COMP 317: Semantics of Programming Languages

Lecture 5 Summary

This lecture continues the inductive definition of [[*P*]](*S*). The syntax of programs is defined by

```
<Program> ::= skip |
              <Variable> := <Expression>; |
              <Program> <Program> |
              if (<BooleanExpression>) {<Program>} else {<Program>} |
              while (<BooleanExpression>) {<Program>}
```

Lecture 4 covered assignments, so four cases remain to complete the inductive definition. This lecture concentrates on `skip`, sequential composition, and conditionals; while-loops will be covered in Lecture 6.

Each of the four remaining cases is straightforward, but it is worth spending some extra time on them in order to get used to working with formal notation. The advantage of using formal notation in the definition of the denotational semantics of expressions is that we can calculate the semantics of expressions in a mechanical way - effectively giving us an interpreter for expressions. We want the same thing for programs.

Using informal descriptions instead of formal notation, recall that our general description of [[*P*]](*S*) was "the state that results from running program *P* in state *S*". We want to translate this into a formal notation that consists of variables, functions, function applications, etc.

In the case that *P* is `skip`, we want to say that `skip` does nothing: the resulting state is the same as the starting state - but this is just *S* itself. So our translation is just:

for all States *S*, [[`skip`]](*S*) = *S* .

And this describes the semantics of `skip` formally.

For sequential composition, a program of the form *P1 P2* is executed by doing *P1* and then doing *P2*. In other words, to run *P1 P2* in a state *S*, we run *P2* in the state that results from running *P1* in state *S*. Now "the state that results from running *P1* in state *S*" is translated to [[*P1*]](*S*), and to "run *P2* in" that state means applying [[*P2*]] to that state, so we can translate the whole thing to

for all States *S* and Programs *P1* and *P2*, [[*P1 P2*]](*S*) = [[*P2*]]([[*P1*]](*S*)) .

For conditionals, to execute `if(`*B*`){`*P1*`}else{`*P2*`}` *in a state S, we first test B (translation: [[B]](S)); if that evaluates to true, then we execute P1* *(translation: [[P1]](S)), otherwise we execute P2 (translation: [[P2]](S)). Putting all that together, we have:*

for all States S and Programs $P1$ and $P2$ and Boolean Expressions B,
[[`if (`B`) {`$P1$`} else {`$P2$`}`]](S) = [[$P1$]](S)   if [[B]](S)
[[`if (`B`) {`$P1$`} else {`$P2$`}`]](S) = [[$P2$]](S)   if not [[B]](S)

Key Points

Giving inductive definitions in formal notation allows us to calculate in a mechanical way. Expressing ideas in formal notation involves translating those ideas; that act of translation can be carried out step-by-step (partial translations such as [[P2]](S) will form a part of the whole translation).

You should be able to

- *translate all the boxed definitions above into (informal) words;*
- *calculate results such as [[`'x := 2; 'y := 'x + 'y;`]](S) for any state S.*

Grant Malcolm