# CS415 — Discussion Section Notes 7

Claire Le Goues

March 20, 2008

## Operational Semantics

So far, we've seen two types of judgments:

---

$$(1)\quad \frac{\begin{array}{c} O \vdash e_0 : T \\ T \leq T_0 \\ O[T_0/x] \vdash e_1 : T_1 \end{array}}{O \vdash \text{let } x : T_0 \leftarrow e_0 \text{ in } e_1 : T_1} \qquad (2)\quad \frac{W \vdash T : type}{W \vdash P\langle T \rangle : type}$$

---

We've been using these rules to prove properties of programs, e.g. "this piece of code has type $T_1$" or just "this type is valid."

Guiding question: how did these individual rules help us type check entire programs?

Ok. Now we're interested in saying things about the *meaning* of programs. So instead of ": SomeTypeName" we'll say stuff like "$5 + 7 : 12, S$." We've already ruled out syntax and type errors, so we can assume any program we see is 'legal.'

Another guiding question: how do the new rules help us? Why all this theory, anyway?

Let's check out some opsem judgments:

---

$$\frac{\begin{array}{c} s \text{ is a string literal} \\ n \text{ is the length of } s \end{array}}{so, E, S \vdash s : String(n, s), S} \qquad \frac{}{so, E, S \vdash self : so, S}$$

---

Flashback: functional programming avoids using assignment, and we try to code without side-effects. But, there are side-effects in Cool. How do we handle them in these operational semantics rules?

Or, what's wrong with the following rules:

---

$$\frac{so, E, S \vdash e : v, S \quad S_1 = S[v/E(id)]}{so, E, S \vdash id \leftarrow e : v, S_1} \qquad \frac{so, E, S \vdash e_1 : Bool(false), S}{so, E, S \vdash \text{while } e_1 \text{ loop } e_2 \text{ pool } : void, S}$$

---

REMEMBER SIDE EFFECTS

Consider function calls—when do we evaluate the parameters?

---

$$\frac{\begin{array}{c} so = X(\ldots) \\ T_0 = \begin{cases} X(\ldots) & \text{if } T = SELF\_TYPE \\ T & \text{otherwise} \end{cases} \\ class(T_0) = (a_1 : T_1 \leftarrow e_1, \ldots, a_n : T_n \leftarrow e_n) \\ l_i = newloc(S) \quad \text{for } i = 1, \ldots, n \\ v = T_0(a_1 = l_1, \ldots, a_n = l_n) \\ E' = \emptyset[a_1 : l_1, \ldots, a_n : l_n] \\ S_1 = S[D_{T_1}/l_1, \ldots, D_{T_n}/l_n] \\ v, E', S_1 \vdash \{a_1 \leftarrow e_1; \ldots; a_n \leftarrow e_n;\} : v_n, S_2 \end{array}}{so, E, S \vdash \text{new } T : v, S_2}$$

---

This defines exactly what needs to happen on a `new` call. What is the output for the following piece of code?

Listing 1: init order for COOL

```
class A {                        |
 e3 : String <-                  | class Main inherits IO {
  { e1 <- "Bye\n"; e2; };        |  main() : Object {
 e1 : String;                    |   let a : A <- new A in {
 e2 : String <- e1;              |    out_string(a.gete3());
                                 |    out_string(a.gete2());
 gete2() : String { e2 };        |   }
 gete3() : String { e3 };        |  };
};                               | };
```

Let's do method dispatch as well (while we're here).

---

$$
\frac{
\begin{array}{r@{}l}
so, E, S \vdash e_1 : v_1, S_1 \\
so, E, S_1 \vdash e_2 : v_2, S_2 \\
\vdots \\
so, E, S_{n-1} \vdash e_n : v_n, S_n \\
so, E, S_n \vdash e_0 : v_0, S_{n+1} \\
v_0 = X(a_1 = l_1, \ldots, a_m = l_m) \\
imp(X, f) = (x_1, \ldots, x_n, e_{body}) \\
l_{xi} = newloc(S_{n+1}) \quad \text{for } i = 1, \ldots, n \\
E' = \emptyset[x_1 : l_{x1}, \ldots, x_n : l_{xn}, a_1 : l_1, \ldots, a_m : l_m] \\
S_{n+2} = S_{n+1}[v_1/l_{x1}, \ldots, v_n/l_{xn}] \\
v_0, E', S_{n+2} \vdash e_{body} : v, S_{n+3}
\end{array}
}{
so, E, S \vdash e_0.f(e_1, \ldots, e_n) : v, S_{n+3}
}
$$

---

The bajillion dollar question: how does this turn into code?