL-TR-563, Cambridge University, 2003.

[2] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. ACM Transactions on Programming Languages and Systems, 23(3):396450, 2001.

Statements are mostly self-explanatory, although note that ";" is like a skip command: it is a no-op.

So we now have syntax for our language. Here are a couple of class declarations that follow the syntax.

```
class Cell extends Object {
    Object contents;
    Cell (Object start){
        super();
        this.contents = start;
    }
    void set(Object update){
        this.contents = update;
    }
}

class Recell extends Cell {
    Object undo;
    Recell (Object start){
       super(start);
       this.undo = null;
    }
    void set(Object update){
       this.undo = this.contents;
       this.contents = update;
    }
}
```

1.1 Operational semantics

This language, even though it greatly simplifies Java, has quite a complicated semantics. It has nested variable scoping, imperative update, inheritance, and method overriding. It will take us a bit of time to explore the operational semantics for this language, and also to examine the static judgments that are required to ensure well-typedness.

Let's define a small step operational semantics for the language. A configuration (H, VS, CF, FS) for MJ has 4 components:

- Heap H, which will map object ids to objects.
- Variable stack VS, which will be used to manage the scope of local variables. It is a stack of method scopes, where each method scope is a stack of block scopes, where a block scope is a context that maps local variables to pairs of values and types.
- Closed frame CF, which is the current statement or expression to evaluate.
- Frame stack FS, which indicates the context in which the closed frame is evaluated. One can think of the frame stack as being the rest of the program to execute, as well as expressing the call stack.

This calculus does not use evaluation contexts, but instead has the configuration explicitly separate the current statement or expression to evaluate from the context. This leads to some simplification of inference rules, and proof, since one doesn't need to reason about evaluation contexts directly.

This is the syntax for values, variable stacks, closed grames, and frame stacks. We use [] for an empty stack and write $X \circ XS$ to indicate a stack with top element X, and the rest of the stack is XS. We use $[\cdot]$ to

indicate a hole in an open frame, similar to a hole in an evaluation context.

Values	$v ::= \mathsf{null} \mid o$
Objects	$o ::= (C, \mathbb{F})$
	$\mathbb{F}$ is a partial function from field names to values
Frame stack	$FS ::= F \circ FS \mid []$
Frame	$F ::= CF \mid OF$
Closed frame	$CF ::= \overline{s} \mid \mathsf{return}\ e; \mid \{\ \} \mid e \mid \mathsf{super}(\overline{e})$
Open frame	$OF ::= \mathsf{if}\ ([\cdot] == e)\ s_1\ \mathsf{else}\ s_2 \mid \mathsf{if}\ (v == [\cdot])\ s_1\ \mathsf{else}\ s_2 \mid [\cdot].f \mid [\cdot].f = e; \mid v.f = [\cdot]; \mid (X)[\cdot]$ $\mid x = [\cdot]; \mid \mathsf{return}\ [\cdot]; \mid [\cdot].m(\overline{e}) \mid v.m(v_1, \ldots, v_{i-1}, [\cdot], e_{i+1}, \ldots, e_n)$ $\mid \mathsf{new}\ C(v_1, \ldots, v_{i-1}, [\cdot], e_{i+1}, \ldots, e_n) \mid \mathsf{super}(v_1, \ldots, v_{i-1}, [\cdot], e_{i+1}, \ldots, e_n)$
Variable stack	$VS ::= MS \circ VS \mid []$
Method scope	$MS ::= BS \circ MS \mid []$
Block scope	BS is a partial map from variables to (C, v) pairs

Variables stacks are quite complicated: they are stacks of stacks of contexts. Why do we need this complicated structure? First, we need a stack of method scopes so that when we enter a new method (via a method call), we can push a new method scope of the stack, and when we return from the method we can pop that method scope. Note that we have a return command, so we may return from anywhere in a method body, not just at the end. Within a method, we may have nested blocks, and the scope of a local variable is the block it is declared in, and any nested blocks. We use block scopes to manage the scope of local variables within a method.

We define two utility functions for manipulating block scopes: $eval(MS, x)$ looks up the value of variable x in method scope MS, and $update(MS, x \mapsto v)$ updates the value of variable x in MS.

$$eval(BS \circ MS, x) \triangleq \begin{cases} BS(x) & \text{if } x \in \text{dom}(BS) \\ eval(MS, x) & \text{otherwise} \end{cases}$$

$$update(BS \circ MS, x \mapsto v) \triangleq \begin{cases} BS[x \mapsto (v, C)] & \text{if } BS(x) = (v', C) \\ BS \circ update(MS, x \mapsto v) & \text{otherwise} \end{cases}$$

To define the semantics, we'll need some additional utility functions to help compute information about classes, for example, to find the declaration for method m of class C, or to figure out what the super class of class C is. To figure this information out, we need to look at the class declarations at the beginning of a program. For simplicity, we will assume that there is a fixed program p, that thus defines the set of class declarations $cd_1, \ldots, cd_m$.

Function $superOf(C)$ returns the immediate super class of class C.

$$superof(C) = D \text{ where } cd_i = \mathsf{class}\ C\ \mathsf{extends}\ D\ \{\ \overline{fd}\ cnd\ \overline{md}\ \}.$$

Function $cnd(C)$ returns the constructor declaration of class C.

$$cnd(C) = cnd \text{ where } cd_i = \mathsf{class}\ C\ \mathsf{extends}\ D\ \{\ \overline{fd}\ cnd\ \overline{md}\ \}.$$

Function $mdecl(C, m)$ returns the method declaration of method M in class C. Note that if class C does not define a method m, then we look in the super class of C for the method declaration. In this way, we achieve inheritance of methods, and allow subclasses to override methods.

$$mdecl(C, m) = \begin{cases} md_j & \text{where } cd_i = \mathsf{class}\ C\ \mathsf{extends}\ D\ \{\ \overline{fd}\ cnd\ \overline{md}\ \} \\ & \quad \text{and } md_j = \tau\ m(C_1\ x_1, \ldots, C_n\ x_n)\{\ \overline{s}\ \} \\ mdecl(D, m) & \text{otherwise, where } superOf(C) = D \end{cases}$$

With these utility functions in hand, we are now ready to define the operational semantics.

$$\textsc{VarRead}\ \frac{eval(MS, x) = v}{(H, MS \circ VS, x, FS) \longrightarrow (H, MS \circ VS, v, FS)}$$

$$\textsc{VarWrite}\ \frac{eval(MS, x) = v'}{(H, MS \circ VS, x = v;, FS) \longrightarrow (H, update(MS, x \mapsto v) \circ VS, ;, FS)}$$

$$\textsc{VarIntro}\ \frac{x \notin \text{dom}(BS \circ MS)}{(H, (BS \circ MS) \circ VS, C\ x;, FS) \longrightarrow (H, (BS[x \mapsto (\mathsf{null}, C)] \circ MS) \circ VS, ;, FS)}$$

$$\textsc{BlockIntro}\ \frac{}{(H, MS \circ VS, \{\ \overline{s}\ \}, FS) \longrightarrow (H, (\varnothing \circ MS) \circ VS, \overline{s}, (\{\ \}) \circ FS)}$$

$$\textsc{BlockElim}\ \frac{}{(H, (BS \circ MS) \circ VS, \{\ \}, FS) \longrightarrow (H, MS \circ VS, ;, FS)}$$

$$\textsc{Return}\ \frac{}{(H, MS \circ VS, \mathsf{return}\ v;, FS) \longrightarrow (H, VS, v, FS)}$$

$$\textsc{Skip}\ \frac{}{(H, VS, ;, F \circ FS) \longrightarrow (H, VS, F, FS)} \qquad \textsc{Sub}\ \frac{}{(H, VS, v, F \circ FS) \longrightarrow (H, VS, F[v], FS)}$$

$$\textsc{If1}\ \frac{}{(H, VS, \mathsf{if}\ (v == v)\ s_1\ \mathsf{else}\ s_2, FS) \longrightarrow (H, VS, s_1, FS)}$$

$$\textsc{If2}\ \frac{}{(H, VS, \mathsf{if}\ (v == v')\ s_1\ \mathsf{else}\ s_2, FS) \longrightarrow (H, VS, s_1, FS)}\ v \neq v'$$

$$\textsc{FieldAccess}\ \frac{H(o) = (C, \mathbb{F}) \quad \mathbb{F}(f) = v}{(H, VS, o.f, FS) \longrightarrow (H, VS, v, FS)}\ f \text{ is a field of } C$$

$$\textsc{FieldWrite}\ \frac{H(o) = (C, \mathbb{F}) \quad \mathbb{F}' = \mathbb{F}[f \mapsto v]}{(H, VS, o.f = v, FS) \longrightarrow (H[o \mapsto (C, \mathbb{F}')], VS, ;, FS)}\ f \text{ is a field of } C$$

$$\textsc{Cast}\ \frac{H(o) = (C', \mathbb{F}) \quad C' \text{ is a subclass of } C}{(H, VS, (C)o, FS) \longrightarrow (H, VS, o, FS)} \qquad \textsc{CastNull}\ \frac{}{(H, VS, (C)\mathsf{null}, FS) \longrightarrow (H, VS, \mathsf{null}, FS)}$$

$$\textsc{New}\ \frac{\begin{array}{c} cnd(C) = C(C_1\ x_1, \ldots, C_n\ x_n)\{\ \mathsf{super}(e_1, \ldots, e_k);\ \overline{s}\ \} \\ \mathbb{F} = \{f \mapsto \mathsf{null} \mid f \text{ a field of } C\} \qquad BS = [\mathsf{this} \mapsto (o, C), x_i \mapsto (v_i, C_i)] \end{array}}{(H, VS, \mathsf{new}\ C(\overline{v}), FS) \longrightarrow (H[o \mapsto (C, \mathbb{F})], (BS \circ []) \circ VS, \mathsf{super}(e_1, \ldots, e_k); \overline{s}, (\mathsf{return}\ o;) \circ FS)}\ o \notin \text{dom}(H)$$

$$\textsc{Super}\ \frac{\begin{array}{c} superOf(C) = C' \\ cnd(C') = C'(C_1\ x_1, \ldots, C_n\ x_n)\{\ \mathsf{super}(e_1, \ldots, e_k);\ \overline{s}\ \} \qquad BS = [\mathsf{this} \mapsto MS(\mathsf{this}), x_i \mapsto (v_i, C_i)] \end{array}}{(H, MS \circ VS, \mathsf{super}(\overline{v}), FS) \longrightarrow (H, (BS \circ []) \circ MS \circ VS, \mathsf{super}(e_1, \ldots, e_k); \overline{s}, (\mathsf{return}\ o;) \circ FS)}$$

$$\textsc{Method}\ \frac{\begin{array}{c} H(o) = (C, \mathbb{F}) \qquad mdecl(C, m) = C'\ m(C_1\ x_1, \ldots, C_n\ x_n)\{\ \overline{s}\ \} \\ BS = [\mathsf{this} \mapsto (o, C), x_i \mapsto (v_i, C_i)] \end{array}}{(H, VS, o.m(\overline{v}), FS) \longrightarrow (H, (BS \circ []) \circ VS, \overline{s}, FS)}$$

$$\textsc{MethodVoid}\ \frac{H(o) = (C, \mathbb{F}) \qquad mdecl(C, m) = \mathsf{void}\ m(C_1\ x_1, \ldots, C_n\ x_n)\{\ \overline{s}\ \} \qquad BS = [\mathsf{this} \mapsto (o, C), x_i \mapsto (v_i, C_i)]}{(H,\ VS, o.m(\overline{v}), FS) \longrightarrow (H, (BS \circ []) \circ VS, \overline{s}, (\mathsf{return}\ o;) \circ FS)}$$

This calculus is larger than any we have seen so far in class. It is large enough, and complex enough, that it is impractical to show an example execution in class. If you'd like to see example executions of this language, see the original MJ paper (Footnote 1).

However, complicated as this calculus is, it is both much simpler than an implementation, as it glosses over many details, such as the representation of frame stacks, variable stacks, etc., and it is much much simpler than the real Java programming language. What Java language features and ideas does this calculus not capture? What are some of the object-oriented concepts does this calculus capture? What concepts does it not express?

1.2 Static semantics

Although the operational semantics described above has classes, and objects are associated with classes at runtime, it is untyped. Indeed, there are many ways that a program in this language could get stuck, or could encounter undefined behavior. Here are some of the ways.

- In a field expression $e.f$, field f may not be a field of the object that e can evaluate to.
- In a method call expression $e.m(\overline{e})$, method m may not be defined in the class of the object that e can evaluate to (or a superclass thereof).
- If the class declarations at the start of the program give more than one definition for a class, then $cnd(C)$, $mdecl(C, m)$, and $superOf(C)$ may not be functions.
- For class C and method m, there should be at most one declaration of a method with name m.
- For class C and field f, there should be at most one declaration of a field with name f.
- There should be no cycles in the superclass relation.
- If we use a class C in a program, we must have a definition of that class (with the exception of the distinguished class Object).
- For a method call $e.m(\overline{e})$ or field access $e.f$, what if the target e evaluates to null?

Some of these potential problems can be dealt with by a type system similar to type systems we have seen previously. That is, we can track the type of local variables and expressions, and ensure that when we update a local variable, the type of the expression can be assigned to the type of the local variable, and similarly for method calls, field access, etc.

Let's think about defining a type system that ensures values are used correctly. We first need to define the subtyping relation $\prec$ (which, in this calculus, is conflated with the subclassing relation). Intuitively, if we have C is a subclass of D (written $C \prec D$), then we will allow an object of class C to be used where an object of class D is expected.

$$\frac{superOf(C) = D}{C \prec D} \qquad \frac{}{C \prec C} \qquad \frac{C_1 \prec C_2 \quad C_2 \prec C_3}{C_1 \prec C_3}$$

The typing judgment for expressions and statements is of the form $\Delta; \Gamma \vdash e : \tau$, where Γ maps local variables to types (i.e., classes), and Δ is a *class table*, i.e., Δ describes type information for classes, such as the names and types of fields, the signatures for methods, etc. A class table is derived from the class declarations of a program (and we would need to define what it means for a class table to be compatible with a sequence of class declarations).

Some example typing rules (not exhaustive):

$$\frac{\Gamma(x) = C}{\Delta; \Gamma \vdash x : C} \qquad \frac{\Gamma(x) = C \qquad C' \prec C \qquad \Delta; \Gamma \vdash e : C'}{\Delta; \Gamma \vdash x = e : \mathsf{void}} \qquad \frac{}{\Delta; \Gamma \vdash \mathsf{null} : C} \qquad \frac{\Delta; \Gamma \vdash e : C \quad \Delta(C)(f) = C'}{\Delta; \Gamma \vdash e.f : C'}$$

$$\frac{\Delta; \Gamma \vdash e : C \qquad \forall i.\ \Delta; \Gamma \vdash e_i : C_i \qquad \Delta(C)(m) = C'_1 \times \cdots \times C'_n \to \tau \qquad \forall i.\ C_i \prec C'_i}{\Delta; \Gamma \vdash e.m(\overline{e}) : \tau}$$

A standard type system doesn't take care of all of the potential issues above. For example, the typing judgment used above can not ensure that a class doesn't declare multiple fields with the same name, or ensure that there are no cycles in the superclass relation. We can however define new judgments to ensure that class declarations are well formed, and put in appropriate checks into those judgments. We omit the presentation of these judgments, but think about how you would go about defining them. Take a look at the MJ paper for details.

Even with these judgments, there are still some issues with that could arise at runtime. What if the target of a method call or field access evaluates to null? What about if we cast object o to class C, but the class of o is not a subclass of C? It is possible for us to design a type system that soundly enforces that expressions evaluate to non-null values, and that all casts succeed. Such a type system would statically reject a program unless it can prove that all targets of method calls and field accesses are non-null, and that for cast $(C)e$, the statically known class of an expression e is a subclass of C.

However, there is a balance to strike between permissiveness of the language and type system, and guarantees about run-time behavior. it is potentially too onerous and restrictive to require the programmer to prove that all casts will succeed, and no field access will occur on a null value. The Java programming language chose to allow exceptions at runtime, instead of getting stuck in these situations. Conceptually, this behavior is similar to the runtime error propagation we saw in Lecture 19. In the calculus we add rules that go directly to an error state, instead of propagating it up through evaluation contexts.

$$\frac{}{(H\,VS, \mathsf{null}.f, FS) \longrightarrow \mathbf{NullPointerError}} \qquad \frac{}{(H\,VS, \mathsf{null}.f = v, FS) \longrightarrow \mathbf{NullPointerError}}$$

$$\frac{}{(H\,VS, \mathsf{null}.m(\overline{v}), FS) \longrightarrow \mathbf{NullPointerError}} \qquad \frac{H(o) = (C', \mathbb{F}) \qquad C' \not\prec C}{(H\,VS, (C)o, FS) \longrightarrow \mathbf{ClassCastError}}$$

1.3 Subclassing is not subtyping

In the last lecture, we noted that inheritance is not the same as subtyping, but in some languages (including Java), these notions are conflated. Let's see in a little more detail what that means.[3]

Suppose we have a class that represents Cartesian points. (And let's suppose we have a class that represents real numbers, and a class representing booleans.)

```
class CartPoint extends Object {
    Real x;
    Real y;
    CartPoint(Real px, Real py){
        super();
        this.x = px;
        this.y = py;
    }
    void moveRight(Real dx) {
```

[3]This example is derived from the paper "Inheritance is Not Subtyping" by Cook, Hill, and Canning, in *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp125–135, 1990.

```
        this.x = this.x.add(dx);
    }
    Boolean equals(CartPoint that){
        return this.x.equals(that.x)
                  .and(this.y.equals(that.y));
    }
}
```

Suppose that we want to define a colored point, that is, a Cartesian point that has a color. We'll subclass `CartPoint`, since much of the functionality that we want for the colored point is similar to `CartPoint`.

```
class ColPoint extends CartPoint {
    Color c;
    CartPoint(Real px, Real py, Color pc){
        super(ps, py);
        this.c = pc;
    }
    Boolean equals(ColPoint that){
        return super.equals(that)
                  .and(this.c.equals(that.c));
    }
}
```

(We have used a construct here that isn't actually in MJ: we write `super.equals(that)` to indicate the invocation of the superclass' `equals` method. This is correct Java code, but MJ does not model this aspect of Java.)

Class `ColPoint` inherits from class `CartPoint`. This means `ColPoint` inherits the method `moveRight`, so we could write `new ColPoint(zero, zero, blue).moveRight(seven)` (assuming that `zero`, `blue`, and `seven` refer to appropriate objects).

Note that both classes have a method `equals` which takes as an argument an object of the same class, and returns a boolean.

Is `ColPoint` a subtype of `CartPoint`?

No, it is not, due to the covariance of argument type to method `equals`. Try and write a program that shows that `ColPoint` is not a subtype of `CartPoint`. That is, write a program that tries to use an object of class `ColPoint` where an object of class `CartPoint` is expected, and the program gets stuck.

What does Java do to prevent this unsoundness?