Operational Semantics

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

1. \[ O \vdash e_0 : T \]
   \[ T \leq T_0 \]
   \[ \frac{O[T_0/x] \vdash e_1 : T_1}{O \vdash \text{let } x : T_0 \leftarrow e_0 \text{ in } e_1 : T_1} \]

2. \[ W \vdash T : \text{type} \]
   \[ \frac{W \vdash P(T) : \text{type}}{W \vdash P(T)} \]

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:

1. \( s \) is a string literal
2. \( n \) is the length of \( s \)

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:

\[
\begin{align*}
so, E, S & \vdash e : v, S \\
S_1 & = S[v/E(id)] \\
so, E, S & \vdash id \leftarrow e : v, S_1 \\
so, E, S & \vdash e_1 : \text{Bool}(false), S \\
so, E, S & \vdash \text{while } e_1 \text{ loop } e_2 \text{ pool : void}, S
\end{align*}
\]

REMEMBER SIDE EFFECTS
Consider function calls—when do we evaluate the parameters?

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

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

```plaintext
class A {
    e3 : String <- "Bye\n"; e2; }
    e1 : String;
    e2 : String <- e1;
    gete2() : String { e2; }
    gete3() : String { e3; }
}

class Main inherits IO {
    main() : Object {
        let a : A <- new A in {
            out_string(a.gete3());
            out_string(a.gete2());
        }
    }
}
```
Let’s do method dispatch as well (while we’re here).

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

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