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 \( n \) be an integer constant symbol representing the number \( n \). The BNF grammar for IMP is

\[
\begin{align*}
AExp & ::= \pi \mid x \mid (a_0 \oplus a_1) \\
BExp & ::= \text{true} \mid \text{false} \mid (a_0 \odot a_1) \mid (b_0 \oslash b_1) \mid \lnot b \\
Com & ::= \text{skip} \mid x := a \mid (c_0 ; c_1) \mid (\text{if } b \text{ then } c_1 \text{ else } c_2) \mid (\text{while } b \text{ do } c) \\
\oplus & ::= + \mid * \mid - \\
\odot & ::= \leq \mid = \\
\oslash & ::= \lor \mid \land
\end{align*}
\]

1.2 Stores and Configurations

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

A \textit{configuration} is a pair \( \langle c, \sigma \rangle \), where \( c \in Com \) is a command and \( \sigma \) is a store. Intuitively, the configuration \( \langle c, \sigma \rangle \) represents an instantaneous snapshot of reality during a computation, in which \( \sigma \) 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 \( \langle \text{skip}, \sigma \rangle \) (if ever). We write \( \langle c, \sigma \rangle \rightarrow \langle c', \sigma' \rangle \) to indicate that the configuration \( \langle c, \sigma \rangle \) reduces to \( \langle c', \sigma' \rangle \) in one step, and we write \( \langle c, \sigma \rangle \overset{*}{\rightarrow} \langle c', \sigma' \rangle \) to indicate that \( \langle c, \sigma \rangle \) reduces to \( \langle c', \sigma' \rangle \) in zero or more steps. Thus \( \langle c, \sigma \rangle \overset{*}{\rightarrow} \langle c', \sigma' \rangle \) 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

\[
\begin{align*}
\rightarrow &: (Com \times \Sigma) \rightarrow (Com \times \Sigma) \\
\rightarrow_a &: (AExp \times \Sigma) \rightarrow \mathbb{Z} \\
\rightarrow_b &: (BExp \times \Sigma) \rightarrow 2
\end{align*}
\]

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

### 2.1 Arithmetic and Boolean Expressions

- **Constants:**
  \[
  \langle \pi, \sigma \rangle \rightarrow_a n
  \]

- **Variables:**
  \[
  \langle x, \sigma \rangle \rightarrow_a \sigma(x)
  \]

- **Operations:**
  \[
  \langle a_0, \sigma \rangle \rightarrow_a n_0 \\
  \langle a_1, \sigma \rangle \rightarrow_a n_1 \\
  \langle a_0 \oplus a_1, \sigma \rangle \rightarrow_a n_0 \oplus n_1
  \]

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 $n$ 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 $\sigma$ 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:**
  \[
  \langle x := a, \sigma \rangle \rightarrow \langle \text{skip}, \sigma[n/x] \rangle
  \]

- **Sequences:**
  \[
  \langle c_0; c_1, \sigma \rangle \rightarrow \langle c_0', c_1', \sigma' \rangle \\
  \langle \text{skip}; c_1, \sigma \rangle \rightarrow \langle c_1, \sigma \rangle
  \]

- **Conditionals:**
  \[
  \langle b, \sigma \rangle \rightarrow_b \text{true} \quad \langle b, \sigma \rangle \rightarrow_b \text{false}
  \]

- **While statements:**
  \[
  \langle \text{while } b \text{ do } c, \sigma \rangle \rightarrow \langle \text{if } b \text{ then } (c; \text{while } b \text{ do } c) \text{ else skip}, \sigma \rangle
  \]

There is no rule for $\text{skip}$, since $\langle \text{skip}, \sigma \rangle$ 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 \(<\text{expression}, \text{state}\>) \text{ 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 \text{true} or \text{false}; and for commands, it is a final state. We write

\[
\begin{align*}
\langle c, \sigma \rangle & \Downarrow \sigma' & (\sigma' \text{ is the store of the final configuration } \langle \text{skip}, \sigma' \rangle, \text{ starting in configuration } \langle c, \sigma \rangle) \\
\langle a, \sigma \rangle & \Downarrow n & (n \text{ is the integer value of arithmetic expression } a \text{ evaluated in state } \sigma) \\
\langle b, \sigma \rangle & \Downarrow t & (t \in \{\text{true}, \text{false}\} \text{ is the truth value of Boolean expression } b \text{ evaluated in state } \sigma)
\end{align*}
\]

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:** \(\langle \text{skip}, \sigma \rangle \Downarrow \sigma\)
- **Assignments:** \(\langle x := a, \sigma \rangle \Downarrow \sigma[n/x]\)
- **Sequences:** \(\langle c_0; c_1, \sigma \rangle \Downarrow \sigma'' \quad (\langle c_0, \sigma' \rangle \Downarrow \sigma'\quad \langle c_1, \sigma' \rangle \Downarrow \sigma'')\)
- **Conditionals:** \(\langle \text{if } b \text{ then } c_0 \text{ else } c_1, \sigma \rangle \Downarrow \sigma' \quad (\langle \text{if } b \text{ then } c_0, \sigma \rangle \Downarrow \sigma'\quad \langle \text{if } b \text{ then } c_1, \sigma \rangle \Downarrow \sigma')\)
- **While statements:** \(\langle \text{while } b \text{ do } c, \sigma \rangle \Downarrow \sigma'' \quad (\langle \text{while } b \text{ do } c, \sigma' \rangle \Downarrow \sigma''\quad \langle \text{while } b \text{ do } c, \sigma' \rangle \Downarrow \sigma'')\)

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.