Scribe: Kanghoe Kim, Sharad Goel Lecturer: Radu Rugina

1 Termination of programs

How do we prove that **while true do skip** does not terminate?

Suppose $\langle \textbf{while true do skip}, \sigma\rangle \to \sigma'$ for some σ and σ'. Then it has a proof tree. Note there is only one rule that can be applied with $\langle \textbf{while true do skip}, \sigma\rangle \to \sigma'$ as the conclusion. So we build the proof tree starting with that rule:

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

The third premise is the same as the root. So there is an unending chain of rules for **while true do skip**. Thus there is no proof tree of finite height for this program (assuming there is a unique proof tree for each conclusion, which is the case in **IMP**).

Alternatively, if there may be multiple proofs for the same fact, we could argue that we get to a contradiction, based on the height of the minimum proof tree. Suppose we have a proof tree of minimum height h. It must be of the same form as above. Now its subtree including the third premise is also a proof tree for **while true do skip** of height $h-1$. This is contradiction to assumption about minimum height h.

2 Issues with large step semantics

Below we explore several potential problems with large step semantics. We try to extend the syntax of IMP with additional commands and see how easy is it to capture the effect of these new constructs in the semantics. We also investigate how can operational semantics capture specifications such as the ordering of expression evaluation. In other words, we explore how robust large-step operational semantics is.

Errors If we add operations such as division that can result in an error (e.g. division by 0) then we would need to add and error handling mechanism to our rules. To do this, we must add error values and states: So now we have $n \in \mathbb{Z} \cup \{\text{Error}\}$, t $\in \mathbb{T} \cup \{\text{Error}\}$, and $\sigma \in \Sigma \cup \{\text{Error}\}$. We then need to add rules to explicitly handle the error:

$$\frac{\langle a_0, \sigma\rangle \to n_0 \quad \langle a_1, \sigma\rangle \to 0}{\langle a_0 / a_1, \sigma\rangle \to \text{ Error}}$$

But then, we need to propagate the error. For example, we need to add rules such as,

$$\frac{\langle a_0, \sigma\rangle \to \text{ Error}}{\langle a_0 \ op \ a_1, \sigma\rangle \to \text{ Error}}$$

Order of Evaluation If we have a rule such as,

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

then there is some ambiguity concerning the order of evaluation: 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.

We can specify a particular order of evaluation by using an explicit stack s which keeps track of the already computed sub-expressions and therefore characterizes the depth-first traversal of the proof tree. The evaluation rule using such a stack would be of the following form:

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

Non-Determinism Suppose we extend our commands with a new construct, which non-deterministically executes one of its two sub-commands $c ::= \ldots | c_0 \Box c_1$. The large step operational semantics for such a construct would then be:

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

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. In other words, we have to guess what the right choice is. To address this problem, we can execute both rules in parallel and use the rule that terminates first. But for that, we have to see how we can express parallelism and parallel constructs in our operational semantics framework.

Parallelism To include parallelism we can add a new command construct which executes its components in parallel $c ::= \ldots \ | \ c_0 | c_1$. The operational semantics of such a constructs can be defined as follows:

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

We could alternatively define the semantics of the parallel construct in terms of the non-deterministic choice operator, as follows:

$$\frac{\langle (c_0; c_1) \Box (c_1; c_0), \sigma\rangle \to \sigma'}{\langle c_0 | c_1, \sigma\rangle \to \sigma'}.$$

The problem here is that large step semantics cannot express fine grain interleaved executions; hence, it cannot express parallelism. For example, think of a program $(c_0; c_1) | (c_2; c_3)$. We cannot express the interleaved execution $(c_0; c_2; c_1; c_3)$ using this semantics. With large-step semantics we can only express two interleavings $(c_0; c_1; c_2; c_3)$ and $(c_2; c_3; c_0; c_1)$ out of the $4!/(2! * 2!) = 6$ possible interleavings.

3 Small-step operational semantics

We now present an alternative operational semantics, where the goal is to characterize small (atomic) steps during the the execution of the program. This semantics describes the individual transitions between program configurations $\langle c, \sigma\rangle$ consisting of a command c left to execute and a current state σ. The execution of the program will then be a sequence of such small step transitions:

$$\langle c, \sigma\rangle \to_1 \langle c', \sigma'\rangle \to_1 \langle c'', \sigma''\rangle \to_1 \cdots \to_1 \langle \mathbf{skip}, \sigma_1\rangle$$

Here $\langle \mathbf{skip}, \sigma\rangle$ means there is nothing left to execute and so we halt. The large-step equivalent of this execution would be $\langle c, \sigma\rangle \to \sigma_1$.

Hence, the small-step operational semantics is described using three relations that express how the program execution is carried out in small steps:

- For arithmetic expressions: $\to_1 \subseteq (\mathbf{Aexp} \times \Sigma) \times \mathbf{Aexp}$
- For boolean expressions: $\to_1 \subseteq (\mathbf{Bexp} \times \Sigma) \times \mathbf{Bexp}$
- For commands: $\to_1 \subseteq (\mathbf{Com} \times \Sigma) \times (\mathbf{Com} \times \Sigma)$

4 Small-step semantics for IMP

4.1 Semantics for Commands

- **skip** command:

  $\langle \mathbf{skip}, \sigma\rangle$ has no rule, for it is meant to be the final configuration.

- Assignment

$$\frac{\langle a, \sigma\rangle \rightarrow_1 a'}{\langle X := a, \sigma\rangle \rightarrow_1 \langle X := a', \sigma\rangle}$$

$$\frac{}{\langle X := n, \sigma\rangle \rightarrow_1 \langle \mathbf{skip}, \sigma[X \mapsto n]\rangle}$$

- Sequencing

$$\frac{\langle c_0, \sigma\rangle \rightarrow_1 \langle c_0', \sigma'\rangle}{\langle c_0; c_1, \sigma\rangle \rightarrow_1 \langle c_0'; c_1, \sigma'\rangle}$$

$$\frac{}{\langle \mathbf{skip}; c_1, \sigma\rangle \rightarrow_1 \langle c_1, \sigma\rangle}$$

- Conditional

$$\frac{\langle b, \sigma\rangle \rightarrow_1 b'}{\langle \mathbf{if}\, b\, \mathbf{then}\, c_0\, \mathbf{else}\, c_1, \sigma\rangle \rightarrow_1 \langle \mathbf{if}\, b'\, \mathbf{then}\, c_0\, \mathbf{else}\, c_1, \sigma\rangle}$$

$$\frac{}{\langle \mathbf{if}\, \mathbf{true}\, \mathbf{then}\, c_0\, \mathbf{else}\, c_1, \sigma\rangle \rightarrow_1 \langle c_0, \sigma\rangle}$$

$$\frac{}{\langle \mathbf{if}\, \mathbf{false}\, \mathbf{then}\, c_0\, \mathbf{else}\, c_1, \sigma\rangle \rightarrow_1 \langle c_1, \sigma\rangle}$$

(more on next lecture)