Written Assignment 4

CS 538, Spring 2020

March 24, 2020

In these problems, you'll be working with the basic imperative language (a so-called *While-language*) we saw in class. We first set up the language grammar.

$$
\begin{aligned}
\text{Var } x &:: \cdots \\
\text{AExp } a &::= x \mid \mathbb{Z} \mid a_1 + a_2 \mid a_1 \times a_2 \mid \cdots \\
\text{BExp } b &::= b_1 \,\&\&\, b_2 \mid a_1 < a_2 \mid \cdots \\
\text{Comm } c &::= \textbf{skip} \mid x \leftarrow a \mid c_1;\, c_2 \mid \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2 \mid \textbf{while } b \textbf{ do } c
\end{aligned}
$$

In words, we have:

- program variables (we use the letter x for any variable);
- arithmetic expressions (we use the letter a for any arith expr.);
- boolean expressions (we use the letter b for any boolean expr.);
- commands (we use the letter c for any command).

In our language all variables will be integer-valued; we use the letter n for any numeric constant (like 42).

In imperative languages, the behavior of a program depends on the current state of its variables. We will model this state with a *store* s, a function mapping every variable in Var to an integer in $\mathbb{Z}$. We write $s(x)$ to mean the current value of x in s, and we write $s[x \mapsto n]$ to mean the updated store where all variables hold the same value as in s, except variable x is updated to hold the integer n. Throughout, we use the letter s for any store.

Each arithmetic expression represents an integer in a given store s. We can define:

$$
\begin{aligned}
s(n) &= n \\
s(a_1 + a_2) &= s(a_1) \text{ plus } s(a_2) \\
s(a_1 \times a_2) &= s(a_1) \text{ times } s(a_2)
\end{aligned}
$$

and so on. The right-hand side of each case above is a number—we write "plus" and "times" out to emphasize that these are the mathematical addition and multiplication on numbers, not symbols in a program. Likewise, each boolean expression represents a boolean in a given store s. We can define:

$$
\begin{aligned}
s(b_1 \,\&\&\, b_2) &= s(b_1) \text{ and } s(b_2) \\
s(a_1 < a_2) &= s(a_1) \text{ is less than } s(a_2)
\end{aligned}
$$

and so on. Again, the right-hand side of each case is a boolean.

Running a command *changes* the store—it may write to variables in memory. Furthermore, the next command to execute also depends on the store. Accordingly, we will define an operational semantics on *configurations*, (command, store) pairs: $(c, s) \to (c', s')$ means that running command c on store s leads to a new store s' and a remaining command c'. The operational semantics are as follows:

$$\frac{s(a) = n}{(x \leftarrow a, s) \to (\mathbf{skip}, s[x \mapsto n])} \qquad \frac{(c_1, s) \to (c_1', s')}{(c_1; c_2, s) \to (c_1'; c_2, s')} \qquad \frac{}{(\mathbf{skip}; c_2, s) \to (c_2, s)}$$

$$\frac{s(b) = \mathit{true}}{(\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2, s) \to (c_1, s)} \qquad \frac{s(b) = \mathit{false}}{(\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2, s) \to (c_2, s)}$$

$$\frac{s(b) = \mathit{true}}{(\mathbf{while}\ b\ \mathbf{do}\ c, s) \to (c; \mathbf{while}\ b\ \mathbf{do}\ c, s)} \qquad \frac{s(b) = \mathit{false}}{(\mathbf{while}\ b\ \mathbf{do}\ c, s) \to (\mathbf{skip}, s)}$$

The pair $(\mathbf{skip}, s)$ does not step: this is the halting configuration, representing a program that is done.

1 While-language: basics (15)

Suppose there are just two variables: x and y. We'll write $[n_1, n_2]$ for the store where x holds n_1 and y holds n_2. Using the operational semantics and starting in the initial store $s_0 = [0, 0]$ (i.e., x and y are initialized to zero), step the following configurations until they reach the halting configuration:

(a) $(x \leftarrow x + 1, s_0)$

(b) $(x \leftarrow 3; y \leftarrow y + x, s_0)$

(c) $(\mathbf{if}\ x < y\ \mathbf{then}\ x \leftarrow 2\ \mathbf{else}\ x \leftarrow -2, s_0)$

(d) $(x \leftarrow -1; \mathbf{if}\ x < y\ \mathbf{then}\ x \leftarrow 2\ \mathbf{else}\ x \leftarrow -2, s_0)$

(e) $(x \leftarrow 2; \mathbf{while}\ y < x\ \mathbf{do}\ y \leftarrow y + 1, s_0)$

For each problem, you should write down a sequence $(c_1, s_1) \to (c_2, s_2) \to \cdots \to (c_n, s_n)$, where the last pair should be $(\mathbf{skip}, s)$ for some store s. Of course, your answers should involve specific commands and stores rather than using the letters c and s.

2 While-language: conditionals (10)

Rust's match construct is like a conditional command. We'll model a baby version of this feature, adding the following syntax to our language:

$$\text{Comm } c ::= \cdots \mid \mathbf{cond}\ \{b \Rightarrow c, \_ \Rightarrow c'\} \mid \mathbf{cond}\ \{b_1 \Rightarrow c_1, b_2 \Rightarrow c_2, \_ \Rightarrow c_3\}$$

The idea is that $\mathbf{cond}\ \{b \Rightarrow c, \_ \Rightarrow c'\}$ executes c if b is true in the initial store, otherwise it executes c'. Likewise, $\mathbf{cond}\ \{b_1 \Rightarrow c_1, b_2 \Rightarrow c_2, \_ \Rightarrow c_3\}$ executes c_1 if b_1 is true, otherwise it executes c_2 if b_2 is true, otherwise it executes c_3. Note that the arms are considered in order—left to right—and the guards b_1, b_2 might overlap.

Extend the operational semantics to handle these features. *Do not use if-then-else for this part!* Hint: you can get by with four new rules like the ones above: two for stepping the two-armed conditional, and two for stepping the three-armed conditional.

3 While-language: do-until (5)

We can also extend the language with a do-until loop:

$$\text{Comm } c ::= \cdots \mid \textbf{do } c \textbf{ until } b$$

This command should *always* execute the body c at least once. Then, it should exit if b is true, and continue looping if b is false.

Extend the operational semantics to handle this kind of loop. *Do not use While loops for this part! (You can use everything else, though.)* Hint: you can get by with just one new rule.