CS411 Notes 2 – IMP Small Step Semantics

A. Demers

30 Jan 2001

1 Large Step vs. Small Step Semantics

The operational semantics for **IMP** presented in the previous notes is a "large step" or "natural" semantics. It has a notion of a "state," which for **IMP** gives values to all program variables, and a notion of a "configuration," which for **IMP** is a pair consisting of a piece of program text to be executed and a program state in which to begin execution. The semantics is presented as a collection of proof rules, whose conclusions take the form

$$\langle \text{configuration} \rangle \ \rightarrow \ \langle \text{state} \rangle$$

Each such conclusion should be thought of as describing a complete program execution, that is, relating an initial configuration to the final state that results from executing the program to completion. In particular, the "theorems" of the system – that is, the conclusions of its proofs, or equivalently the $\rightarrow$ relation it defines – make no mention of the intermediate states in a computation.

Insensitivity to intermediate states is arguably a strength in that it makes the semantics more "abstract." The semantics defines *what* result a program computes, without constraining inessential details about *how* the result is computed.

Consider the (only reasonable) definition of program equivalence using large step semantics:

$$(c_0 \equiv c_1) \ \Leftrightarrow \ \forall \sigma, \sigma' (\langle c_0, \sigma \rangle \sigma' \ \Leftrightarrow \ \langle c_1, \sigma \rangle \sigma')$$

What programs are equivalent under this definition?

All nonterminating programs are equivalent.

For any set S of program variables, a pair of programs c_0 and c_1 that assign to no variables outside of S and that clear all the variables of S to (say) 0 before terminating are equivalent, *independent of the sequence of intermediate states they produce.*

There are circumstances under which the complete-execution definitions provided by large step semantics, and the resulting liberal notion of program equivalence, can be undesirable. Perhaps the best-motivated one is concurrency. Imagine extending **IMP** by adding a parallel-execution primitive

$$c \ ::= \ c_0 \ // \ c_1$$

with the intended meaning that c_0 and c_1 are to execute concurrently, possibly interfering with one another through their effects on shared (global) variables (recall *all* program variables in **IMP** are global).

What happens when we attempt to write a large step rule for this construct? The conclusion naturally looks like

$$\frac{\ldots}{\langle c_0 \ // \ c_1, \sigma \rangle \rightarrow \sigma'}$$

What can the hypotheses be? A large step style suggests something of the form

$$\frac{\ldots \ \langle c_0, \sigma_{00} \rangle \rightarrow \sigma_{01} \ \ \ldots \ \langle c_1, \sigma_{10} \rangle \rightarrow \sigma_{11} \ \ \ldots}{\langle c_0 \ // \ c_1, \sigma \rangle \rightarrow \sigma'}$$

A bit of thought should convince you that this *cannot work*. The hypotheses describe the behavior of complete executions of c_0 and c_1 run *in isolation*. The first assignment to a global variable referenced by both c_0 and c_1 invalidates the "isolation" assumption and renders the proofs of hypotheses useless for describing the behavior of $c_0 \ // \ c_1$.

An alternative way to arrive at this pessimistic conclusion is to ask the question

> Suppose c_0 and c_0' are equivalent programs. Shouldn't I be able to conclude, for any other program c_1, that $c_0 \ // \ c_1$ and $c_0' \ // \ c_1$ are equivalent?

Certainly you'd like to be able to conclude that, and clearly you can't, since c_0 and c_0' could produce pretty arbitrarily different sequences of intermediate states during their execution, and thus could interfere with the concurrent execution of c_1 in pretty arbitrarily different ways.

To summarize all this, we have a problem that – at least for some purposes – our large step semantic definition is "too abstract."

One approach to dealing with this problem is not to give proof rules for the complete-execution relation $\rightarrow$, but instead to axiomatize a new *one-step* relation $\rightarrow_1$ that defines the individual atomic steps of program execution rather

than execution to completion. This "small step" approach is the one we shall develop here.

The first thing to notice is that an individual "atomic step" does not take execution to completion, and this has an effect on the type of the execution relation. Unlike $\rightarrow$, which relates configurations to final states or result values, the one-step relation $\rightarrow_1$ relates configurations to other configurations – it relates the "current" configuration to the "next" configuration in a computation. Final results of computations are defined by identifying a subset of the configurations that are "irreducible" in the sense of allowing no further computation steps. To define a complete computation, the $\rightarrow_1$ relation is applied repeatedly until an irreducible configuration is reached.

Configurations are defined almost exactly as they were for large step semantics:

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

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

A *command configuration* is a pair $\langle c, \sigma \rangle$, where $c \in \mathbf{Com}$ and $\sigma \in \Sigma$. As a special case, we allow configurations of the form $\langle , \sigma \rangle$, in which the command part is empty. A command configuration is *irreducible* if its command part is empty.

> Actually, the definition of "irreducible" above is slightly fraudulent. The *true* definition of irreducible is the set of configurations that cannot occur on the left hand side of $\rightarrow_1$ in the conclusion of any proof. Of course this definition depends on the particular set of rules. After reading the small step rules presented below, you might try to convince yourself that the two definitions are equivalent for these rules.

Our small step semantics consists of proof rules defining the one-step relation $\rightarrow_1$ between configurations. Results of complete program executions will be defined by considering the (reflexive and) transitive closure $\rightarrow_1^*$ and restricting that relation to irreducible configurations on the right hand side. The large and small step definitions of **IMP** are equivalent in the sense that

$$\langle c, \sigma \rangle \rightarrow \sigma' \qquad \Leftrightarrow \qquad \langle c, \sigma \rangle \rightarrow_1^* \langle , \sigma' \rangle$$

As we shall see, this fact is not at all trivial to prove.

2 A Small Step Definition of IMP

Here is our complete small step semantics for **IMP**. This is still the original **IMP** language as described in Chapter 2 of the text. In particular, expression evaluation always terminates and has no side effects, and we have not (yet) added any concurrency.

2.1 Arithmetic Expressions

Variable reference

$$\frac{}{\langle X, \sigma\rangle \to_1 \langle n, \sigma\rangle} \qquad \text{where } n = \sigma(X) \tag{S1.vref}$$

A variable reference evaluates in a single step to the contents of the state at the variable's location.

Binary Terms The following rules apply for any binary operator θ.

$$\frac{\langle a_0, \sigma\rangle \to_1 \langle a_0', \sigma'\rangle}{\langle a_0 \theta a_1, \sigma\rangle \to_1 \langle a_0' \theta a_1, \sigma'\rangle} \tag{S1.aopl}$$

Apply a single-step transition to the left operand if possible.

$$\frac{\langle a_1, \sigma\rangle \to_1 \langle a_1', \sigma'\rangle}{\langle n_0 \theta a_1, \sigma\rangle \to_1 \langle n_0 \theta a_1', \sigma'\rangle} \tag{S1.aopr}$$

If the left operand is irreducible, apply a single-step transition to the right operand if possible.

$$\frac{}{\langle n_0 \theta n_1, \sigma\rangle \to_1 \langle n, \sigma\rangle} \qquad \text{where } n = n_0 \theta n_1 \tag{S1.aopn}$$

If both terms are irreducible, produce the correct answer in a single step.

2.2 Boolean Expressions

Arithmetic Comparison The following rules apply for any comparison operator θ from among $=$, $\neq$, $<$, $\leq$, ...

$$\frac{\langle a_0, \sigma\rangle \to_1 \langle a_0', \sigma'\rangle}{\langle a_0 \theta a_1, \sigma\rangle \to_1 \langle a_0' \theta a_1, \sigma'\rangle} \tag{S1.cmpl}$$

Apply a single-step transition to the left operand if possible.

$$\frac{\langle a_1, \sigma\rangle \to_1 \langle a_1', \sigma'\rangle}{\langle n_0 \theta a_1, \sigma\rangle \to_1 \langle n_0 \theta a_1', \sigma'\rangle} \tag{S1.cmpr}$$

If the left operand is irreducible, apply a single-step transition to the right operand if possible.

$$\frac{}{\langle n_0 \theta n_1, \sigma\rangle \to_1 \langle \mathbf{true}, \sigma\rangle} \quad \text{where } n_0\theta n_1 \text{ is true} \tag{S1.cmpt}$$

$$\frac{}{\langle n_0 \theta n_1, \sigma\rangle \to_1 \langle \mathbf{false}, \sigma\rangle} \quad \text{where } n_0\theta n_1 \text{ is false} \tag{S1.cmpf}$$

If both terms are irreducible, produce the correct answer in a single step.

Unary Boolean Expressions Logical negation is the only such expression in IMP.

$$\frac{\langle b_0, \sigma\rangle \to_1 \langle b_0', \sigma'\rangle}{\langle \neg b_0, \sigma\rangle \to_1 \langle \neg b_0', \sigma'\rangle} \tag{S1.not}$$

Apply a single-step transition to the operand of a negation if possible.

$$\frac{}{\langle \neg \mathbf{true}, \sigma\rangle \to_1 \langle \mathbf{false}, \sigma\rangle} \tag{S1.notf}$$

$$\frac{}{\langle \neg \mathbf{false}, \sigma\rangle \to_1 \langle \mathbf{true}, \sigma\rangle} \tag{S1.nott}$$

If the operand of a negation is irreducible, produce the correct result in a single step.

Binary Boolean Expressions The following rules describe left-to-right, non-strict evaluation of binary Boolean expressions. In particular, if the left operand of $\wedge$ evaluates to **false**, or the left operand of $\vee$ evaluates to **true**, then the correct answer is produced in a single step without evaluating the right operand. The possibility of skipping evaluation of right operands has no effect on the result of Boolean expression evaluation in IMP, because expression evaluation in IMP is guaranteed to terminate. So the worst thing that happens is we evaluate a right subexpression and immediately discard its value, effectively wasting the time we spent computing it. We can never compute an incorrect answer, nor can we fail to compute an answer. This situation will change later, when we add side-effects and nontermination to expression evaluation.

$$\frac{\langle b_0, \sigma\rangle \to_1 \langle b_0', \sigma'\rangle}{\langle b_0 \phi b_1, \sigma\rangle \to_1 \langle b_0' \phi b_1, \sigma'\rangle} \tag{S1.bopl}$$

Apply a single-step transition to the left operand if possible.

$$\frac{}{\langle \mathbf{false} \wedge b_1, \sigma\rangle \to_1 \langle \mathbf{false}, \sigma\rangle} \tag{S1.andf}$$

If the left operand of $\wedge$ is **false**, produce the answer **false** in a single step independent of the right operand.

$$\frac{}{\langle \mathbf{true} \wedge b_1, \sigma\rangle \to_1 \langle b_1, \sigma\rangle} \quad \text{(S1.andt)}$$

If the left operand of $\wedge$ is **true**, discard it. The final answer will be the result of evaluating the right operand.

$$\frac{}{\langle \mathbf{true} \vee b_1, \sigma\rangle \to_1 \langle \mathbf{true}, \sigma\rangle} \quad \text{(S1.ort)}$$

If the left operand of $\vee$ is **true**, produce the answer **true** in a single step independent of the right operand.

$$\frac{}{\langle \mathbf{false} \vee b_1, \sigma\rangle \to_1 \langle b_1, \sigma\rangle} \quad \text{(S1.orf)}$$

If the left operand of $\vee$ is **false**, discard it. The final answer will be the result of evaluating the right operand.

2.3 Commands

Null Command

$$\frac{}{\langle \mathbf{skip}, \sigma\rangle \to_1 \langle , \sigma\rangle} \quad \text{(S1.skip)}$$

As usual, **skip** has no effect; it reduces to a configuration with an empty command part.

Assignment

$$\frac{\langle a_0, \sigma\rangle \to_1 \langle a_0', \sigma'\rangle}{\langle X \leftarrow a_0, \sigma\rangle \to_1 \langle X \leftarrow a_0', \sigma'\rangle} \quad \text{(S1.asgn)}$$

Apply a single-step transition to the assignment right-hand-side expression if possible.

$$\frac{}{\langle X \leftarrow n, \sigma\rangle \to_1 \langle , \sigma[n/X]\rangle} \quad \text{(S1.asgn0)}$$

If the assignment right-hand-side expression is irreducible, update the state appropriately in a single step.

Sequencing These rules actually require careful thought.

$$\frac{\langle c_0, \sigma\rangle \to_1 \langle c_0', \sigma'\rangle}{\langle c_0; c_1, \sigma\rangle \to_1 \langle c_0'; c_1, \sigma'\rangle} \quad \text{(S1.seql)}$$

Execute a single step of the first command c_0

$$\frac{\langle c_0, \sigma\rangle \to_1 \langle , \sigma'\rangle}{\langle c_0; c_1, \sigma\rangle \to_1 \langle c_1, \sigma'\rangle} \quad \text{(S1.seqr)}$$

If the first command c_0 completes in a single step, discard it and continue using the updated state σ'. The final state will be the the one resulting from the second command c_1.

Conditionals

$$\frac{\langle b_0, \sigma\rangle \to_1 \langle b_0', \sigma'\rangle}{\langle \textbf{if } b_0 \textbf{ then } c_0 \textbf{ else } c_1, \sigma\rangle \to_1 \langle \textbf{if } b_0' \textbf{ then } c_0 \textbf{ else } c_1, \sigma'\rangle} \quad \text{(S1.if)}$$

Apply a single-step transition to the condition if possible.

$$\frac{}{\langle \textbf{if true then } c_0 \textbf{ else } c_1, \sigma\rangle \to_1 \langle c_0, \sigma\rangle} \quad \text{(S1.ift)}$$

Execution of a conditional with a condition value of **true** is equivalent to executing the **then** part c_0.

$$\frac{}{\langle \textbf{if false then } c_0 \textbf{ else } c_1, \sigma\rangle \to_1 \langle c_1, \sigma\rangle} \quad \text{(S1.iff)}$$

Execution of a conditional with a condition value of **false** is equivalent to executing the **else** part c_1.

Iteration Just as in the large step semantics, the rule for iteration is the heavy one. Here we use only a single rule, allowing most of the real work to be done by the conditional rules given above.

$$\frac{}{\langle \textbf{while } b_0 \textbf{ do } c_0, \sigma\rangle \to_1 \langle \textbf{if } b_0 \textbf{ then } c_0; \textbf{ while } b_0 \textbf{ do } c_0 \textbf{ else skip}, \sigma\rangle} \quad \text{(S1.wh)}$$

A loop is equivalent to the version that has been "unrolled" once.

This is the **while** rule, and **while** is the only source of nontermination. Therefore, just as in our previous large step semantics, the potential for an "infinite proof tree" must lie in this rule.

Recall in the large step semantics, the potential for an infinite proof tree arose because one of the **while** rules had a hypothesis that was no smaller than the conclusion. Intuitively, this rule could be used infinitely many times without making progress.

In the current small step semantics, the behavior is slightly different. There is only one **while** rule, and it has *no* hypotheses. Here the growth to an infinite

proof tree happens because the conclusion of the rule is an instance of $\rightarrow_1$ in which a complete copy of the left-hand-ide command is embedded in the right-hand-side command.

I should mention that the **while** rule is the only one can lead to infinite proof trees, in either the small step or large step semantics for **IMP**. But that is an artifact of the simplicity of the language. Other constructs such as function definitions would introduce similar behavior.