In this lecture we introduce a very simple imperative language IMP, along with two systems of rules for evaluation called *small-step* and *big-step* semantics. These both fall under the general style called *structural operational semantics* (SOS) (although there is a view of big-step semantics that one might call denotational). We will also discuss why both the big-step and small-step approaches can be useful.

1 The IMP Language

1.1 Syntax of IMP

There are three distinct types of expressions in IMP:

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

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

Let $n, n_0, n_1, \ldots$ denote integers (elements of $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$). We will use n both as a number (a semantic object) and as a numeral (a syntactic object) representing the number n. Ordinarily this little bit of ambiguity does not cause any confusion, as the numbers and the numerals are in one-to-one correspondence, so there is really no need to distinguish them. Similarly, there is usually no need to distinguish between the Boolean constants true and false (syntactic objects) and the Boolean values $\mathit{true}, \mathit{false} \in \mathbb{2}$ (semantic objects).

Let Var be a countable set of variables ranging over $\mathbb{Z}$. Elements of Var are denoted $x, x_0, x_1 \ldots$.

The BNF grammar for IMP is

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

Note that in this definition, $n + m$ denotes the syntactic expression with three symbols n, $+$, and m, not to the number that is the sum of n and m.

1.2 Environments and Configurations

An *environment* (also known as a *valuation* or *store*) is a function $\sigma : \mathit{Var} \to \mathbb{Z}$ that assigns an integer to each variable. Environments are denoted $\sigma, \sigma_1, \tau, \ldots$ and the set of all environments is denoted Env. In IMP semantics, environments are always total functions.

A *configuration* is a pair $\langle c, \sigma \rangle$, where $c \in \mathsf{Com}$ is a command and σ is an environment. 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 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 $\langle \mathsf{skip}, \sigma\rangle$ (if ever). We write $\langle c, \sigma\rangle \xrightarrow{1}_c \langle c', \sigma'\rangle$ to indicate that the configuration $\langle c, \sigma\rangle$ reduces to $\langle c', \sigma'\rangle$ in one step, and we write $\langle c, \sigma\rangle \to_c \langle c', \sigma'\rangle$ to indicate that $\langle c, \sigma\rangle$ reduces to $\langle c', \sigma'\rangle$ in zero or more steps. Thus $\langle c, \sigma\rangle \to_c \langle c', \sigma'\rangle$ iff there exist $k \geq 0$ 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 $\langle c_i, \sigma_i\rangle \xrightarrow{1}_c \langle c_{i+1}, \sigma_{i+1}\rangle$ for $0 \leq i \leq k-1$.

To be completely proper, we will define auxiliary small-step operators $\to_a$ and $\to_b$ for arithmetic and Boolean expressions, respectively, as well as $\to_c$ for commands. The types of these operators are

$$\to_a : (\mathsf{AExp} \times \mathit{Env}) \to \mathsf{AExp} \qquad \to_b : (\mathsf{BExp} \times \mathit{Env}) \to \mathsf{BExp} \qquad \to_c : (\mathsf{Com} \times \mathit{Env}) \to (\mathsf{Com} \times \mathit{Env})$$

Intuitively, $\langle a, \sigma\rangle \to_a n$ if the expression a evaluates to the constant n in environment σ.

We now present the small-step rules for evaluation in IMP. Just as with the λ-calculus, evaluation is defined by a set of inference rules that inductively define relations consisting of acceptable computation steps.

2.1 Arithmetic and Boolean Expressions

- Variables:

$$\frac{}{\langle x, \sigma\rangle \xrightarrow{1}_a \sigma(x)}$$

- Arithmetic:

$$\frac{}{\langle n_1 + n_2, \sigma\rangle \xrightarrow{1}_a n_3} \text{ (where } n_3 \text{ is the sum of } n_1 \text{ and } n_2\text{)}$$

$$\frac{\langle a_1, \sigma\rangle \xrightarrow{1}_a a_1'}{\langle a_1 + a_2, \sigma\rangle \xrightarrow{1}_a a_1' + a_2} \qquad \frac{\langle a_2, \sigma\rangle \xrightarrow{1}_a a_2'}{\langle n_1 + a_2, \sigma\rangle \xrightarrow{1}_a n_1 + a_2'}$$

  and similar rules for $*$ and $-$. These rules say: If the reduction above the line can be performed, then the reduction below the line can be performed. The rules are thus inductive on the structure of the expression to be evaluated.

  Note that there is no rule that applies in the case $\langle n, \sigma\rangle$. This configuration is *irreducible.* In all other cases, there is exactly one rule that applies.

The rules for Booleans and comparison operators are similar. We leave them as exercises.

2.2 Commands

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

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

The construct $[n/x]$ is called a *rebinding operator*. It is a meta-operator that is used to rebind a variable to a different value in the environment. Note that in the definition of $\sigma[n/x]$, the condition "$y \neq x$" does not mean that y and x are bound to different values, but that they are syntactically different variables.

- Assignment:

$$\frac{}{\langle x := n,\, \sigma\rangle \xrightarrow{1}_c \langle \mathsf{skip},\, \sigma[n/x]\rangle} \qquad \frac{\langle a,\, \sigma\rangle \xrightarrow{1}_a a'}{\langle x := a,\, \sigma\rangle \xrightarrow{1}_c \langle x := a',\, \sigma\rangle}$$

- Sequence:

$$\frac{\langle c_0,\, \sigma\rangle \xrightarrow{1}_c \langle c_0',\, \sigma'\rangle}{\langle c_0 \,;\, c_1,\, \sigma\rangle \xrightarrow{1}_c \langle c_0' \,;\, c_1,\, \sigma'\rangle} \qquad \frac{}{\langle \mathsf{skip} \,;\, c_1,\, \sigma\rangle \xrightarrow{1}_c \langle c_1,\, \sigma\rangle}$$

- Conditional:

$$\frac{\langle b,\, \sigma\rangle \xrightarrow{1}_b b'}{\langle \mathsf{if}\ b\ \mathsf{then}\ c_0\ \mathsf{else}\ c_1,\, \sigma\rangle \xrightarrow{1}_c \langle \mathsf{if}\ b'\ \mathsf{then}\ c_0\ \mathsf{else}\ c_1,\, \sigma\rangle}$$

$$\frac{}{\langle \mathsf{if\ true\ then}\ c_0\ \mathsf{else}\ c_1,\, \sigma\rangle \xrightarrow{1}_c \langle c_0,\, \sigma\rangle} \qquad \frac{}{\langle \mathsf{if\ false\ then}\ c_0\ \mathsf{else}\ c_1,\, \sigma\rangle \xrightarrow{1}_c \langle c_1,\, \sigma\rangle}$$

- While statement:

$$\frac{}{\langle \mathsf{while}\ b\ \mathsf{do}\ c,\, \sigma\rangle \xrightarrow{1}_c \langle \mathsf{if}\ b\ \mathsf{then}\ (c \,;\, \mathsf{while}\ b\ \mathsf{do}\ c)\ \mathsf{else\ skip},\, \sigma\rangle}$$

There is no rule for skip; the configuration $\langle \mathsf{skip},\, \sigma\rangle$ is irreducible. In all other cases, there is exactly one rule that applies. These rules tell us all we need to know to run IMP programs.

3 Big-Step Semantics

As an alternative to small-step structural 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, environment⟩ pair) to a final value. For arithmetic expressions, the final value is an integer $n \in \mathbb{Z}$; for Boolean expressions, it is a Boolean truth value $\mathit{true}, \mathit{false} \in \mathbb{2}$; and for commands, it is an environment $\sigma : \mathit{Var} \to \mathbb{Z}$. Thus

$$\Downarrow_a\ : (\mathsf{AExp} \times \mathit{Env}) \to \mathbb{Z} \qquad \Downarrow_b\ : (\mathsf{BExp} \times \mathit{Env}) \to \mathbb{2} \qquad \Downarrow_c\ : (\mathsf{Com} \times \mathit{Env}) \rightharpoonup \mathit{Env}$$

Note that $\Downarrow_a$ and $\Downarrow_b$ are total functions (denoted with $\to$) because the evaluation of arithmetic and Boolean expressions always halts and results in a final value, whereas $\Downarrow_c$ is a partial function (denoted with $\rightharpoonup$) since programs do not necessarily halt.

Here $\mathbb{2}$ represents the two-element Boolean algebra consisting of the two truth values $\{\mathit{true}, \mathit{false}\}$ with the usual Boolean operations. Then

- $\langle a,\, \sigma\rangle \Downarrow_a n$ says that $n \in \mathbb{Z}$ is the integer value of arithmetic expression a evaluated in environment σ;
- $\langle b,\, \sigma\rangle \Downarrow_b t$ says that $t \in \mathbb{2}$ is the truth value of Boolean expression b evaluated in environment σ; and
- $\langle c,\, \sigma\rangle \Downarrow_c \sigma'$ says that σ' is the environment of the final configuration, starting in configuration $\langle c,\, \sigma\rangle$.

3.1 Arithmetic and Boolean Expressions

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. The rules for arithmetic expressions are:

- Constants:

$$\frac{}{\langle n,\, \sigma\rangle \Downarrow_a n}$$

- Variables:

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

- Operations:

$$\frac{\langle a_0,\, \sigma\rangle \Downarrow_a n_0 \qquad \langle a_1,\, \sigma\rangle \Downarrow_a n_1}{\langle a_0 + a_1,\, \sigma\rangle \Downarrow_a n_2} \text{ (where } n_2 \text{ is the sum of } n_0 \text{ and } n_1\text{)}$$

  and similarly for $*$ and $-$.

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

3.2 Commands

- Skip:

$$\frac{}{\langle \mathsf{skip},\, \sigma\rangle \Downarrow_c \sigma}$$

- Assignments:

$$\frac{\langle a,\, \sigma\rangle \Downarrow_a n}{\langle x := a,\, \sigma\rangle \Downarrow_c \sigma[n/x]}$$

- Sequences:

$$\frac{\langle c_0,\, \sigma\rangle \Downarrow_c \sigma' \qquad \langle c_1,\, \sigma'\rangle \Downarrow_c \sigma''}{\langle c_0\, ;\, c_1,\, \sigma\rangle \Downarrow_c \sigma''}$$

- Conditionals:

$$\frac{\langle b,\, \sigma\rangle \Downarrow_b \mathit{true} \qquad \langle c_0,\, \sigma\rangle \Downarrow_c \sigma'}{\langle \mathsf{if}\ b\ \mathsf{then}\ c_0\ \mathsf{else}\ c_1,\, \sigma\rangle \Downarrow_c \sigma'} \qquad \frac{\langle b,\, \sigma\rangle \Downarrow_b \mathit{false} \qquad \langle c_1,\, \sigma\rangle \Downarrow_c \sigma'}{\langle \mathsf{if}\ b\ \mathsf{then}\ c_0\ \mathsf{else}\ c_1,\, \sigma\rangle \Downarrow_c \sigma'}$$

- While statements:

$$\frac{\langle b,\, \sigma\rangle \Downarrow_b \mathit{false}}{\langle \mathsf{while}\ b\ \mathsf{do}\ c,\, \sigma\rangle \Downarrow_c \sigma} \qquad \frac{\langle b,\, \sigma\rangle \Downarrow_b \mathit{true} \qquad \langle c,\, \sigma\rangle \Downarrow_c \sigma' \qquad \langle \mathsf{while}\ b\ \mathsf{do}\ c,\, \sigma'\rangle \Downarrow_c \sigma''}{\langle \mathsf{while}\ b\ \mathsf{do}\ c,\, \sigma\rangle \Downarrow_c \sigma''}$$

4 Big-Step vs. Small-Step SOS

If the big-step and small-step semantics both describe the same language, we would expect them to agree. In particular, we would expect that if $\langle c,\, \sigma\rangle$ is a configuration that evaluates in the small-step semantics to $\langle \mathsf{skip},\, \sigma'\rangle$, then σ' should also be the result of the big-step evaluation, and vice-versa. Formally,

Theorem 8.1. *For all commands $c \in \mathsf{Com}$ and environments $\sigma, \tau \in \mathit{Env}$,*

$$\langle c,\, \sigma\rangle \to_c \langle \mathsf{skip},\, \tau\rangle \;\Leftrightarrow\; \langle c,\, \sigma\rangle \Downarrow_c \tau.$$

Proof. We can express the idea that the two semantics should agree on terminating executions by connecting the relations $\rightarrow$ and $\Downarrow$:

$$\langle a,\, \sigma\rangle \rightarrow_a \overline{n} \;\Leftrightarrow\; \langle a,\, \sigma\rangle \Downarrow_a n \tag{1}$$
$$\langle b,\, \sigma\rangle \rightarrow_b \overline{t} \;\Leftrightarrow\; \langle b,\, \sigma\rangle \Downarrow_b t \tag{2}$$
$$\langle c,\, \sigma\rangle \rightarrow_c \langle \mathsf{skip},\, \sigma'\rangle \;\Leftrightarrow\; \langle c,\, \sigma\rangle \Downarrow_c \sigma', \tag{3}$$

where $\overline{\mathit{false}} = \mathsf{false}$ and $\overline{\mathit{true}} = \mathsf{true}$. These can all be proved using induction. We can prove (1) and (2) as lemmas separately, then use these lemmas to prove (3).

Let us just show a few of the cases for (3). To prove the forward implication of (3), we can use structural induction on c. The converse requires induction on the derivation of the big-step evaluation.

Suppose we are given $\langle c,\, \sigma\rangle \Downarrow_c \sigma'$. The form of the derivation $\langle c,\, \sigma\rangle \Downarrow_c \sigma'$ depends on the form of c.

- Case skip. In this case we know $\sigma = \sigma'$, and trivially, $\langle \mathsf{skip},\, \sigma\rangle \rightarrow \langle \mathsf{skip},\, \sigma\rangle$.
- Case $x := a$. In this case we know from the premises that $\langle a,\, \sigma\rangle \Downarrow_a n$ for some n and that $\sigma' = \sigma[n/x]$. We will need a lemma to the effect that $\langle a,\, \sigma\rangle \rightarrow_a \overline{n}$ implies that $\langle x := a,\, \sigma\rangle \rightarrow_c \langle x := \overline{n},\, \sigma\rangle$. This result can be proved easily using an induction on the number of steps in the derivation $\langle a,\, \sigma\rangle \rightarrow_a \overline{n}$. Given this and (1), we have that $\langle x := a,\, \sigma\rangle \rightarrow_c \langle x := \overline{n},\, \sigma\rangle$ and $\langle x := \overline{n},\, \sigma\rangle \xrightarrow{1}_c \langle \mathsf{skip},\, \sigma[n/x]\rangle$, so $\langle x := a,\, \sigma\rangle \rightarrow_c \langle \mathsf{skip},\, \sigma[n/x]\rangle$.
- Case $\mathsf{while}\ b\ \mathsf{do}\ c$, where $\langle b,\, \sigma\rangle \Downarrow_b \mathit{false}$. Then $\sigma = \sigma'$. In small-step semantics, we have an initial step

  $$\langle \mathsf{while}\ b\ \mathsf{do}\ c,\, \sigma\rangle \;\xrightarrow{1}_c\; \langle \mathsf{if}\ b\ \mathsf{then}\ (c\ ;\ \mathsf{while}\ b\ \mathsf{do}\ c)\ \mathsf{else\ skip},\, \sigma\rangle. \tag{4}$$

  By (2), $\langle b,\, \sigma\rangle \rightarrow_b \mathsf{false}$. We need another lemma to the effect that if $\langle b,\, \sigma\rangle \rightarrow_b t$, then

  $$\langle \mathsf{if}\ b\ \mathsf{then}\ c\ \mathsf{else}\ c',\, \sigma\rangle \;\rightarrow_c\; \langle \mathsf{if}\ t\ \mathsf{then}\ c\ \mathsf{else}\ c',\, \sigma\rangle. \tag{5}$$

  In this case $t = \mathsf{false}$, thus (4) becomes $\langle \mathsf{skip},\, \sigma\rangle$ as desired.
- Case $\mathsf{while}\ b\ \mathsf{do}\ c$, where $\langle b,\, \sigma\rangle \Downarrow_b \mathit{true}$. This is the most interesting case in the entire proof. In small-step semantics, we have the initial step (4), as in the previous case. From (5) with $t = \mathsf{true}$, (4) becomes $\langle c\ ;\ \mathsf{while}\ b\ \mathsf{do}\ c,\, \sigma\rangle$. We need one more lemma for stitching together small-step executions:

  $$\langle c_1,\, \sigma\rangle \rightarrow_c \langle \mathsf{skip},\, \sigma''\rangle \;\wedge\; \langle c_2,\, \sigma''\rangle \rightarrow_c \langle \mathsf{skip},\, \sigma'\rangle \;\Rightarrow\; \langle c_1\ ;\ c_2,\, \sigma\rangle \rightarrow_c \langle \mathsf{skip},\, \sigma'\rangle. \tag{6}$$

  This can be proved by induction on the number of steps.

  Because $\langle \mathsf{while}\ b\ \mathsf{do}\ c,\, \sigma\rangle \Downarrow_c \sigma'$ and $\langle b,\, \sigma\rangle \Downarrow_b \mathit{true}$, we know that $\langle c,\, \sigma\rangle \Downarrow_c \sigma''$ and $\langle \mathsf{while}\ b\ \mathsf{do}\ c,\, \sigma''\rangle \Downarrow_c \sigma'$ for some σ''. Because these two derivations are subderivations of the derivation $\langle \mathsf{while}\ b\ \mathsf{do}\ c,\, \sigma\rangle \Downarrow_c \sigma'$, the induction hypothesis gives us $\langle c,\, \sigma\rangle \rightarrow_c \langle \mathsf{skip},\, \sigma''\rangle$ and $\langle \mathsf{while}\ b\ \mathsf{do}\ c,\, \sigma''\rangle \rightarrow_c \sigma'$. Using (6), we have $\langle c\ ;\ \mathsf{while}\ b\ \mathsf{do}\ c,\, \sigma\rangle \rightarrow_c \langle \mathsf{skip},\, \sigma'\rangle$.

  We could not have used structural induction for this proof, because the induction step involved relating an evaluation of the command $\mathsf{while}\ b\ \mathsf{do}\ c$ to a different evaluation of the same command rather than to an evaluation of a subexpression.

$\square$

Note that this statement about the agreement of big-step and small-step 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 τ such that $\langle c,\, \sigma\rangle \Downarrow_c \tau$.

Small-step semantics can model more complex features such as nonterminating programs and concurrency. However, in many cases it involves unnecessary extra work.

If we do not care about modeling nonterminating computations, it is often easier to reason in terms of big-step semantics. Moreover, big-step semantics more closely models an actual recursive interpreter. However, because evaluation skips over intermediate steps, all programs without final configurations are indistinguishable.

5 Big-Step as Denotational Semantics

Denotational semantics refers to the interpretation of programs as mathematical objects. We mentioned at the beginning that there is a view of big-step operational semantics that one might call denotational in this sense. As presented earlier, the big-step relations for arithmetic, Booleans, and commands had types

$$\Downarrow_a\ : (\mathsf{AExp} \times \mathit{Env}) \to \mathbb{Z} \qquad \Downarrow_b\ : (\mathsf{BExp} \times \mathit{Env}) \to \mathbb{2} \qquad \Downarrow_c\ : (\mathsf{Com} \times \mathit{Env}) \rightharpoonup \mathit{Env}$$

respectively. However, these functions can be curried to have types

$$\Downarrow_a\ : \mathsf{AExp} \to (\mathit{Env} \to \mathbb{Z}) \qquad \Downarrow_b\ : \mathsf{BExp} \to (\mathit{Env} \to \mathbb{2}) \qquad \Downarrow_c\ : \mathsf{Com} \to (\mathit{Env} \rightharpoonup \mathit{Env})$$

which says that

- arithmetic expressions can be interpreted as functions $\mathit{Env} \to \mathbb{Z}$;
- Boolean expressions can be interpreted as functions $\mathit{Env} \to \mathbb{2}$; and
- commands can be interpreted as partial functions $\mathit{Env} \rightharpoonup \mathit{Env}$.