Statements and Operational Semantics

Stefan Muller, based on material by Jim Sasaki

CS 536: Science of Programming, Fall 2023
Lecture 5

1 A Simple Programming Language, continued

Today, we'll talk about the syntax and semantics of another part of the language, *statements*. Below, we'll use e for expressions. When we know an expression is going to be a Boolean, we might use B instead (but the syntax of these expressions is still the same).

Statements	e	::=	$x := e$	Assignment
		|	$a[e] := e$	Array Assignment
		|	if B then S else S fi	Conditional
		|	while B do S od	Loop
		|	$S; S$	Sequence
		|	skip	Skip

A few notes on the syntax:

- We can sequence statements together using semicolons: e.g. $S_1; S_2$. We can continue this with multiple statements—while it rarely matters semantically, we'll say that sequencing is right-associative, i.e. $S_1; S_2; S_3 \equiv S_1; (S_2; S_3)$.
- This makes a sequence of statements look almost like in C/C++/Java, where a semicolon ends a statement. Technically though, we shouldn't have a semicolon at the end (e.g., $S_1; S_2;$ is not syntactically valid).
- skip is a no-op statement. Why is this useful? Maybe you don't want an "else" branch of a conditional:

$$\text{if } x < 0 \text{ then } x := x + \overline{1} \text{ else skip fi}$$

- $x := e$ evaluates e and assigns the value to variable x. $a[e_1] := e_2$ evaluates e_2 and assigns the value to the e_1^{th} element of array a (a can also be multi-dimensional; recall the syntactic sugar from last class).
- If and while statements work like in most major programming languages.
- As with expressions, we've omitted a lot in the name of keeping the language simple, but there's a lot of room for syntactic sugar. For example, want for loops? We can represent `for (x = 0; x < n; x++) { S }` as:

$$x := \overline{0}; \text{while } x < n \text{ do } S; x := x + \overline{1} \text{ od}$$

Examples.

- The following program computes the Factorial of n (assuming n is positive): at the end of the program, $n \geq 0 \rightarrow r = n!$.

$r := \overline{1};$
while $n \geq 1$
do
  $r := r * n;$
  $n := n - 1$
od

- Remember, we don't have assignment expressions (like in C, where `y = x++` assigns `x + 1` to `x` and returns the old `x` which is assigned to `y`. However, we can express this as

$$y := x; x := x + \overline{1}$$

2 (Small-step) Operational Semantics of Programs

Last time, we defined the *semantics* of expressions (i.e., what expressions mean or how they evaluate) with $\sigma(e)$. Evaluating statements is a bit more complicated. We'll evaluate them using a "small-step operational semantics". To unpack that a bit:

- *Semantics* are what a program/expression/statement/etc. means.
- *Operational semantics* describes how a program executes.
- *Small-step operational semantics* describes a program executing one step at a time, like how a computer does. Next time, we'll contrast this with *big-step operational semantics*.

We'll actually define the semantics not just over statements, but over "configurations", which are a pair of a statement and a state, which we'll write as $\langle S, \sigma \rangle$. This is because we need the state to evaluate the statement, and also evaluating the statement might change the state. We'll write

$$\langle S_1, \sigma_1 \rangle \to \langle S_2, \sigma_2 \rangle$$

to mean that in one step, if we're in state σ_1 and we execute S_1, the program changes to S_2 (this is a bit confusing; it'll become clearer in a bit) and the state changes to σ_2.

As an example,

$$\langle x := \overline{1}; y := \overline{2}, \{x = 0, y = 0\} \rangle \to \langle y := \overline{2}, \{x = 1, y = 0\} \rangle$$

(technically, this will actually be two steps with the rules we'll give, but this is just to give an idea).

We can keep going:

$$\langle x := \overline{1}; y := \overline{2}, \{x = 0, y = 0\} \rangle \to \langle y := \overline{2}, \{x = 1, y = 0\} \rangle \to \langle \mathsf{skip}, \{x = 1, y = 2\} \rangle$$

How do we know when we're done? When we reach $\langle \mathsf{skip}, \sigma \rangle$ for some σ, which is the final result of the program.

We specify the operational semantics using *inference rules*. These are written with a horizontal line, and a statement (called a *conclusion*) about a configuration stepping to another configuration below the line. Above the line are zero or more *premises* that say when we can apply the rule. If all the premises are true (determining whether they're true may require applying more rules), then we can apply the rule to determine that the conclusion is true.

$$\frac{}{\langle x := e, \sigma \rangle \to \langle \mathsf{skip}, \sigma[x \mapsto \sigma(e)] \rangle} \qquad \frac{0 \leq \sigma(e_1) < |\sigma(a)|}{\langle a[e_1] := e_2, \sigma \rangle \to \langle \mathsf{skip}, \sigma[a[\sigma(e_1)] \mapsto \sigma(e_2)] \rangle}$$

$$\frac{\langle S_1, \sigma \rangle \to \langle S_1', \sigma \rangle}{\langle S_1; S_2, \sigma \rangle \to \langle S_1'; S_2, \sigma \rangle} \qquad \frac{}{\langle \mathsf{skip}; S, \sigma \rangle \to \langle S, \sigma \rangle} \qquad \frac{\sigma(e) = T}{\langle \mathsf{if}\ e\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi}, \sigma \rangle \to \langle S_1, \sigma \rangle}$$

$$\frac{\sigma(e) = F}{\langle \mathsf{if}\ e\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi}, \sigma \rangle \to \langle S_2, \sigma \rangle} \qquad \frac{}{\langle \mathsf{while}\ e\ \mathsf{do}\ S\ \mathsf{od}, \sigma \rangle \to \langle \mathsf{if}\ e\ \mathsf{then}\ S; \mathsf{while}\ e\ \mathsf{do}\ S\ \mathsf{od}\ \mathsf{else}\ \mathsf{skip}\ \mathsf{fi}, \sigma \rangle}$$

Example 1 Let's formally apply the rules to the example above.

$$\begin{array}{ll} & \langle x := \overline{1}; y := \overline{2}, \{x = 0, y = 0\}\rangle \\ \to & \langle \mathsf{skip}; y := \overline{2}, \{x = 0, y = 0\}[x \mapsto 1]\rangle \\ \to & \langle y := \overline{2}, \{x = 1, y = 0\}\rangle \\ \to & \langle \mathsf{skip}, \{x = 1, y = 2\}\rangle \end{array}$$

Note that we write $\{x = 0, y = 0\}[x \mapsto 1]$ for clarity. This *doesn't* step to $\{x = 1, y = 0\}$, it is *equal to* $\{x = 1, y = 0\}$. We could just as easily have written it this way instead.

As we see, the first assignment actually steps to skip and then we use another rule (and another step) to remove that skip. This will get a little tiresome, so we can also write

$$\begin{array}{ll} & \langle x := \overline{1}; y := \overline{2}, \{x = 0, y = 0\}\rangle \\ \to^2 & \langle y := \overline{2}, \{x = 1, y = 0\}\rangle \\ \to & \langle \mathsf{skip}, \{x = 1, y = 2\}\rangle \end{array}$$

where $\to^2$ means that this is actually 2 steps. (More generally, we'll use $\to^n$ to mean n steps and $\to^*$ to mean any number of steps, including 0.) We can use this to skip over "boring" steps. How do we know if a step is "boring"? There's no hard and fast rule, but if a step doesn't update the state, check a condition or do something else of interest like that, it's probably safe to skip. When in doubt, write out the steps.

Example 2 Below, let $W = \mathsf{while}\ x \geq \overline{0}\ \mathsf{do}\ x := x - \overline{1}\ \mathsf{od}$.

$$\begin{array}{ll} & \langle W, \{x = 1\}\rangle \\ \to & \langle \mathsf{if}\ x \geq \overline{0}\ \mathsf{then}\ x := x - \overline{1}; W\ \mathsf{else\ skip\ fi}, \{x = 1\}\rangle \\ \to & \langle x := x - \overline{1}; W, \{x = 1\}\rangle \\ \to^2 & \langle W, \{x = 1\}[x \mapsto 0]\rangle \\ \to & \langle \mathsf{if}\ x \geq \overline{0}\ \mathsf{then}\ x := x - \overline{1}; W\ \mathsf{else\ skip\ fi}, \{x = 0\}\rangle \\ \to & \langle x := x - \overline{1}; W, \{x = 0\}\rangle \\ \to^2 & \langle W, \{x = 0\}[x \mapsto -1]\rangle \\ \to & \langle \mathsf{if}\ x \geq \overline{0}\ \mathsf{then}\ x := x - \overline{1}; W\ \mathsf{else\ skip\ fi}, \{x = -1\}\rangle \\ \to & \langle \mathsf{skip}, \{x = -1\}\rangle \end{array}$$