IMP: An Imperative Programming Language

CPSC 509: Programming Language Principles

Ronald Garcia

28 January 2013

The Boolean Language was rather small and possibly a bit unfamiliar for some (although using Racket or some other functional programming language may have given you a similar flavor).

Now it's time for a language that may look more familiar. It is considered to be an *imperative* programming language in that it has commands that act as instructions to do something, not simply expressions to be reduced. We'll make clearer what we mean by that below. This is also a chance to see the semantics of an even more complex language

1 IMP

In this section we define the IMP Language. IMP is an imperative language in the spirit of Pascal, Algol, and other languages that support so-called *structured programming*.

$$
\begin{array}{rcl}
n \in \mathbb{Z}, \quad bv \in \textsc{Bool}, \quad X \in \textsc{Loc}, \quad a \in \textsc{AExp}, \quad b \in \textsc{BExp}, \quad c \in \textsc{Com}, \\
a & ::= & X \mid n \mid a + a \mid a - a \mid a * a \\
b & ::= & \mathsf{true} \mid \mathsf{false} \mid a = a \mid a \leq a \mid \neg b \mid b \wedge b \mid b \vee b \\
c & ::= & \mathsf{skip} \mid X := a \mid c; c \mid \mathsf{if}\ b\ \mathsf{then}\ c\ \mathsf{else}\ c \mid \mathsf{while}\ b\ \mathsf{do}\ c \\
bv & ::= & \mathsf{true} \mid \mathsf{false}
\end{array}
$$

This is just the syntax, but given its similarity to many programming languages in the wild, you can probably guess what some programs do.

IMP has arithmetic expressions, boolean expressions, and commands. We refer to all three categories generically as *expressions*. The language features locations X[1]. Since locations are arithmetic expressions, we know that they always stand for numbers.

1.1 Big-step semantics

For the semantics of this language, we use the big-step approach. First, we need to define a few sets that are used in specifying the semantics.

$$
\begin{array}{c}
\sigma \in \textsc{Store} = \textsc{Loc} \to \mathbb{Z} \\
\textsc{ACfg} = \textsc{AExp} \times \textsc{Store} \\
\textsc{BCfg} = \textsc{BExp} \times \textsc{Store} \\
\textsc{CCfg} = \textsc{Com} \times \textsc{Store}
\end{array}
$$

First, we introduce the set of Stores σ, each of which is a mapping from locations to numbers $\mathbb{Z}$. We use stores to model the changing values associated with particular locations. You can think of it as an abstract representation of the state of a computer's memory while running a program.

[1] These are typically called *variables* by programmers, but we're saving that name for a very particular concept

In the IMP language, we must always interpret the meaning of its terms (arithmetic expressions, boolean expressions, and commands) relative to the values in the current store. Notice that arithmetic expressions include references to locations, whose values are determined by the store. Furthermore, boolean expressions can contain arithmetic expressions, and finally commands can contain boolean expressions. These relationships make our depenence on stores deep. For this reason we need to introduce a notion of *configurations* for each kind of term, which pairs a term with a store. It's these configurations that our semantics will consider.

$$\begin{aligned} \Downarrow_{\text{AExp}} &\subseteq \text{ACfg} \times \mathbb{N} \\ \Downarrow_{\text{BExp}} &\subseteq \text{BCfg} \times \text{Bool} \\ \Downarrow_{\text{Com}} &\subseteq \text{CCfg} \times \text{Store} \end{aligned}$$

$$\text{(enum)}\ \frac{}{\langle n, \sigma\rangle \Downarrow_{\text{AExp}} n} \qquad \text{(eloc)}\ \frac{}{\langle X, \sigma\rangle \Downarrow_{\text{AExp}} \sigma(X)} \qquad \text{(eplus)}\ \frac{\langle a_1, \sigma\rangle \Downarrow_{\text{AExp}} n_1 \quad \langle a_2, \sigma\rangle \Downarrow_{\text{AExp}} n_2}{\langle a_1 + a_2, \sigma\rangle \Downarrow_{\text{AExp}} n_1 + n_2}$$

$$\text{(eminus)}\ \frac{\langle a_1, \sigma\rangle \Downarrow_{\text{AExp}} n_1 \quad \langle a_2, \sigma\rangle \Downarrow_{\text{AExp}} n_2}{\langle a_1 - a_2, \sigma\rangle \Downarrow_{\text{AExp}} n_1 - n_2} \qquad \text{(etimes)}\ \frac{\langle a_1, \sigma\rangle \Downarrow_{\text{AExp}} n_1 \quad \langle a_2, \sigma\rangle \Downarrow_{\text{AExp}} n_2}{\langle a_1 * a_2, \sigma\rangle \Downarrow_{\text{AExp}} n_1 * n_2}$$

$$\text{(etrue)}\ \frac{}{\langle \text{true}, \sigma\rangle \Downarrow_{\text{BExp}} \text{true}} \qquad \text{(efalse)}\ \frac{}{\langle \text{false}, \sigma\rangle \Downarrow_{\text{BExp}} \text{false}}$$

$$\text{(eeq)}\ \frac{\langle a_1, \sigma\rangle \Downarrow_{\text{AExp}} n_1 \quad \langle a_2, \sigma\rangle \Downarrow_{\text{AExp}} n_2}{\langle a_1 = a_2, \sigma\rangle \Downarrow_{\text{BExp}} bv}\ bv = \begin{cases} \text{true} & n_1 = n_2 \\ \text{false} & n_1 \neq n_2 \end{cases}$$

$$\text{(eleq)}\ \frac{\langle a_1, \sigma\rangle \Downarrow_{\text{AExp}} n_1 \quad \langle a_2, \sigma\rangle \Downarrow_{\text{AExp}} n_2}{\langle a_1 \leq a_2, \sigma\rangle \Downarrow_{\text{BExp}} bv}\ bv = \begin{cases} \text{true} & n_1 \leq n_2 \\ \text{false} & n_1 > n_2 \end{cases}$$

$$\text{(enot)}\ \frac{\langle b, \sigma\rangle \Downarrow_{\text{BExp}} bv_1}{\langle \neg b, \sigma\rangle \Downarrow_{\text{BExp}} bv_2}\ bv_2 = \begin{cases} \text{true} & bv_1 = \text{false} \\ \text{false} & bv_1 = \text{true} \end{cases}$$

$$\text{(eand)}\ \frac{\langle b_1, \sigma\rangle \Downarrow_{\text{BExp}} bv_1 \quad \langle b_2, \sigma\rangle \Downarrow_{\text{BExp}} bv_2}{\langle b_1 \wedge b_2, \sigma\rangle \Downarrow_{\text{BExp}} bv_3}\ bv_3 = \begin{cases} \text{true} & bv_1 = bv_2 = \text{true} \\ \text{false} & \text{otherwise} \end{cases}$$

$$\text{(eor)}\ \frac{\langle b_1, \sigma\rangle \Downarrow_{\text{BExp}} bv_1 \quad \langle b_2, \sigma\rangle \Downarrow_{\text{BExp}} bv_2}{\langle b_1 \vee b_2, \sigma\rangle \Downarrow_{\text{BExp}} bv_3}\ bv_3 = \begin{cases} \text{true} & bv_1 = \text{true or } bv_2 = \text{true} \\ \text{false} & \text{otherwise} \end{cases}$$

$$\text{(eskip)}\ \frac{}{\langle \text{skip}, \sigma\rangle \Downarrow_{\text{Com}} \sigma} \qquad \text{(eassign)}\ \frac{\langle a, \sigma\rangle \Downarrow_{\text{AExp}} n}{\langle X := a, \sigma\rangle \Downarrow_{\text{Com}} \sigma[X \mapsto n]}$$

$$\text{(eseq)}\ \frac{\langle c_1, \sigma\rangle \Downarrow_{\text{Com}} \sigma' \quad \langle c_2, \sigma'\rangle \Downarrow_{\text{Com}} \sigma''}{\langle c_1 ; c_2, \sigma\rangle \Downarrow_{\text{Com}} \sigma''} \qquad \text{(eif-t)}\ \frac{\langle b, \sigma\rangle \Downarrow_{\text{BExp}} \text{true} \quad \langle c_1, \sigma\rangle \Downarrow_{\text{Com}} \sigma'}{\langle \text{if } b \text{ then } c_1 \text{ else } c_2, \sigma\rangle \Downarrow_{\text{Com}} \sigma'}$$

$$\text{(eif-f)}\ \frac{\langle b, \sigma\rangle \Downarrow_{\text{BExp}} \text{false} \quad \langle c_2, \sigma\rangle \Downarrow_{\text{Com}} \sigma'}{\langle \text{if } b \text{ then } c_1 \text{ else } c_2, \sigma\rangle \Downarrow_{\text{Com}} \sigma'} \qquad \text{(ewhile-f)}\ \frac{\langle b, \sigma\rangle \Downarrow_{\text{BExp}} \text{false}}{\langle \text{while } b \text{ do } c, \sigma\rangle \Downarrow_{\text{Com}} \sigma}$$

$$\text{(ewhile-t)}\ \frac{\langle b, \sigma\rangle \Downarrow_{\text{BExp}} \text{true} \quad \langle c, \sigma\rangle \Downarrow_{\text{Com}} \sigma' \quad \langle \text{while } b \text{ do } c, \sigma'\rangle \Downarrow_{\text{Com}} \sigma''}{\langle \text{while } b \text{ do } c, \sigma\rangle \Downarrow_{\text{Com}} \sigma''}$$

1.2 Defining the evaluator

We define the evaluator for the language as follows:

$$
\begin{aligned}
&eval : \text{CCFG} \to \text{STORE} \cup \{\infty\} \\
&eval(c) = \sigma \text{ iff } \langle c, \sigma_z \rangle \Downarrow_{\text{COM}} \sigma, \\
&eval(c) = \infty \text{ otherwise}
\end{aligned}
$$

where σ_z is the everywhere-zero store $\sigma_z(X) = 0$. Our semantics for this language is defined in terms of the big-step semantics, but imposes the additional constraint that the program is run starting from a well-defined store. For now, we do not explicitly describe divergence. Instead we decree that any program which does not converge (i.e. big-step to a final store) diverges. This perspective is common in the literature, albeit somewhat unsatisfying and dangerous.

1.3 An Example

Now given our semantics, we can calculate the evaluation of a progam. Consider the program:

$$
\begin{aligned}
&\mathsf{X} := 1; \\
&\textsf{while } \neg(\mathsf{X} = 0) \textsf{ do } \mathsf{X} := \mathsf{X} - 1
\end{aligned}
$$

Let's compute it. Recall that we need to find out how it big-steps from σ_z. Let's do it!

$$
\dfrac{
  \dfrac{}{\langle 1, \sigma_z \rangle \Downarrow_{\text{AExp}} 1}
}{
  \langle \mathsf{X} := 1, \sigma_z \rangle \Downarrow_{\text{Com}} \sigma_z[X \mapsto 1]
}
\quad
\dfrac{
  \dfrac{
    \dfrac{
      \dfrac{}{\langle \mathsf{X}, \sigma_z[X \mapsto 1] \rangle \Downarrow_{\text{AExp}} 1}
      \quad
      \dfrac{}{\langle 0, \sigma_z[X \mapsto 1] \rangle \Downarrow_{\text{AExp}} 0}
    }{
      \langle \mathsf{X} = 0, \sigma_z[X \mapsto 1] \rangle \Downarrow_{\text{BExp}} \mathsf{false}
    }
  }{
    \langle \neg(\mathsf{X} = 0), \sigma_z[X \mapsto 1] \rangle \Downarrow_{\text{BExp}} \mathsf{true}
  }
  \quad
  \dfrac{
    \dfrac{
      \dfrac{}{\langle \mathsf{X}, \sigma_z[X \mapsto 1] \rangle \Downarrow_{\text{AExp}} 1}
      \quad
      \dfrac{}{\langle 1, \sigma_z[X \mapsto 1] \rangle \Downarrow_{\text{AExp}} 1}
    }{
      \langle \mathsf{X} - 1, \sigma_z[X \mapsto 1] \rangle \Downarrow_{\text{AExp}} 0
    }
  }{
    \langle \mathsf{X} := \mathsf{X} - 1, \sigma_z[X \mapsto 1] \rangle \Downarrow_{\text{Com}} \sigma_z
  }
  \quad
  \dfrac{
    \dfrac{
      \dfrac{
        \dfrac{}{\langle \mathsf{X}, \sigma_z \rangle \Downarrow_{\text{AExp}} 0}
        \quad
        \dfrac{}{\langle 0, \sigma_z \rangle \Downarrow_{\text{AExp}} 0}
      }{
        \langle \mathsf{X} = 0, \sigma_z \rangle \Downarrow_{\text{BExp}} \mathsf{true}
      }
    }{
      \langle \neg(\mathsf{X} = 0), \sigma_z \rangle \Downarrow_{\text{BExp}} \mathsf{false}
    }
  }{
    \langle \mathsf{while}\ \neg(\mathsf{X} = 0)\ \mathsf{do}\ \mathsf{X} := \mathsf{X} - 1, \sigma_z \rangle \Downarrow_{\text{Com}} \sigma'
  }
}{
  \langle \mathsf{while}\ \neg(\mathsf{X} = 0)\ \mathsf{do}\ \mathsf{X} := \mathsf{X} - 1, \sigma_z[X \mapsto 1] \rangle \Downarrow_{\text{Com}} \sigma_z
}
$$

$$
\overline{\langle \mathsf{X} := 1;\ \mathsf{while}\ \neg(\mathsf{X} = 0)\ \mathsf{do}\ \mathsf{X} := \mathsf{X} - 1, \sigma_z \rangle \Downarrow_{\text{Com}} \sigma_z}
$$

Notice that in this tree, the same while command is big-stepped twice, but it produces different values because it starts at a different store. This behavior is at the heart of imperative programming.

1.4 Some commentary on IMP

One interesting aspect of IMP compared to our language of Boolean and Arithmetic expressions is that IMP programs by design can never crash. That is to say, you don't have to worry about predicates not evaluating to booleans, arithmetics not evaluating to numbers etc. The language is syntactically constrained to prevent such errors. This is a taste of a more sophisticated syntactic restriction mechanism that we call *types*. We discuss this in more depth later.

2 Backward Reasoning Principles

As with our previous inductive definitions, we can easily establish a set of very useful backward reasoning principles.

Proposition 1 (Backward Reasoning)**.**

1. *If* $\langle n_1, \sigma\rangle \Downarrow_{\text{AExp}} n_2$, *then* $n_1 = n_2$.
2. *If* $\langle X, \sigma\rangle \Downarrow_{\text{AExp}} n$ *then* $n = \sigma(X)$.
3. *If* $\langle a_1 + a_2, \sigma\rangle \Downarrow_{\text{AExp}} n$ *then* $\langle a_1, \sigma\rangle \Downarrow_{\text{AExp}} n_1$, $\langle a_2, \sigma\rangle \Downarrow_{\text{AExp}} n_2$, *and* $n = n_1 + n_2$.
4. *If* $\langle a_1 * a_2, \sigma\rangle \Downarrow_{\text{AExp}} n$ *then* $\langle a_1, \sigma\rangle \Downarrow_{\text{AExp}} n_1$, $\langle a_2, \sigma\rangle \Downarrow_{\text{AExp}} n_2$, *and* $n = n_1 * n_2$.
5. *etc.* ...
6. *If* $\langle c_1;c_2, \sigma\rangle \Downarrow_{\text{Com}} \sigma''$ *then* $\langle c_1, \sigma\rangle \Downarrow_{\text{Com}} \sigma'$ *and* $\langle c_2, \sigma'\rangle \Downarrow_{\text{Com}} \sigma''$
7. *If* $\langle$ *if* b *then* c_1 *else* $c_2, \sigma\rangle \Downarrow_{\text{Com}} \sigma$ *then either*
   (a) $\langle b, \sigma\rangle \Downarrow_{\text{BExp}}$ *true and* $\langle c_1, \sigma\rangle \Downarrow_{\text{Com}} \sigma'$; *or*
   (b) $\langle b, \sigma\rangle \Downarrow_{\text{BExp}}$ *false and* $\langle c_2, \sigma\rangle \Downarrow_{\text{Com}} \sigma'$.
8. *If* $\langle$ *while* b *do* $c, \sigma\rangle \Downarrow_{\text{Com}} \sigma'$ *then either*
   (a) $\langle b, \sigma\rangle \Downarrow_{\text{BExp}}$ *true*, $\langle c, \sigma\rangle \Downarrow_{\text{Com}} \sigma''$, *and* $\langle$ *while* b *do* $c, \sigma''\rangle \Downarrow_{\text{Com}} \sigma'$; *or*
   (b) $\langle b, \sigma\rangle \Downarrow_{\text{BExp}}$ *false and* $\sigma' = \sigma$.

As we saw in class, the inversion lemmas can be used as the basis for implementing an evaluator for IMP. Conceptually, each lemma suggests a strategy for searching for a proof that an expression big-steps.

3 Reasoning about IMP Programs

The big-step semantics for IMP gives us a means of proving whether and what a particular program evaluates to, and even gives us some guidance for implementing the language as an interpreter. One of the most important things that our semantics gives us, though, is a means or reasoning more broadly about the properties of programs written in IMP.

As a basic example, we can use the big-step semantics of IMP to prove that some program transformations preserve the meaning of programs. In class I gave the example that whenever we write a program with a while loop while b do c, σ, I can always safely replace it with an expression that unrolls the loop once: if b then c; (while b do c) else skip.

Why might I make this change? It could be that the latter program will run faster than the former (this semantics doesn't tell us anything about this, but other kinds of semantics can do so). However, it's important to know that this transformation is meaning-preserving. Let's prove that the two expressions are equivalent[2].

[2]In class we proved this slightly using a less formal style. This is a more refined and preferred approach to the proof.

Proposition 2. $\langle \textsf{while}\ b\ \textsf{do}\ c, \sigma\rangle \Downarrow_{\text{COM}} \sigma'$ *iff* $\langle \textsf{if}\ b\ \textsf{then}\ c; (\textsf{while}\ b\ \textsf{do}\ c)\ \textsf{else skip}, \sigma\rangle \Downarrow_{\text{COM}} \sigma'$.

Proof. We prove both directions of the bi-implication separately. Here we only show the forward direction and leave the reverse direction as an exercise for the reader.

Case ($\Rightarrow$ (only if)). Suppose $\langle \textsf{while}\ b\ \textsf{do}\ c, \sigma\rangle \Downarrow_{\text{COM}} \sigma'$. Then by the inversion lemma we have two cases:

Case ($\langle b, \sigma\rangle \Downarrow_{\text{BEXP}} \textsf{true}$). Then $\langle c, \sigma\rangle \Downarrow_{\text{COM}} \sigma''$ and $\langle \textsf{while}\ b\ \textsf{do}\ c, \sigma''\rangle \Downarrow_{\text{COM}} \sigma'$. Really this means that there is some derivation $\mathcal{D}_1 :: \langle b, \sigma\rangle \Downarrow_{\text{BEXP}} \textsf{true}$, and there are two derivations $\mathcal{D}_2$ and $\mathcal{D}_3$, respectively, of the latter statements. We can use these to build the following derivation tree.

$$\dfrac{\dfrac{\mathcal{D}_1}{\langle b, \sigma\rangle \Downarrow_{\text{BEXP}} \textsf{true}} \qquad \dfrac{\dfrac{\mathcal{D}_2}{\langle c, \sigma\rangle \Downarrow_{\text{COM}} \sigma''} \quad \dfrac{\mathcal{D}_3}{\langle \textsf{while}\ b\ \textsf{do}\ c, \sigma''\rangle \Downarrow_{\text{COM}} \sigma'}}{\langle c; (\textsf{while}\ b\ \textsf{do}\ c), \sigma\rangle \Downarrow_{\text{COM}} \sigma'}}{\langle \textsf{if}\ b\ \textsf{then}\ c; (\textsf{while}\ b\ \textsf{do}\ c)\ \textsf{else skip}, \sigma\rangle \Downarrow_{\text{COM}} \sigma'.}$$

Case ($\langle b, \sigma\rangle \Downarrow_{\text{BEXP}} \textsf{false}$). Then $\sigma' = \sigma$, and we can build the following tree:

$$\dfrac{\dfrac{\mathcal{D}_1}{\langle b, \sigma\rangle \Downarrow_{\text{BEXP}} \textsf{false}} \qquad \dfrac{}{\langle \textsf{skip}, \sigma\rangle \Downarrow_{\text{COM}} \sigma}}{\langle \textsf{if}\ b\ \textsf{then}\ c; (\textsf{while}\ b\ \textsf{do}\ c)\ \textsf{else skip}, \sigma\rangle \Downarrow_{\text{COM}} \sigma.}$$

□

Notice how we use the Inversion lemmas to decompose our hypothetical derivation; then once we have the pieces that we want, we build up a new derivation of something else. Many proofs have this overall structure.