CS611 Lecture 4 **Large-Step Operational Semantics (II)** September 5, 2003

Scribe: Thomas Finley, Pavel Dmitriev Lecturer: Radu Rugina

1 Proof Tree Properties

In a proof tree, nodes are instances of the operational semantic rules that have been defined, the leaves are axioms, and the root is the original command. We have a program's meaning in the form of a proof tree, but how do we describe its execution? For example, we have the following proof tree for the command $x := x * (x + 1)$.

$$\cfrac{\cfrac{\cfrac{}{\langle x, \sigma\rangle \to 1} \quad \cfrac{\cfrac{}{\langle x, \sigma\rangle \to 1} \quad \cfrac{}{\langle 1, \sigma\rangle \to 1}}{\langle x+1, \sigma\rangle \to 2}}{\langle x*(x+1), \sigma\rangle \to 2}}{\langle x := x*(x+1), \sigma\rangle \to \sigma[x \to 2]}$$

The execution of the command is the depth-first traversal of the proof tree.

2 Program Properties

2.1 Equivalence

Informally, we can describe two commands c_0, c_1 as equivalent, written $c_0 \sim c_1$, if for any initial state σ, they either both don't terminate or terminate in the same final state σ'. Similarly, two expressions are equivalent, $a_0 \sim a_1$ or $b_0 \sim b_1$, if the evaluation of both expressions either doesn't terminate, or it terminates and yields the same value. Formally:

$$\begin{aligned} c_0 \sim c_1 \quad &\Leftrightarrow \quad (\forall \sigma, \sigma' \;.\; \langle c_0, \sigma\rangle \to \sigma' \Leftrightarrow \langle c_1, \sigma\rangle \to \sigma') \\ a_0 \sim a_1 \quad &\Leftrightarrow \quad (\forall \sigma, n \;.\; \langle a_0, \sigma\rangle \to n \Leftrightarrow \langle a_1, \sigma\rangle \to n) \\ b_0 \sim b_1 \quad &\Leftrightarrow \quad (\forall \sigma, t \;.\; \langle b_0, \sigma\rangle \to t \Leftrightarrow \langle b_1, \sigma\rangle \to t) \end{aligned}$$

Note that the following definition of equivalence of commands is wrong:

$$c_0 \sim c_1 \quad \Leftrightarrow \quad (\forall \sigma, \sigma', \sigma'' \;.\; (\langle c_0, \sigma\rangle \to \sigma' \wedge \langle c_1, \sigma\rangle \to \sigma'') \Rightarrow (\sigma' = \sigma''))$$

The problem with this definition is that it does not cover the case where one or the other of c_0 or c_1 does not terminate. Suppose that c_0 does not terminate on σ but that c_1 does. Then the left side of the precondition becomes false since there is no $\langle c_0, \sigma\rangle \to \sigma'$. This means that the $\Rightarrow$ relation is true, which then implies that $c_0 \sim c_1$. Therefore, with this definition, a command that doesn't terminate is always equivalent to any command that terminates, which is not the meaning that we want for equivalence.

2.2 Termination

The execution of the command c in state σ terminates iff $\exists\ \sigma' \;.\; \langle c, \sigma\rangle \to \sigma'$. Similarly, an arithmetic expression a terminates iff $\exists\ n \;.\; \langle a, \sigma\rangle \to n$. A boolean expression b terminates if $\exists\ t \;.\; \langle b, \sigma\rangle \to t$.

2.3 Determinism

The execution of command c is deterministic if all evaluations starting with the same input state yield the same final state. Formally:

$$\forall \sigma, \sigma', \sigma'' \;.\; (\langle c, \sigma\rangle \to \sigma' \wedge \langle c, \sigma\rangle \to \sigma'') \Rightarrow (\sigma' = \sigma'')$$

Similarly, for arithmetic expressions, an arithmetic expression a is deterministic iff:

$$\forall \sigma, n, n' \;.\; (\langle a, \sigma \rangle \to n \wedge \langle a, \sigma \rangle \to n') \Rightarrow (n = n')$$

The definition for determinism of boolean expressions is similar.

Example 1 (equivalence)

As an example, we are going to prove that two commands are equivalent. Specifically, we want to prove the following:

$$\underbrace{\mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1}_{c_{if}} \;\sim\; \underbrace{\mathbf{if}\ \neg b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_0}_{c'_{if}}$$

Let $\sigma, \sigma' \in \Sigma$ be any two states. We want to prove that $\langle c_{if}, \sigma \rangle \to \sigma' \Leftrightarrow \langle c'_{if}, \sigma \rangle \to \sigma'$. This requires to prove the two implications of the equivalence. We will prove the implication "$\Rightarrow$" (the implication "$\Leftarrow$" is very similar and is left as an exercise)

Since $\langle c_{if}, \sigma \rangle \to \sigma'$, there must be a finite proof tree for this this evaluation. By examining all the rules which have an if command as the conclusion, the proof tree must be of one of the following forms:

$$\frac{\dfrac{\vdots(1)}{\langle b, \sigma \rangle \to \mathit{true}} \quad \dfrac{\vdots(2)}{\langle c_0, \sigma \rangle \to \sigma'}}{\langle c_{if}, \sigma \rangle \to \sigma'} \qquad \text{or} \qquad \frac{\langle b, \sigma \rangle \to \mathit{false} \quad \langle c_1, \sigma \rangle \to \sigma'}{\langle c_{if}, \sigma \rangle \to \sigma'}$$

We are going to examine only the subcase when the proof tree is the one on the left, where the condition b evaluates to *true* (the right subcase is similar, and is left as an exercise). We can use the subtrees (1) and (2) to build a new proof tree for $\langle c'_{if}, \sigma \rangle \to \sigma'$:

$$\frac{\dfrac{\dfrac{\vdots(1)}{\langle b, \sigma \rangle \to \mathit{true}}}{\langle \neg b, \sigma \rangle \to \mathit{false}} \; \begin{array}{c}\vdots(2) \\ \langle c_0, \sigma \rangle \to \sigma'\end{array}}{\langle c'_{if}, \sigma \rangle \to \sigma'}$$

This proves that indeed the relation $\langle c'_{if}, \sigma \rangle \to \sigma'$ holds, which completes this subcase. We can then similarly prove the other subcase (when b evaluates to *false*) and the other implication ("$\Leftarrow$"), to complete the proof.

Example 2 (non-termination)

Now, let's prove that the command **while true do skip** does not terminate. To prove this, it is enough to show that there is no proof tree of finite height for this command.

We will prove this fact by contradiction. Suppose $\langle \mathbf{while\ true\ do\ skip}, \sigma \rangle \to \sigma'$ for some σ and σ'. Then there must exist a finite proof tree having this relation as conclusion. Since we don't know if this tree is unique, consider the proof tree with minimum height h, which proves the evaluation of our while loop. By inspecting all of the semantic rules, we notice that there are two rules which have the while loop in their conclusion; of these, only one is applicable, the one where the test condition evaluates to *true*. Hence, our minimum-height proof tree has the following form:

$$\frac{\dfrac{}{\langle \mathbf{true}, \sigma \rangle \to \mathit{true}} \quad \dfrac{}{\langle \mathbf{skip}, \sigma \rangle \to \sigma} \quad \dfrac{\vdots(\text{repetition})}{\langle \mathbf{while\ true\ do\ skip}, \sigma \rangle \to \sigma'}}{\langle \mathbf{while\ true\ do\ skip}, \sigma \rangle \to \sigma'}$$

We see that the third premise is the same as the conclusion. The subtree for this premise has proof height $h-1$, and proves the same evaluation relation. This contradicts the fact that h is the minimum height of a proof tree for the evaluation of the while loop. Hence, there is no finite proof tree for the execution of the loop.

2.4 Error Handling

Suppose we extend the set of arithmetic operations with division:

$$a ::= \ldots \quad | \quad a_0/a_1$$

If $a_1 = 0$, evaluation of the expression should result in an error. One alternative for expressing this is to change the rule for arithmetic expressions as follows:

$$\frac{\langle a_0, \sigma\rangle \to n_0 \quad \langle a_1, \sigma\rangle \to n_1}{\langle a_0/a_1, \sigma\rangle \to n} \text{ where } n = n_0/n_1,\ n_1 \neq 0$$

However, there are problems with this solution. Suppose we are given the command $X := 1/(1-1)$. The result of evaluating $(1-1)$ is zero, so the above rule does not apply. If we try to build a proof tree for this assignment, we get to a point where none of the rules apply. Hence, we cannot build a proof tree in this case. Since non-termination also results in the imposibility of constructing a proof tree, it means that we cannot distinguish between errors and non-termination. Note, however, that the reason for not being able to construct a tree is different in the two cases: in one case none of the rules apply, in the other, rules always apply, but are repetitive.

Another alternative is to extend $\mathbb{Z}$ with a special error value: $\mathbb{Z} = \mathbb{Z} \cup \{\mathit{error}\}$, which allows us to distinguish between errors and nontermination. Then, whenever we detect an error case, the evaluation yields an *error* value:

$$\frac{\langle a_0, \sigma\rangle \to n \quad \langle a_1, \sigma\rangle \to 0}{\langle a_0/a_1, \sigma\rangle \to \mathit{error}}$$

But we are not done yet. We need to propagate the error down to the root of the tree. For example, among others, we need to add the following rule:

$$\frac{\langle a_0, \sigma\rangle \to \mathit{error}}{\langle a_0 \oplus a_1, \sigma\rangle \to \mathit{error}}$$

Similarly, we need to extend $\mathbb{T}$ and Σ, and add special rules for them. Thus, when using explicit *error* values in large step semantics, we end up needing many rules for dealing with propagation of errors.

2.5 Order of Evaluation

Consider the following rule:

$$\frac{\langle a_0, \sigma\rangle \to n_0 \quad \langle a_1, \sigma\rangle \to n_1}{\langle a_0 + a_1, \sigma\rangle \to n} \text{ where } n = n_0 + n_1$$

Should we evaluate $\langle a_0, \sigma\rangle \to n_0$ or $\langle a_1, \sigma\rangle \to n_1$ first? The semantics described so far does not enforce any specific order.

There are many ways how we can extend our semantics to specify order of evaluation. For example, if arithmetic expressions have side effects, we can enforce a particular order of evaluation with the following rule:

$$\frac{\langle a_0, \sigma\rangle \to \langle n_0, \sigma'\rangle \quad \langle a_1, \sigma'\rangle \to \langle n_1, \sigma''\rangle}{\langle a_0 + a_1, \sigma\rangle \to \langle n, \sigma''\rangle} \text{ where } n = n_0 + n_1$$

The evaluation in the second premise needs the state σ', which is the result of the evaluation in the first premise. Hence, the expression a_0 in the first premise has to be evaluated before a_1.

It is possible to specify a particular order of evaluation for expressions, even if these do not have side effects. For this, one needs to extend the configurations in the operational semantics with additional information (e.g., a stack of evaluation results for sub-expressions), to capture the fact that the result of the evaluation for a subexpression must be available when another subexpression is being evaluated. The particular form of the configurations and of the semantic rules is left as an exercise.

2.6 Non-Deterministic Choice

Suppose we extend our set of commands with a new command that non-deterministically executes one of its two subcommands:

$$c ::= \ldots \;\mid\; c_0 \Box c_1$$

The large-step operational semantics for this command is:

$$\frac{\langle c_0, \sigma\rangle \to \sigma'}{\langle c_0 \Box c_1, \sigma\rangle \to \sigma'} \qquad \frac{\langle c_1, \sigma\rangle \to \sigma'}{\langle c_0 \Box c_1, \sigma\rangle \to \sigma'}$$

An issue here is if one rule terminates and the other one does not, then we should choose the command that terminates, so that the non-deterministic command also terminates. But, how do we know which command to choose? An language interpreter implemented according to the above semantic rules must guess what the right choice is.

2.7 Parallelism

To support parallelism we can add a new command which executes its two subcommands in parallel:

$$c ::= \ldots \;\mid\; c_0 \| c_1$$

The large-step operational semantics for this command is:

$$\frac{\langle c_0, \sigma\rangle \to \sigma' \quad \langle c_1, \sigma'\rangle \to \sigma''}{\langle c_0 \| c_1, \sigma\rangle \to \sigma''} \qquad \frac{\langle c_1, \sigma\rangle \to \sigma' \quad \langle c_0, \sigma'\rangle \to \sigma''}{\langle c_0 \| c_1, \sigma\rangle \to \sigma''}$$

Essentially, the above rules define $c_0 \| c_1$ as being equivalent to $(c_0; c_1) \Box (c_1; c_0)$. Note that this expresses only two possible interleavings of commands from the parallel threads: either c_0 executes first, followed by c_1, or c_1 executes first, followed by c_0. In other words, this expresses only coarse-grain parallelism; large-step operational semantics is unable to specify fine-grained parallelism. For example, consider the program $(c_0; c_1) \| (c_2; c_3)$. We cannot express the execution order $(c_0; c_2; c_1; c_3)$ in large-step operational semantics.