Formal Type Systems, OOP, and Project 4

Formal Type Systems

Recall the typing rules for function abstraction and application from lecture:

$$\frac{\Gamma, v : T_1 \vdash t_2 \ : \ T_2}{\Gamma \vdash (\textbf{lambda}\ v : T_1 .\, t_2) \ : \ T_1 \to T_2} \qquad \frac{\Gamma \vdash t_1 \ : \ T_2 \to T_3 \qquad \Gamma \vdash t_2 \ : \ T_2}{\Gamma \vdash (t_1\ t_2) \ : \ T_3}$$

Also recall the typing rule for the **let** binding construct:

$$\frac{\Gamma \vdash t_1 \ : \ T_1 \qquad \Gamma, v : T_1 \vdash t_2 \ : \ T_2}{\Gamma \vdash (\textbf{let}\ v = t_1\ \textbf{in}\ t_2) \ : \ T_2}$$

Consider the following program fragment

(**let** *f* = (**lambda** *x*: *Int*. (*x* <= 10)) **in** (*f* 3))

Derive the type of this expression in any type environment

Project 4

Consider the following code samples from the uC language in Project 4. Consult the uC spec and the Project 4 spec to determine the appropriate errors output by your semantic analyzer (if any) as well as which phase the errors are caught in.

<table>
<tr><th>uC code</th><th>Error(s) Reported</th><th>Phase</th></tr>
<tr><td><pre>int foo(int a)(boolean a) {
    return;
}</pre></td><td></td><td></td></tr>
<tr><td><pre>void main(string[] args)(int c,
                         int b)
{
    c = b = 7;
    5 = 7;
}</pre></td><td></td><td></td></tr>
<tr><td><pre>void int_to_long()() {}</pre></td><td></td><td></td></tr>
<tr><td><pre>struct foo(boolean b);

void main(string[] args)() {
  new foo(null);
  new foo(true, false);
  new foo { 3 };
}</pre></td><td></td><td></td></tr>
<tr><td><pre>void main(string[] args)(foo f)
{
  f = new foo(args[0]);
  bar(f);
}

void bar(foo f)() {
  println(f.s);
}

struct foo(string s);</pre></td><td></td><td></td></tr>
</table>

In our Python code, every uC type of expression is represented as a subclass of `ExpressionNode`. Furthermore, each expression in uC is made up of subexpressions. How might Python's `super()` method be useful when performing type checking on expressions?