Lecturer: Dexter Kozen

1 The IMP Language

Today we present a very simple imperative language, IMP, along with small-step and big-step rules for evaluation. We will give

- the IMP language syntax;
- a small-step semantics for IMP;
- a big-step semantics for IMP;
- some notes on why both can be useful.

1.1 Syntax

There are three types of statements in IMP:

- arithmetic expressions *AExp* (elements are denoted $a, a_0, a_1, \ldots$)
- Boolean expressions *BExp* (elements are denoted $b, b_0, b_1, \ldots$)
- commands *Com* (elements are denoted $c, c_0, c_1, \ldots$)

A program in the IMP language is a command in *Com*.

Let *Var* be a countable set of variables. Elements of *Var* are denoted $x, x_0, x_1 \ldots$. Let $n, n_0, n_1, \ldots$ denote integers (elements of $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$). Let $\overline{n}$ be an integer constant symbol representing the number n. The BNF grammar for IMP is

$$
\begin{array}{rcl}
\mathit{AExp} & ::= & \overline{n} \;\mid\; x \;\mid\; (a_0 \,\oplus\, a_1) \\
\mathit{BExp} & ::= & \textbf{true} \;\mid\; \textbf{false} \;\mid\; (a_0 \,\odot\, a_1) \;\mid\; (b_0 \,\oslash\, b_1) \;\mid\; (\neg b) \\
\mathit{Com} & ::= & \textbf{skip} \;\mid\; x := a \;\mid\; (c_0\,;\,c_1) \;\mid\; (\textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2) \;\mid\; (\textbf{while}\ b\ \textbf{do}\ c\ ) \\
\oplus & ::= & + \;\mid\; * \;\mid\; - \\
\odot & ::= & \leq \;\mid\; = \\
\oslash & ::= & \vee \;\mid\; \wedge
\end{array}
$$

1.2 Stores and Configurations

A *store* (also known as a *state*) is a function $Var \to \mathbb{Z}$ that assigns an integer to each variable. The set of all stores is denoted Σ.

A *configuration* is a pair $\langle c, \sigma \rangle$, where $c \in \mathit{Com}$ is a command and σ is a store. Intuitively, the configuration $\langle c, \sigma \rangle$ represents an instantaneous snapshot of reality during a computation, in which σ represents the current values of the variables and c represents the next command to be executed.

2 Structural Operational Semantics (SOS): Small-Step Semantics

Small-step semantics specifies the operation of a program one step at a time. There is a set of rules that we continue to apply to configurations until reaching a final configuration $\langle \textbf{skip}, \sigma \rangle$ (if ever). We write $\langle c, \sigma \rangle \to \langle c', \sigma' \rangle$ to indicate that the configuration $\langle c, \sigma \rangle$ reduces to $\langle c', \sigma' \rangle$ in one step, and we write $\langle c, \sigma \rangle \xrightarrow{*} \langle c', \sigma' \rangle$ to indicate that $\langle c, \sigma \rangle$ reduces to $\langle c', \sigma' \rangle$ in zero or more steps. Thus $\langle c, \sigma \rangle \xrightarrow{*} \langle c', \sigma' \rangle$ iff there is a $k \geq 0$ and configurations $\langle c_0, \sigma_0\rangle, \ldots, \langle c_k, \sigma_k\rangle$ such that $\langle c, \sigma\rangle = \langle c_0, \sigma_0\rangle$, $\langle c', \sigma'\rangle = \langle c_k, \sigma_k\rangle$, and $\langle c_i, \sigma_i\rangle \rightarrow \langle c_{i+1}, \sigma_{i+1}\rangle$ for $0 \leq i \leq k-1$.

To be completely proper, we will define auxiliary small-step operators $\rightarrow_a$ and $\rightarrow_b$ for arithmetic and Boolean expressions, respectively, as well as $\rightarrow$ for commands[1]. The types of these operators are

$$\begin{aligned} \rightarrow &\;:\; (\mathit{Com} \times \Sigma) \rightarrow (\mathit{Com} \times \Sigma) \\ \rightarrow_a &\;:\; (\mathit{AExp} \times \Sigma) \rightarrow \mathbb{Z} \\ \rightarrow_b &\;:\; (\mathit{BExp} \times \Sigma) \rightarrow \mathbf{2} \end{aligned}$$

Here **2** represents the two-element Boolean algebra consisting of the two truth values $\{\mathit{true}, \mathit{false}\}$ with the usual Boolean operations $\wedge, \vee, \neg$. Intuitively, $\langle a, \sigma\rangle \xrightarrow{*}_a n$ if the expression a evaluates to the integer value n in state σ.

2.1 Arithmetic and Boolean Expressions

- Constants: $\dfrac{}{\langle \overline{n}, \sigma\rangle \rightarrow_a n}$
- Variables: $\dfrac{}{\langle x, \sigma\rangle \rightarrow_a \sigma(x)}$
- Operations: $\dfrac{\langle a_0, \sigma\rangle \rightarrow_a n_0 \quad \langle a_1, \sigma\rangle \rightarrow_a n_1}{\langle a_0 \oplus a_1, \sigma\rangle \rightarrow_a n_0 \oplus n_1}$

The rules for evaluating Boolean expressions and comparison operators are similar.

One subtle point: in the rule for arithmetic operations $\oplus$, the $\oplus$ appearing in the expression $a_0 \oplus a_1$ represents the operation symbol in the IMP language, which is a syntactic object; whereas the $\oplus$ appearing in the expression $n_0 \oplus n_1$ represents the actual operation in $\mathbb{Z}$, which is a semantic object. These are two different things, just as $\overline{n}$ and n are two different things and **true** and *true* are two different things. In this case, at the risk of confusion, we have used the same metanotation $\oplus$ for both of them.

2.2 Commands

Let $\sigma[n/x]$ denote the store that is identical to σ except possibly for the value of x, which is n. That is,

$$\sigma[n/x](y) \;\triangleq\; \begin{cases} \sigma(y), & \text{if } y \neq x, \\ n, & \text{if } y = x. \end{cases}$$

- Assignments: $\dfrac{\langle a, \sigma\rangle \rightarrow_a n}{\langle x := a, \sigma\rangle \rightarrow \langle \mathbf{skip}, \sigma[n/x]\rangle}$
- Sequences: $\dfrac{\langle c_0, \sigma\rangle \rightarrow \langle c_0', \sigma'\rangle}{\langle c_0; c_1, \sigma\rangle \rightarrow \langle c_0'; c_1, \sigma'\rangle} \qquad \dfrac{}{\langle \mathbf{skip}; c_1, \sigma\rangle \rightarrow \langle c_1, \sigma\rangle}$
- Conditionals: $\dfrac{\langle b, \sigma\rangle \rightarrow_b \mathit{true}}{\langle \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, \sigma\rangle \rightarrow \langle c_0, \sigma\rangle} \qquad \dfrac{\langle b, \sigma\rangle \rightarrow_b \mathit{false}}{\langle \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, \sigma\rangle \rightarrow \langle c_1, \sigma\rangle}$
- While statements: $\dfrac{}{\langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma\rangle \rightarrow \langle \mathbf{if}\ b\ \mathbf{then}\ (c; \mathbf{while}\ b\ \mathbf{do}\ c)\ \mathbf{else\ skip}, \sigma\rangle}$

There is no rule for **skip**, since $\langle \mathbf{skip}, \sigma\rangle$ is a final configuration.

[1]Winskel uses $\rightarrow_1$ instead of $\rightarrow$ to emphasize that only a single step is performed.

3 Structural Operational Semantics: Big-Step Semantics

As an alternative to small-step operational semantics, which specifies the operation of the program one step at a time, we now consider big-step operational semantics, in which we specify the entire transition from a configuration (an ⟨expression, state⟩ pair) to a final value. This relation is denoted $\Downarrow$. For arithmetic expressions, the final value is an integer; for Boolean expressions, it is a Boolean truth value *true* or *false*; and for commands, it is a final state. We write

$\langle c, \sigma \rangle \Downarrow \sigma'$	(σ' is the store of the final configuration $\langle \mathbf{skip}, \sigma' \rangle$, starting in configuration $\langle c, \sigma \rangle$)
$\langle a, \sigma \rangle \Downarrow n$	(n is the integer value of arithmetic expression a evaluated in state σ)
$\langle b, \sigma \rangle \Downarrow t$	($t \in \{true, false\}$ is the truth value of Boolean expression b evaluated in state σ)

The big-step rules for arithmetic and Boolean expressions are the same as the small-step rules. However, the rules for commands are different:

- Skip: $\overline{\langle \mathbf{skip}, \sigma \rangle \Downarrow \sigma}$
- Assignments: $\dfrac{\langle a, \sigma \rangle \Downarrow n}{\langle x := a, \sigma \rangle \Downarrow \sigma[n/x]}$
- Sequences: $\dfrac{\langle c_0, \sigma \rangle \Downarrow \sigma' \quad \langle c_1, \sigma' \rangle \Downarrow \sigma''}{\langle c_0; c_1, \sigma \rangle \Downarrow \sigma''}$
- Conditionals: $\dfrac{\langle b, \sigma \rangle \Downarrow \mathit{true} \quad \langle c_0, \sigma \rangle \Downarrow \sigma'}{\langle \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, \sigma \rangle \Downarrow \sigma'} \quad \dfrac{\langle b, \sigma \rangle \Downarrow \mathit{false} \quad \langle c_1, \sigma \rangle \Downarrow \sigma'}{\langle \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, \sigma \rangle \Downarrow \sigma'}$
- While statements: $\dfrac{\langle b, \sigma \rangle \Downarrow \mathit{false}}{\langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma \rangle \Downarrow \sigma} \quad \dfrac{\langle b, \sigma \rangle \Downarrow \mathit{true} \quad \langle c, \sigma \rangle \Downarrow \sigma' \quad \langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma' \rangle \Downarrow \sigma''}{\langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma \rangle \Downarrow \sigma''}$

4 Comparison of Big-Step vs. Small-Step SOS

4.1 Small-Step

- Small-step semantics can model more complex features, like programs that run forever and concurrency.
- Although one-step-at-a-time evaluation is useful for proving certain properties, in many cases it is unnecessary extra work.

4.2 Big-Step

- Big steps in reasoning make it easier to prove things.
- Big-step semantics more closely models an actual recursive interpreter.
- Because evaluation skips over intermediate steps, all programs without final configurations (infinite loops, errors, stuck configurations) look the same.