CS411 Notes 1: IMP and Large Step Operational Semantics

A. Demers

23-25 Jan 2001

This material is primarily from Ch. 2 of the text. We present an imperative language called **IMP**; we give a formal definition of its syntax and the first of several operational semantic definitions.

1 The Syntax of IMP

Here we present the syntax of an imperative programming language called **IMP**. **IMP** is about the simplest imperative language you can imagine. It has a fixed (possibly infinite) set of integer-valued global variables with assignments, conditional commands and **while**-loops, and that's about all. In particular, it lacks procedures, functions, variable declarations, type declarations, any many other creature comforts you would expect to find in a "real" programming language. Despite its simplicity, **IMP** is Turing-complete, as we shall see later on.

The syntax of **IMP** will be defined by simultaneous definition of the following six *syntactic sets*:

N the (positive and negative) integers.

T the (Boolean) truth values, **true** and **false**.

Loc the set of locations of program variables.

Aexp the set of arithmetic expressions.

Bexp the set of Boolean expressions.

Com the set of commands.

We assume the first three syntactic sets – numbers, Booleans and locations – are already fully specified, and the reader's intuitive understanding will be sufficient. There are just couple of points we should note:

- Technically these are *syntactic* sets, and we probably should not be identifying **N** with the integers. In any real programming language there are multiple literals denoting the same number, e.g. '1', '01', '001', ... To simplify the exposition, we shall initially define this possibility away – we shall assume elements of **N** are canonicalized so there is *exactly one* way to write each (signed) integer.
- The name **Loc** suggests numeric memory addresses, but we can use any set that can uniquely identify memory locations; it will be most convenient to assume **Loc** is an infinite set of program variable names.

For the remaining three syntactic sets – the arithmetic and Boolean expressions, and the program commands (or statements) – we give *formation rules* describing how to build up elements of each syntactic set from smaller elements of the same set, or from elements of the other syntactic sets. In these rules (as well as in the operational semantics to follow) we use simple notational conventions to associate variables with the sets they range over.

N contains $n, n', n'', n_0, n_1, \ldots$

T contains $t, t', t'', t_0, t_1, \ldots$.

Loc the set of locations of program variables.

Aexp contains $a, a', a'', a_0, a_1, \ldots$

Bexp contains $b, b', b'', b_0, b_1, \ldots$

Com contains $c, c', c'', c_0, c_1, \ldots$

Using these conventions, the formation rules for **IMP** are given by the following BNF description:

$$\begin{aligned}
a &::= n \mid X \mid a_0 + a_1 \mid a_0 - a_1 \mid a_0 * a_1 \\
b &::= \mathbf{true} \mid \mathbf{false} \mid a_0 = a_1 \mid a_0 \leq a_1 \mid \neg b_0 \mid b_0 \vee b_1 \mid b_0 \wedge b_1 \\
c &::= \mathbf{skip} \mid X \leftarrow a_0 \mid c_0 ; c_1 \mid \mathbf{if}\ b_0\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1 \mid \mathbf{while}\ b_0\ \mathbf{do}\ c_0
\end{aligned}$$

These rules have the usual interpretation for BNF definitions (inductive definition of the syntactic sets). That is, they define the least sets (under subset ordering, $\subseteq$) containing the basic sets and closed under the formation rules.

2 Large Step Operational Semantics

Here we give the first of several operational semantic definitions we shall discuss for **IMP**.

Our first definition is called a "Large Step" operational semantics. The basic approach is first to define formally the notion of a program execution *configuration* – intuitively, a pair consisting of some program text to be executed and a memory state from which execution is to begin – and then to give a set of logical inference rules that define a *reduction* relation $\rightarrow$ between such configurations and (final) memory states. A configuration is related by $\rightarrow$ to the final state that would result from a terminating program execution starting from that initial configuration.

We begin by defining the states: the set Σ of (memory) *states* consists of the total functions $\sigma : \mathbf{Loc} \rightarrow \mathbf{N}$. Such a function maps each location to a number; intuitively, $\sigma(X)$ gives the value (or contents) of location X in state σ.

We next define a *command configuration* as a pair $\langle c, \sigma \rangle$, where $c \in \mathbf{Com}$ and $\sigma \in \Sigma$.

We lied just a bit in the first paragraph of this section. There really need to be three different (but related) notions of a configuration. **IMP** programs do not just execute commands; they also evaluate arithmetic and Boolean expressions. Thus, we require the additional definitions:

An *arithmetic expression configuration* is a pair $\langle a, \sigma \rangle$, where $a \in \mathbf{Aexp}$ and $\sigma \in \Sigma$.

A *Boolean expression configuration* is a pair $\langle b, \sigma \rangle$, where $b \in \mathbf{Bexp}$ and $\sigma \in \Sigma$.

Technically, there are also three different reduction relations; but since no confusion should result we overload the symbol $\rightarrow$ to represent all three of them:

$$\langle a, \sigma \rangle \rightarrow n$$

holds when evaluation of a starting in state σ produces the number $n \in \mathbf{N}$;

$$\langle b, \sigma \rangle \rightarrow t$$

holds when evaluation of b starting in state σ produces the Boolean value $t \in \mathbf{T}$; and

$$\langle c, \sigma \rangle \rightarrow \sigma'$$

holds when execution of c starting in state σ terminates in state σ'.

Below we give inference rules to define each of these relations.

2.1 Arithmetic Expression Rules

These are the rules for evaluating arithmetic expressions. As noted above, each rule has a conclusion of the form

$$\langle a, \sigma \rangle \rightarrow n$$

which you should interpret to mean expression a in state σ evaluates to n.

Constants

$$\frac{}{\langle n, \sigma\rangle \rightarrow n} \qquad \text{(L1.n)}$$

A number (a numeric literal) evaluates to itself independent of the state.

Variables

$$\frac{}{\langle X, \sigma\rangle \rightarrow \sigma(X)} \qquad \text{(L1.vref)}$$

A variable reference evaluates to the contents of the variable's location in the memory state.

Binary Operations

$$\frac{\langle a_0, \sigma\rangle \rightarrow n_0 \qquad \langle a_1, \sigma\rangle \rightarrow n_1}{\langle a_0 + a_1, \sigma\rangle \rightarrow n} \quad \text{where } n = n_0 + n_1 \qquad \text{(L1.add)}$$

The condition "where $n = n_0 + n_1$" should be interpreted to mean that the rule is applicable only if n_0, n_1 and n are instantiated by numbers whose values satisfy the stated equality. We can interpret the rule to mean that if the two subterms of a sum evaluate respectively to n_0 and n_1, then the sum evaluates to $n_0 + n_1$.

$$\frac{\langle a_0, \sigma\rangle \rightarrow n_0 \qquad \langle a_1, \sigma\rangle \rightarrow n_1}{\langle a_0 - a_1, \sigma\rangle \rightarrow n} \quad \text{where } n = n_0 - n_1 \qquad \text{(L1.sub)}$$

This is the analogous rule for subtraction. Finally,

$$\frac{\langle a_0, \sigma\rangle \rightarrow n_0 \qquad \langle a_1, \sigma\rangle \rightarrow n_1}{\langle a_0 * a_1, \sigma\rangle \rightarrow n} \quad \text{where } n = n_0 * n_1 \qquad \text{(L1.mul)}$$

is the analogous rule for multiplication.

2.2 Boolean Expression Rules

Evaluation rules for Boolean expressions follow a similar pattern.

Boolean Constants

$$\frac{}{\langle \mathbf{true}, \sigma\rangle \rightarrow \mathbf{true}} \qquad \text{(L1.t)}$$

$$\frac{}{\langle \mathbf{false}, \sigma\rangle \rightarrow \mathbf{false}} \qquad \text{(L1.f)}$$

Boolean constants evaluate to themselves without reference to the state.

Atomic Propositions

$$\frac{\langle a_0, \sigma\rangle \to n_0 \qquad \langle a_1, \sigma\rangle \to n_1}{\langle a_0 = a_1, \sigma\rangle \to \mathbf{true}} \quad \text{where } n_0 = n_1 \qquad \text{(L1.eqt)}$$

$$\frac{\langle a_0, \sigma\rangle \to n_0 \qquad \langle a_1, \sigma\rangle \to n_1}{\langle a_0 = a_1, \sigma\rangle \to \mathbf{false}} \quad \text{where } n_0 \neq n_1 \qquad \text{(L1.eqf)}$$

$$\frac{\langle a_0, \sigma\rangle \to n_0 \qquad \langle a_1, \sigma\rangle \to n_1}{\langle a_0 \leq a_1, \sigma\rangle \to \mathbf{true}} \quad \text{where } n_0 \leq n_1 \qquad \text{(L1.leqt)}$$

$$\frac{\langle a_0, \sigma\rangle \to n_0 \qquad \langle a_1, \sigma\rangle \to n_1}{\langle a_0 \leq a_1, \sigma\rangle \to \mathbf{false}} \quad \text{where } n_0 > n_1 \qquad \text{(L1.leqf)}$$

These rules are all straightforward. Note the conclusions are Boolean expression reduction relations, while the hypotheses require proving arithmetic expression reduction relations.

Unary Operations

$$\frac{\langle b, \sigma\rangle \to \mathbf{true}}{\langle \neg b, \sigma\rangle \to \mathbf{false}} \qquad \text{(L1.nott)}$$

$$\frac{\langle b, \sigma\rangle \to \mathbf{false}}{\langle \neg b, \sigma\rangle \to \mathbf{true}} \qquad \text{(L1.notf)}$$

These are the obvious rules relating the value of a unary Boolean expression to the value of its negation.

Binary Operations

$$\frac{\langle b_0, \sigma\rangle \to t_0 \qquad \langle b_1, \sigma\rangle \to t_1}{\langle b_0 \wedge b_1, \sigma\rangle \to t} \quad \text{where } t_0 \wedge t_1 \equiv t \qquad \text{(L1.and)}$$

$$\frac{\langle b_0, \sigma\rangle \to t_0 \qquad \langle b_1, \sigma\rangle \to t_1}{\langle b_0 \vee b_1, \sigma\rangle \to t} \quad \text{where } t_0 \vee t_1 \equiv t \qquad \text{(L1.or)}$$

These rules relate the value of a binary Boolean expression to the values of its subexpressions. Note most "real" programming languages would not evaluate both subexpressions unnecessarily – that is, if the left operand of an $\vee$ reduced to **true** or the left operand of an $\wedge$ reduced to **false**, then the right operand would not be evaluated at all. This technique is sometimes called "short-circuit evaluation," as opposed to the "strict evaluation" performed by our inference rules. In the current version of **IMP** this optimization has no effect on the result of program execution. The value of a Boolean expression, and the result of executing a program containing Boolean expressions, is the same whether strict or short-circuit evaluation is used. Later we will discuss variants of **IMP** in which expression evaluation can modify the store or can fail to terminate. In such variants, the distinction between strict and non-strict evaluation will be significant.

2.3 Command Execution Rules

Before we present rules for command execution, we need some additional notation. Recall the state is a function mapping locations of program variables to their values. Intuitively, assigning a new value to a program variable should have the effect of changing the state function at a single argument point (the location of the affected variable). We express this by the following:

$$\sigma[m/X](Y) = \begin{cases} m & \text{if } Y = X \\ \sigma(Y) & \text{o.w.} \end{cases}$$

That is, $\sigma[m/X]$ is a new state function that is identical to σ except on the argument X, where it returns m rather than $\sigma(X)$.

Null Commands

$$\frac{}{\langle \mathbf{skip}, \sigma\rangle \rightarrow \sigma} \qquad \text{(L1.skip)}$$

The **skip** command has no effect on the state.

Assignments

$$\frac{\langle a, \sigma\rangle \rightarrow n}{\langle X \leftarrow a, \sigma\rangle \rightarrow \sigma[n/X]} \qquad \text{(L1.asgn)}$$

An assignment updates the state at the specified location to the value of the right-hand-side expression.

Sequential Composition

$$\frac{\langle c_0, \sigma\rangle \rightarrow \sigma'' \qquad \langle c_1, \sigma''\rangle \rightarrow \sigma'}{\langle c_0; c_1, \sigma\rangle \rightarrow \sigma'} \qquad \text{(L1.seq)}$$

The execution of the sequential composition $c_0; c_1$ can be understood as the consecutive execution of c_0 and c_1 where the intermediate state σ'' (the result of executing c_0) appears explicitly in the instantiated rule.

Conditionals

$$\frac{\langle b, \sigma\rangle \to \textbf{true} \qquad \langle c_0, \sigma\rangle \to \sigma'}{\langle \textbf{if } b \textbf{ then } c_0 \textbf{ else} c_1, \sigma\rangle \to \sigma'} \tag{L1.ift}$$

Execution of a conditional whose test evaluates to **true** is equivalent to execution of the **then** clause, c_0.

$$\frac{\langle b, \sigma\rangle \to \textbf{false} \qquad \langle c_1, \sigma\rangle \to \sigma'}{\langle \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1, \sigma\rangle \to \sigma'} \tag{L1.iff}$$

Execution of a conditional whose test evaluates to **false** is equivalent to execution of the **else** clause, c_1.

Loops The loop rules are intuitively straightforward.

$$\frac{\langle b, \sigma\rangle \to \textbf{false}}{\langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma} \tag{L1.whf}$$

A loop whose condition evaluates to **false** terminates immediately with no effect on the store.

$$\frac{\langle b, \sigma\rangle \to \textbf{true} \qquad \langle c, \sigma\rangle \to \sigma'' \qquad \langle \textbf{while } b \textbf{ do } c, \sigma''\rangle \to \sigma'}{\langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma'} \tag{L1.wht}$$

Execution of a loop whose condition evaluates to **true** is equivalent to "unrolling" the loop once – that is, executing the loop body c and then *re-executing* the entire loop starting from the new state σ'' in which the first execution of c terminated.

One aspect of this rule is new, and has important consequences. Note that the *entire* **while**- loop

$$\textbf{while } b \textbf{ do } c$$

from the rule's conclusion appears in the third hypothesis. You should verify that this is new behavior in the following sense: in *every* other rule, the program pieces appearing in hypothesis configurations are subparts of the program piece in the conclusion configuration, and thus are *strictly smaller*.

Without **while** loops and the **while** rule, you could exploit this diminishing-size property to compute an upper bound on the size of any possible proof of a given execution relation. That is, you could come up with an easy-to-compute function f such that no proof tree for $\langle c, \sigma\rangle \to \sigma'$ could possibly require more than $f(|c|)$ rule instances. In principle, you could write a computer program to enumerate all proof trees up to that size and check each tree against the desired conclusion $\langle c, \sigma\rangle \to \sigma'$. Such a program would check termination, which is not possible for a Turing-complete language. Of course, without **while** commands

IMP is *not* Turing-complete, and all loop-free **IMP** *do* terminate, so there is no contradiction in any of this.

With **while** loops, there is no recursive bound on the size of the proof tree required to prove a terminating program execution. Any algorithm to construct a proof tree would be guaranteed, on at least some nonterminating programs and inputs, to loop forever, trying to construct an "infinite proof tree."