CS611 Lecture 2 — The Large-Step Semantics of IMP — 8/27/00

Scribe: Daniel Marques and Vicky Weissman — Lecturer: Andrew Myers

1 IMP in further details

1.1 A Review of IMP's Syntax

Given n ∈ integers, X ∈ **Loc**, and **op** ∈ {+, -, *},

$$\begin{array}{l} a ::= n \mid X \mid a_0\ \mathbf{op}\ a_1 \\ b ::= \mathbf{true} \mid \mathbf{false} \mid \neg b \mid b_0 \wedge b_1 \mid b_0 \vee b_1 \\ c ::= \mathbf{skip} \mid X := a \mid \mathbf{if}\, b\, \mathbf{then}\, c_0\, \mathbf{else}\, c_1 \mid \mathbf{while}\, b\, \mathbf{do}\, c \mid c_0; c_1 \end{array}$$

1.2 Evaluation Rules

Evaluation rules can be one of three types:

$\langle a, \sigma\rangle \Downarrow n$, where n is an integer
$\langle b, \sigma\rangle \Downarrow t$, where t is a truth value
$\langle c, \sigma\rangle \Downarrow \sigma'$

The evaluation rules for arithmetic expressions are :

$\langle n, \sigma\rangle \Downarrow n$
$\langle X, \sigma\rangle \Downarrow \sigma(X)$

$$\frac{\langle a_0, \sigma\rangle \Downarrow n_0 \quad \langle a_1, \sigma\rangle \Downarrow n_1}{\langle a_0\ \mathbf{op}\ a_1, \sigma\rangle \Downarrow n} \qquad \text{where } n = n_0 \text{ op } n_1$$

The evaluation rules for boolean expressions are:

$\langle \mathbf{true}, \sigma\rangle \Downarrow \text{true}$
$\langle \mathbf{false}, \sigma\rangle \Downarrow \text{false}$

$$\frac{\langle b, \sigma\rangle \Downarrow t}{\langle \neg b, \sigma\rangle \Downarrow t'} \qquad \text{where } t' = \bar{t} \text{ (mathematical negation)}$$

$$\frac{\langle b_0, \sigma\rangle \Downarrow \mathbf{true} \qquad \langle b_1, \sigma\rangle \Downarrow \mathbf{true}}{\langle b_0 \wedge b_1, \sigma\rangle \Downarrow \mathbf{true}}$$

With the **false** case, it's more complicated, because the following two rules do not specify an order of evaluation:

$$\frac{\langle b_0, \sigma\rangle \Downarrow \mathbf{false}}{\langle b_0 \wedge b_1, \sigma\rangle \Downarrow \mathbf{false}} \qquad \frac{\langle b_1, \sigma\rangle \Downarrow \mathbf{false}}{\langle b_0 \wedge b_1, \sigma\rangle \Downarrow \mathbf{false}}$$

To implement ∧ in a *short-circuit* manner, we use the following set of rules:

$$\frac{\langle b_0, \sigma\rangle \Downarrow \mathbf{false}}{\langle b_0 \wedge b_1, \sigma\rangle \Downarrow \mathbf{false}} \qquad \frac{\langle b_0, \sigma\rangle \Downarrow \mathbf{true} \qquad \langle b_1, \sigma\rangle \Downarrow \mathbf{false}}{\langle b_0 \wedge b_1, \sigma\rangle \Downarrow \mathbf{false}}$$

In IMP, these alternative sets of rules define the same semantics, because boolean expressions do not have side effects. In most imperative languages, however, the rules would define different semantics.

The evaluation rules for IMP's commands are:

$$\frac{}{\langle \mathbf{skip},\, \sigma\rangle \Downarrow \sigma} \qquad \frac{\langle a,\, \sigma\rangle \Downarrow n}{\langle X := a,\, \sigma\rangle \Downarrow \sigma\,[X \mapsto n]} \qquad \frac{\langle c_0,\, \sigma\rangle \Downarrow \sigma'' \qquad \langle c_1,\, \sigma''\rangle \Downarrow \sigma'}{\langle c_0; c_1,\, \sigma\rangle \Downarrow \sigma'}$$

$$\frac{\langle b,\, \sigma\rangle \Downarrow \mathbf{true} \qquad \langle c_0,\, \sigma\rangle \Downarrow \sigma'}{\langle \mathbf{if}\, b\, \mathbf{then}\, c_0\, \mathbf{else}\, c_1,\, \sigma\rangle \Downarrow \sigma'} \qquad \frac{\langle b,\, \sigma\rangle \Downarrow \mathbf{false} \qquad \langle c_1,\, \sigma\rangle \Downarrow \sigma'}{\langle \mathbf{if}\, b\, \mathbf{then}\, c_0\, \mathbf{else}\, c_1,\, \sigma\rangle \Downarrow \sigma'}$$

$$\frac{\langle b,\, \sigma\rangle \Downarrow \mathbf{false}}{\langle \mathbf{while}\, b\, \mathbf{do}\, c,\, \sigma\rangle \Downarrow \sigma} \qquad \frac{\langle b,\, \sigma\rangle \Downarrow \mathbf{true} \qquad \langle c; \mathbf{while}\, b\, \mathbf{do}\, c,\, \sigma\rangle \Downarrow \sigma'}{\langle \mathbf{while}\, b\, \mathbf{do}\, c,\, \sigma\rangle \Downarrow \sigma'}$$

2 Using Semantics to Prove Properties of IMP

2.1 The Notion of a Proof

We have been attempting to define when $\langle c,\, \sigma\rangle \Downarrow \sigma'$ holds true, i.e. all cases where a command c, executed in σ, results in σ'. We can view this configuration as the triple, (c, σ, σ'). We would then like to determine if a particular triple (or subset of the set of all such triples) *follows* from the rules of our language. If we can prove that, we know that c executed in σ will indeed result in σ'.

2.2 Using a Proof Tree

In section 1.2, we described all the evaluation rules used to determine the behavior of IMP programs. We will now use those rules to prove that specific commands will have specific results.

Applying a rule to a specific case results in a **rule instance** — we substitute actual values (or variables) for the meta-variables in the rule. For each of the premises, p, of our rule instance, we attempt to make another rule instance where the conclusion of this new instance is p. We continue to do this until our premises are axioms of the language. If we are successful, the resulting application of rules is called a **proof tree**.

For example, given the rule:

$$\frac{\langle b,\, \sigma\rangle \Downarrow t}{\langle \neg b,\, \sigma\rangle \Downarrow t'} \qquad \text{where } t' = \bar{t} \text{ (mathematical negation)}$$

we can construct the two rule-instances:

$$\frac{\langle \mathbf{false},\, \sigma\rangle \Downarrow \mathbf{false}}{\langle \neg\mathbf{false},\, \sigma\rangle \Downarrow \mathbf{true}} \quad \text{and} \quad \frac{\langle \mathbf{false},\, \sigma\rangle \Downarrow \mathbf{true}}{\langle \neg\mathbf{false},\, \sigma\rangle \Downarrow \mathbf{false}}.$$

The first one is a proof of program behavior, because its premise is an axiom. The second is a valid rule instance, but it is not part of any proof, because no rule (or chain of rules) could have its meta-variables substituted for and conclude with $\langle \mathbf{false},\, \sigma\rangle \Downarrow \mathbf{true}$.

2.3 Execution As Proof

An execution of a command in a state, $\langle c,\, \sigma\rangle \Downarrow \sigma'$ is legal only if we can prove that behavior. Because our method of proof is a proof tree where every step is the application of an inference rule, a legal execution is

a depth-first walk of a proof tree.
In the following proof tree, these substitutions of actual variables for meta-variables have been made:

$b \leftarrow x < y$
$c_0 \leftarrow x := 0$
$c_1 \leftarrow \textbf{skip}$
$\sigma \leftarrow [x \mapsto 1, y \mapsto 2]$
$\sigma' \leftarrow [x \mapsto 0, y \mapsto 2]$

$$\dfrac{\dfrac{\langle x,\, [x \mapsto 1,\, \ldots]\rangle \Downarrow 1 \quad \langle y,\, [\ldots,\, y \mapsto 2]\rangle \Downarrow 2}{\langle x < y,\, [x \mapsto 1, y \mapsto 2]\rangle \Downarrow \textbf{true}} \qquad \dfrac{\langle 0,\, [x \mapsto 1,\, \ldots]\rangle \Downarrow 0}{\langle x := 0,\, [x \mapsto 1,\, \ldots]\rangle \Downarrow [x \mapsto 0, y \mapsto 2]}}{\langle \textbf{if}\, x < y\, \textbf{then}\, x := 0\, \textbf{else}\, \textbf{skip},\, [x \mapsto 1, y \mapsto 2]\rangle \Downarrow [x \mapsto 0, y \mapsto 2]}$$

2.4 Non-terminating Programs

A depth-first walk down the proof tree corresponds to an interpreter running our IMP program. As long as it is always able to choose the right rule, given the current configuration, our interpreter will behave as expected. Of course, it may never terminate.

Example: prove that the program **while true do skip** will never terminate.

Proof by contradiction.
Assume $\exists \sigma'$ s.t. $\langle \textbf{while true do skip},\, \sigma\rangle \Downarrow \sigma'$
Then we should be able to construct a proof tree leading to that configuration:

$$\dfrac{\langle \textbf{true},\, \sigma\rangle \Downarrow \textbf{true} \qquad \langle \textbf{skip},\, \sigma\rangle \Downarrow \sigma \qquad \dfrac{\text{Repeat of Proof Tree}}{\langle \textbf{while true do skip},\, \sigma\rangle \Downarrow \sigma'}}{\langle \textbf{while true do skip},\, \sigma\rangle \Downarrow \sigma'}$$

The proof tree contains a copy of itself. This is not possible, because the proof tree is necessarily finite. Because any depth-first walk of this proof tree will never terminate, an interpreter using this proof will never terminate. Therefore, $\nexists\, \sigma'$ s.t. $\langle \textbf{while true do skip},\, \sigma\rangle \Downarrow \sigma'$.

2.5 Errors

Lets try to add the $\div$ operator to IMP with the following rule:

$$\dfrac{\langle a_0,\, \sigma\rangle \Downarrow n_0 \qquad \langle a_1,\, \sigma\rangle \Downarrow n_1}{\langle a_0 \div a_1,\, \sigma\rangle \Downarrow n} \qquad \text{where n} = \lfloor n_0/n_1 \rfloor.$$

Applying this rule to the case when a$_1$ evaluates to 0, (e.g. $\langle 2 \div 0,\, \sigma\rangle \Downarrow n$), the interpreter will try to find n s.t. $\lfloor n_0/0 \rfloor =$ n. The interpreter will never find an evaluation of an expression containing division by zero, because there is no proof tree of the form:

$$\dfrac{\dfrac{\vdots}{\langle a_0,\, \sigma\rangle \Downarrow n_0} \qquad \dfrac{\vdots}{\langle a_1,\, \sigma\rangle \Downarrow 0}}{\langle a_0 \div a_1,\, \sigma\rangle \Downarrow n}$$

By looking solely at the set of legal evaluations, we cannot distinguish this behavior from the infinite loop in section 2.4.

We could introduce an extra "state" to store and propagate errors: $\langle a, \sigma\rangle \Downarrow (\sigma, \textbf{error})$ but this would complicate our operational semantics. Although we will not add division to IMP now, we will discuss small-step semantics which deals with error conditions.

3 Using Induction to Prove Characteristics of IMP

An examination of the semantics of IMP tell us that the language is deterministic.
Formally, $\forall c, \sigma, \sigma'' : \langle c, \sigma\rangle \Downarrow \sigma' \wedge \langle c, \sigma\rangle \Downarrow \sigma'' \Leftrightarrow \sigma' = \sigma''$
We could prove this by using induction on the size of a proof tree. This will be discussed further later in the course.

4 Non-determinism and Parallelism

4.1 Dijkstra's Non-deterministic Choice Operator

Let $c_0 \,\square\, c_1$ denote an application of Dijkstra's non-deterministic choice operator to the two operands c_0 and c_1. If both commands will terminate, then the operator can choose either. If only one command will terminate, then the operator chooses that one.

Rules for such a command would be:

$$\frac{\langle c_0, \sigma\rangle \Downarrow \sigma'}{\langle c_0 \,\square\, c_1, \sigma\rangle \Downarrow \sigma'} \quad \text{and} \quad \frac{\langle c_1, \sigma\rangle \Downarrow \sigma'}{\langle c_0 \,\square\, c_1, \sigma\rangle \Downarrow \sigma'}$$

An interpreter with this behavior must always make the right choice between executing c_0 or c_1. In practice, this requires both commands to be executed. The choice operator illustrates the fact that it is easy to write large-step semantics which correspond to a language that is difficult to implement.

4.2 Parallelism and the **cobegin** Command

Let the configuration $\langle \textbf{cobegin}\, c_0 c_1, \sigma\rangle \Downarrow \sigma'$ denote running two commands c_0, c_1 in parallel, keeping the side effects of each. If we try to write rules to specify this behavior, for example,

$$\frac{\langle c_0, \sigma\rangle \Downarrow \sigma'' \cdots}{\langle \textbf{cobegin}\, c_0 c_1, \sigma\rangle \Downarrow \sigma'}$$

we realize that we cannot describe the possible interleaving of the execution of c_0 that of c_1. Large-step semantics are not well-suited to describing parallel executions. Small-step semantics, however, will allow us to discuss individual steps of computation performed during command execution.

5 Small Step Semantics

The examples in section 4 highlight the limits of the **large step semantics** that we have been using to describe the behavior of our IMP programs. Large step semantics (also called natural semantics) are limited to showing the *result* of executing a certain command — they cannot show the steps taken during that execution. For that, we introduce a new system called **small step semantics** which will be discussed in the next lecture.

Briefly, small step semantics describe what happens during each step of the execution. For example, $\langle c, \sigma\rangle \to \langle c', \sigma'\rangle$, shows us that after the first step of executing c in σ we arrive at a new configuration, one where we are about to execute c' in state σ'.