Scribe: Piti Irawan, Milind Kulkarni Lecturer: Radu Rugina

# 1 Commands in Small Step Semantics - Continued

Definition of **while** command in small step operational semantics:

$$\frac{\langle b,\ \sigma\rangle \to_1 b'}{\langle \textbf{while}\ b\ \textbf{do}\ c,\ \sigma\rangle \to_1 \langle \textbf{while}\ b'\ \textbf{do}\ c,\ \sigma\rangle}$$

$$\frac{}{\langle \textbf{while false do}\ c,\ \sigma\rangle \to_1 \langle \textbf{skip},\ \sigma\rangle}$$

$$\frac{}{\langle \textbf{while true do}\ c\ \sigma\rangle \to_1\ ?}$$

But this will not work, as we have essentially evaluated $b$ for all iterations of the **while** loop (and furthermore, we have lost track of $b$ once we evaluate it to **true** in the last step), although we would actually like to evaluate $b$ separately for each iteration of the loop. Thus, we must consider an alternate method for expressing the operational semantics of a **while** loop:

$$\frac{}{\langle \textbf{while}\ b\ \textbf{do}\ c,\ \sigma\rangle \to_1 \langle \textbf{if}\ b\ \textbf{then}\ (c; \textbf{while}\ b\ \textbf{do}\ c)\ \textbf{else skip},\ \sigma\rangle}$$

This recursive definition of **while** allows us to evaluate $b$ separately for each iteration of the loop.

# 2 Arithmetic Expressions in Small Step Semantics

To do arithmetic expressions (i.e. expressions of the form $a_0\ op\ a_1$ we use three rules:

$$\frac{\langle a_0,\ \sigma\rangle \to_1\ a_0'}{\langle a_0\ op\ a_1,\ \sigma\rangle \to_1\ a_0'\ op\ a_1}$$

$$\frac{\langle a_1,\ \sigma\rangle \to_1\ a_1'}{\langle n\ op\ a_1,\ \sigma\rangle \to_1\ n\ op\ a_1'}$$

$$\frac{}{\langle n_0\ op\ n_1,\ \sigma\rangle \to_1\ n}\ n = n_0\ op\ n_1$$

By way of example, let us consider the expression $(1+2)+(3+4)$.

$$\frac{\overline{\langle (1+2),\ \sigma\rangle \to_1 3}}{\langle (1+2)+(3+4),\ \sigma\rangle \to_1 3+(3+4)} \Rightarrow \frac{\overline{\langle (3+4),\ \sigma\rangle \to_1 7}}{\langle (3+(3+4)),\ \sigma\rangle \to_1 3+7} \Rightarrow \frac{}{\langle 3+7,\ \sigma\rangle \to_1 10}$$

Note: Unlike in large step semantics, each rule has at most one premise. Thus, proofs for each small step are constructed as "chains" instead of the trees found in large step semantics. The individual small steps can then be combined to construct the execution sequence for the whole program.

For example, consider the following simple program, and the resulting proofs using both large step and small step semantics:

```
if x < y
   then x := x + 1
   else skip
```

$\sigma(x) = 1$ and $\sigma(y) = 2$

The proof tree for this in large step semantics is as follows:

$$\dfrac{\dfrac{\dfrac{}{\langle x,\ \sigma\rangle \to 1} \quad \dfrac{}{\langle y,\ \sigma\rangle \to 2}}{\langle x < y,\ \sigma\rangle \to \textbf{true}} \quad \dfrac{\dfrac{\dfrac{}{\langle x,\ \sigma\rangle \to 1} \quad \dfrac{}{\langle 1,\ \sigma\rangle \to 1}}{\langle x+1,\ \sigma\rangle \to 2}}{\langle x := x+1,\ \sigma\rangle \to \sigma[x \mapsto 2]}}{\langle \textbf{if}\, x < y\, \textbf{then}\, x := x+1\, \textbf{else}\, \textbf{skip},\ \sigma\rangle \to \sigma[x \mapsto 2]}$$

The following shows the execution sequence according to the small step semantics. The proof chain for each small step is not shown here.

$$\begin{aligned}
&\langle \textbf{if}\, x < y\, \textbf{then}\, x := x+1\, \textbf{else}\, \textbf{skip},\ \sigma\rangle\\
&\to_1 \langle \textbf{if}\, 1 < \text{y}\, \textbf{then}\, x := x+1\, \textbf{else}\, \textbf{skip},\ \sigma\rangle\\
&\to_1 \langle \textbf{if}\, 1 < 2\, \textbf{then}\, x := x+1\, \textbf{else}\, \textbf{skip},\ \sigma\rangle\\
&\to_1 \langle \textbf{if}\, \textbf{true}\, \textbf{then}\, x := x+1\, \textbf{else}\, \textbf{skip},\ \sigma\rangle\\
&\to_1 \langle x := x+1,\ \sigma\rangle\\
&\to_1 \langle x := 1+1,\ \sigma\rangle\\
&\to_1 \langle x := 2,\ \sigma\rangle\\
&\to_1 \langle \textbf{skip},\ \sigma[x \mapsto 2]\rangle
\end{aligned}$$

# 3 Program Properties with Small Step Semantics

Now we introduce the *transitive closure* for $\to_1$. The symbol $\to_1^*$ represents zero or more small steps:

$$\langle c,\ \sigma\rangle \to_1^* \langle c', \sigma'\rangle \Leftrightarrow \langle c,\ \sigma\rangle \to_1 \langle c_1,\ \sigma_1\rangle \to_1 \ldots \to_1 \langle c',\ \sigma'\rangle$$

With the definition of $\to_1^*$ small step semantics can now express program properties just as large step semantics.

## 3.1 Termination

$$\forall c, \sigma,\ \exists \sigma' : \langle c,\ \sigma\rangle \to_1^* \langle \textbf{skip},\ \sigma'\rangle$$

## 3.2 Determinism

$$\forall c, \sigma, \sigma', \sigma'' \langle c,\ \sigma\rangle \to_1^* \langle \textbf{skip},\ \sigma'\rangle \wedge \langle c,\ \sigma\rangle \to_1^* \langle \textbf{skip},\ \sigma''\rangle \Rightarrow \sigma' = \sigma''$$

## 3.3 Program Equivalence

$$\forall c_1, c_2,\ c_1 \sim c_2 \text{ iff } \forall \sigma, \sigma'\ \langle c_1,\ \sigma\rangle \to_1^* \langle \textbf{skip},\ \sigma'\rangle \Leftrightarrow \langle c_2,\ \sigma\rangle \to_1^* \langle \textbf{skip},\ \sigma'\rangle$$

# 4 Equivalence of Semantics

Large step semantics and small step semantics are, in fact, equivalent. Formally:

$$\forall c, \sigma, \sigma',\ \langle c,\ \sigma\rangle \to \sigma' \Leftrightarrow \langle c,\ \sigma\rangle \to_1^* \langle \textbf{skip},\ \sigma'\rangle$$

Let us look at the following example:

```
while true do skip
```

We have already shown, using large step semantics, that this program will not terminate. We can show the same thing using small step semantics.

$$\begin{array}{l}\langle \textbf{while true do skip},\ \sigma\rangle \\ \rightarrow_1 \langle \textbf{if true then (skip; while true do skip) else skip},\ \sigma\rangle \\ \rightarrow_1 \langle \textbf{skip; while true do skip},\ \sigma\rangle \\ \rightarrow_1 \langle \textbf{while true do skip},\ \sigma\rangle \\ \rightarrow_1 \ldots\end{array}$$

Because the final step is equivalent to the first step, the small-step execution sequence will loop, and thus the program does not terminate.

# 5 Issues in Operational Semantics

## 5.1 Errors

$$\frac{}{\langle n_0/n_1,\ \sigma\rangle \rightarrow_1 n}\ n = n_0/n_1, n_1 \neq 0$$

With the introduction of the division operator, we can get divide-by-zero errors:

$$\langle (1+2)/(1-1),\ \sigma\rangle \rightarrow_1 \ldots \rightarrow_1 \langle (3/0),\ \sigma\rangle$$

However, as soon as an error occurs in small step semantics, it is immediately visible, thus allowing us to quickly distinguish between presence of errors and non-termination. This is unlike large step semantics, where errors must be propagated throughout a command to explicitly determine whether an error occurred.

## 5.2 Order of Operations

Unlike in large step semantics, there is no ambiguity in order of operations. Left to right evaluation is encapsulated in the rules for Arithmetic Expressions in small step semantics.

## 5.3 Parallelism

Small step semantics provides a simple way to express the fine-grain interleaved execution of a parallel command, because it is able capture the small step transitions during program execution.

$$\frac{\langle c_0,\ \sigma\rangle \rightarrow_1 \langle c_0',\ \sigma'\rangle}{\langle c_0|c_1,\ \sigma\rangle \rightarrow_1 \langle c_0'|c_1,\ \sigma'\rangle} \quad \frac{\langle c_1,\ \sigma\rangle \rightarrow_1 \langle c_1',\ \sigma'\rangle}{\langle c_0|c_1,\ \sigma\rangle \rightarrow_1 \langle c_0|c_1',\ \sigma'\rangle}$$

Recall that large-step semantics cannot model the interleaved execution of commands, hence it cannot model parallelism.

## 5.4 Non-determinism

There are two ways to execute a non-deterministic statement such as $(c_0 \Box c_1)$:

$$\frac{}{\langle c_0 \Box c_1,\ \sigma\rangle \rightarrow_1 \langle c_0,\ \sigma\rangle} \quad \frac{}{\langle c_0 \Box c_1,\ \sigma\rangle \rightarrow_1 \langle c_1,\ \sigma\rangle}$$

Note: if one of the two commands does not terminate, the entire non-deterministic construct might not terminate, even though there could exist a choice that does terminate.

# 6 Conclusion

Small step semantics provide all of the capabilities of large step semantics, with some additional benefits (command interleaving to express parallelism, easy order of operations evaluation, etc). So why use large step semantics?

- Easier to construct proofs
- Structural induction vs. induction on length of execution sequences