CS 6110 Lecture 5 IMP: Big-Step and Small-Step Semantics 1 February 2013

Lecturer: Andrew Myers

1 The IMP Language

We present a simple imperative language, IMP, along with structural operational semantics in two styles: small-step and big-step.

- 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 $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$).

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

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. We will assume that the initial value of every variable in the program is zero; that is, the initial store is $\lambda x.\, 0$.

A *configuration* is a pair $\langle c, \sigma \rangle$, where $c \in Com$ 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 $\langle c, \sigma \rangle \longrightarrow \langle c', \sigma' \rangle$ to indicate that the configuration $\langle c, \sigma \rangle$ reduces to $\langle c', \sigma' \rangle$ in one step, and we write $\langle c, \sigma \rangle \longrightarrow^* \langle c', \sigma' \rangle$ to indicate that

$\langle c,\sigma\rangle$ reduces to $\langle c',\sigma'\rangle$ in zero or more steps. Thus $\langle c,\sigma\rangle \longrightarrow^* \langle c',\sigma'\rangle$ 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 $\langle c_i,\sigma_i\rangle \longrightarrow \langle c_{i+1},\sigma_{i+1}\rangle$ for $0 \leq i \leq k-1$.

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

$$\begin{aligned}\longrightarrow \quad &: \quad (Com \times \Sigma) \to (Com \times \Sigma)\\ \longrightarrow_a \quad &: \quad (\mathit{AExp} \times \Sigma) \to \mathit{AExp}\\ \longrightarrow_b \quad &: \quad (\mathit{BExp} \times \Sigma) \to \mathit{BExp}\end{aligned}$$

2.1 Small-Step Operational Semantics

We now present the small-step semantics for evaluation of arithmetic and Boolean expressions and commands in IMP. Just as with the λ-calculus, the evaluation rules are presented as inference rules, which inductively define relations consisting of the acceptable computational steps in IMP.

2.1.1 Arithmetic Expressions

Variables:

$$\frac{}{\langle x,\sigma\rangle \longrightarrow_a \sigma(x)}$$

Arithmetic reductions:

$$\frac{}{\langle n_1 \oplus n_2,\sigma\rangle \longrightarrow_a n_3}\ (\text{where } n_3 = n_1 \oplus n_2)$$

Arithmetic subexpressions:

$$\frac{\langle a_1,\sigma\rangle \longrightarrow_a a_1'}{\langle a_1 \oplus a_2,\sigma\rangle \longrightarrow_a a_1' \oplus a_2} \qquad \frac{\langle a_2,\sigma\rangle \longrightarrow_a a_2'}{\langle n_1 \oplus a_2,\sigma\rangle \longrightarrow_a n_1 \oplus a_2'}$$

One subtle point: in the rule for arithmetic operations $\oplus$, the $\oplus$ appearing in the expression $a_1 \oplus a_2$ represents the operation symbol in the IMP language, which is a syntactic object; whereas the $\oplus$ appearing in the expression $n_1 \oplus n_2$ represents the actual operation in $\mathbb{Z}$, which is a semantic object. In this case, at the risk of confusion, we have used the same metanotation $\oplus$ for both of them.

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

2.2 Commands

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

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

- Assignments: $\dfrac{}{\langle x := n,\sigma\rangle \longrightarrow \langle \mathbf{skip},\sigma[x \mapsto n]\rangle} \qquad \dfrac{\langle a,\sigma\rangle \longrightarrow_a a'}{\langle x := a,\sigma\rangle \longrightarrow \langle x := a',\sigma\rangle}$

- Sequences: $\dfrac{\langle c_0,\sigma\rangle \longrightarrow \langle c_0',\sigma'\rangle}{\langle c_0; c_1,\sigma\rangle \longrightarrow \langle c_0'; c_1,\sigma'\rangle} \qquad \dfrac{}{\langle \mathbf{skip}; c_1,\sigma\rangle \longrightarrow \langle c_1,\sigma\rangle}$

[1]Winskel uses $\longrightarrow_1$ instead of $\longrightarrow$ to emphasize that only a single step is performed. Sometimes people use the arrow $\mapsto$ when the evaluation relation is a function.

- Conditionals:

$$\frac{\langle b,\sigma\rangle \longrightarrow_b b'}{\langle \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1,\sigma\rangle \longrightarrow \langle \textbf{if } b' \textbf{ then } c_0 \textbf{ else } c_1,\sigma\rangle}$$

$$\frac{}{\langle \textbf{if true then } c_0 \textbf{ else } c_1,\sigma\rangle \longrightarrow \langle c_0,\sigma\rangle} \qquad \frac{}{\langle \textbf{if false then } c_0 \textbf{ else } c_1,\sigma\rangle \longrightarrow \langle c_1,\sigma\rangle}$$

- While statements: $\dfrac{}{\langle \textbf{while } b \textbf{ do } c,\sigma\rangle \longrightarrow \langle \textbf{if } b \textbf{ then } (c; \textbf{while } b \textbf{ do } c) \textbf{ else skip},\sigma\rangle}$

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

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*[2], in which we specify the entire transition from a configuration (an $\langle$expression, state$\rangle$ pair) to a final value or store. 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 $\langle \textbf{skip},\sigma'\rangle$, starting in configuration $\langle c,\sigma\rangle$)
$\langle a,\sigma\rangle \Downarrow_a n$	(n is the integer value of arithmetic expression a evaluated in state σ)
$\langle b,\sigma\rangle \Downarrow_b t$	($t \in \{\mathit{true},\mathit{false}\}$ is the truth value of Boolean expression b evaluated in state σ)

The big-step rules for arithmetic and Boolean expressions are straightforward. The key when writing big-step rules is to think about how a recursive interpreter would evaluate the expression in question. Thus, the rules for arithmetic expressions are:

- Constants: $\dfrac{}{\langle n,\sigma\rangle \Downarrow_a n}$

- Variables: $\dfrac{}{\langle x,\sigma\rangle \Downarrow_a \sigma(x)}$

- Operations: $\dfrac{\langle a_0,\sigma\rangle \Downarrow_a n_0 \quad \langle a_1,\sigma\rangle \Downarrow_a n_1}{\langle a_0 \oplus a_1,\sigma\rangle \Downarrow_a n}$ (where $n = n_0 \oplus n_1$)

The rules for commands are a bit trickier:

- Skip: $\dfrac{}{\langle \textbf{skip},\sigma\rangle \Downarrow \sigma}$

- Assignments: $\dfrac{\langle a,\sigma\rangle \Downarrow_a n}{\langle x := a,\sigma\rangle \Downarrow \sigma[x \mapsto n]}$

- 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: $\dfrac{\langle b,\sigma\rangle \Downarrow_b \mathit{true} \quad \langle c_0,\sigma\rangle \Downarrow \sigma'}{\langle \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1,\sigma\rangle \Downarrow \sigma'} \quad \dfrac{\langle b,\sigma\rangle \Downarrow_b \mathit{false} \quad \langle c_1,\sigma\rangle \Downarrow \sigma'}{\langle \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1,\sigma\rangle \Downarrow \sigma'}$

- While statements: $\dfrac{\langle b,\sigma\rangle \Downarrow_b \mathit{false}}{\langle \textbf{while } b \textbf{ do } c,\sigma\rangle \Downarrow \sigma} \quad \dfrac{\langle b,\sigma\rangle \Downarrow_b \mathit{true} \quad \langle c,\sigma\rangle \Downarrow \sigma' \quad \langle \textbf{while } b \textbf{ do } c,\sigma'\rangle \Downarrow \sigma''}{\langle \textbf{while } b \textbf{ do } c,\sigma\rangle \Downarrow \sigma''}$

[2]Big-step semantics is also known as *natural semantics*.

4 Agreement of the big-step and small-step SOS

If the big-step and small-step semantics are both describing the same language, we would expect them to agree in some sense. In particular, the relations $\longrightarrow^*$ and $\Downarrow$ both capture the idea of a "large" evaluation. The expected agreement is the following: if $\langle c, \sigma \rangle$ is a configuration, and it evaluates in the small-step semantics to $\langle \textbf{skip}, \sigma' \rangle$, we expect that the small state σ' should be result of big-step evaluation. And vice-versa: the small-step semantics should be able to reproduce big-step evaluations. Formally,

$$\langle c, \sigma \rangle \longrightarrow^* \langle \textbf{skip}, \sigma' \rangle \iff \langle c, \sigma \rangle \Downarrow \sigma'$$

In fact, it is possible to prove this assertion using induction. However, we need some mathematical tools we have not yet acquired. Note that this statement about the agreement of the semantics has nothing to say about the agreement of nonterminating computations. This is because big-step semantics cannot talk directly about nontermination. If $\langle c, \sigma \rangle$ does not terminate, then there is no σ' such that $\langle c, \sigma \rangle \Downarrow \sigma'$.

5 Comparison of big-step vs. small-step SOS

5.1 Small-step

- Small-step semantics can clearly model more complex features, like concurrency, divergence, and run-time errors.
- Although one-step-at-a-time evaluation is useful for proving certain properties, in some cases it is unnecessary work to talk about each small step.

5.2 Big-step

- Big-step semantics more closely models a recursive interpreter.
- Big steps in reasoning make it quicker to prove some things, because there are fewer rules. The "boring" rules of the small-step semantics that specify order of evaluation are folded into the way that state is threaded through the big-step rules.
- Because evaluation skips over intermediate steps, all programs without final configurations (infinite loops, errors, stuck configurations) look the same. So you sometimes can't prove things related to these kinds of configurations.