Programming Languages and Computation

Week 7: Semantics of the While Language

We consider the semantics of the While language and reason about it.

We don't reason about programs written in it in this sheet because the last two videos for the week have broken sound, and I'll be shifting content to next week instead. This might turn out to be much more exciting.

Unrolling Execution Traces

In this section, justify each of your answers with a derivation based on the formal definitions of semantics given in the reference material.

Evaluating Arithmetic and Boolean Expressions

* 1. Evaluate the following arithmetic expressions in state $\sigma = \begin{bmatrix} \mathtt{a} \mapsto 42 \\ \mathtt{b} \mapsto 154 \\ \mathtt{x} \mapsto 11 \end{bmatrix}$

(a) `a + x * 14 - b`

(b) `x * (a - x) * b`

Solution

The results can be obtained easily by doing a simple substitution and evaluating. The interest of asking for a derivation based on the formal definitions is to force thinking about associativity and the correspondence between the abstract syntax and the ASTs.

I only expand the first. The second has similar associativity.

(a)

$$\begin{aligned}
[\![\mathtt{a + x * 14 - b}]\!]^{\mathscr{A}}(\sigma) &= [\![\mathtt{a + x * 14}]\!]^{\mathscr{A}}(\sigma) - [\![\mathtt{b}]\!]^{\mathscr{A}}(\sigma) \\
&= [\![\mathtt{a}]\!]^{\mathscr{A}}(\sigma) + [\![\mathtt{x * 14}]\!]^{\mathscr{A}}(\sigma) - [\![\mathtt{b}]\!]^{\mathscr{A}}(\sigma) \\
&= [\![\mathtt{a}]\!]^{\mathscr{A}}(\sigma) + [\![\mathtt{x}]\!]^{\mathscr{A}}(\sigma) * 14 - [\![\mathtt{b}]\!]^{\mathscr{A}}(\sigma) \\
&= \sigma(a) + \sigma(x) * 14 - \sigma(b) \\
&= 42 + 11 * 14 - 154 \\
&= 42
\end{aligned}$$

(b)

$$[\![\mathtt{x * (a - x) * b}]\!]^{\mathscr{A}}(\sigma) = 52514$$

* 2. Evaluate the following arithmetic expressions in state $\sigma = \begin{bmatrix} \mathtt{x} \mapsto 11 \\ \mathtt{b} \mapsto 12 \\ \mathtt{h} \mapsto 84 \end{bmatrix}$

(a) `a + x * 14 - b`

(b) `b - g * x + h`

Solution

The only additional difficulty is that you need to remember that looking up a variable we have not explicitly defined yields value 0.

(a)

$$[\![\mathtt{a + x * 14 - b}]\!]^{\mathscr{A}}(\sigma) = 142$$

(b)

$$[\![\mathtt{b - g * x + h}]\!]^{\mathscr{A}}(\sigma) = 86$$

* 3. Evaluate the following boolean expressions in state $\sigma = \begin{bmatrix} \mathtt{a} \mapsto 42 \\ \mathtt{b} \mapsto 154 \\ \mathtt{x} \mapsto 11 \end{bmatrix}$

(a) $\mathtt{a + x * 14 - b} \leq \mathtt{b + x} \land \mathtt{true}$

(b) $\mathtt{!x * (a - x) * b} = \mathtt{b + x} \land \mathtt{x} = \mathtt{11}$

(c) $\mathtt{b - g * x + h} \leq \mathtt{0}$

Solution

No additional difficulty here again. Recall that `!` binds stronger than $\land$.

(a)

$$[\![\mathtt{a + x * 14 - b} \leq \mathtt{b + x} \land \mathtt{true}]\!]^{\mathscr{B}}(\sigma) = \top$$

(b)

$$[\![\mathtt{!x * (a - x) * b} = \mathtt{b + x} \land \mathtt{x} = \mathtt{11}]\!]^{\mathscr{B}}(\sigma) = \top$$

(c)

$$[\![\mathtt{b - g * x + h} \leq \mathtt{0}]\!]^{\mathscr{B}}(\sigma) = \bot$$

```
r ← 1
while (0 < n) {
r ← n * r
n ← n - 1
```

Figure 1: The `factorial` program.

Writing While programs

* 4. Write the following While programs. You may find it useful to refer to the example While program shown in Figure 1, which computes a factorial, taking its input from variable `n`, and placing its output in variable `r`.

   (a) A program that increments the value stored in variable `n` by 2.

   (b) A program that stores in variable `r` the absolute value of the variable initially stored in variable `n`.

   (c) A program that swaps the values stored in variables `a` and `b`. (While programs operate over arbitrary integers in $\mathbb{Z}$ so you do not need to use a temporary variable, but you may do so.)

   (d) A program that stores in variable `r` the remainder of the initial value of variable `n` in the division by 2. (If I were to write this as a C program: `r = n % 2`.) Assume that `n` is initially non-positive.

   (e) A program that loops forever, constantly increasing by 1 the value of variable `n`.

Solution

The following show decently bracketed and parenthesized programs. Other options exist. We use **unlines** to pretend we have multi-line strings. Note that our choice to not have semicolons as line enders means it's really a good idea to make these multi-line strings...

(a) `n := n + 2`

(b) `if (-1 < n) then r := n else r := 0 - n`

(c)
```
b ← a + b
a ← b - a
b ← b - a
```

(d)
```
while (0 < n) {
r ← n
n ← n - 2
}
```

(e)
```
 while (true) {
n ← n + 1
}
```

** 5. Modify the program from Figure 1 so that, if the value n of variable `n` is initially negative, the program terminates with the value of $-(|n|!)$ in variable `r`. (That is, the opposite of the factorial of the opposite of n.) Its behaviour on non-negative values of n should be unchanged.

Solution

```
if (n < 0)
then m ← 0 - n
```

```
else m ← n
r ← 1
while (0 < m) {
r ← r * m
m ← m - 1
}
if (n < 0)
then r ← 0 - r
else
```

Execution Traces

** 6. Consider the While programs you wrote in Question 1 in Week 5. (You may use those from the solution.) Give their execution trace in the following state σ when the trace is finite.

$$\sigma = \begin{bmatrix} \mathtt{n} \mapsto 3 \\ \mathtt{a} \mapsto 5 \end{bmatrix}$$

Solution

(a) `n ← n + 2`

$$\langle \mathtt{n \leftarrow n + 2}, \sigma\rangle \Rightarrow \left\langle \mathtt{skip}, \begin{bmatrix} \mathtt{n} \mapsto 5 \\ \mathtt{a} \mapsto 5 \end{bmatrix} \right\rangle$$

(b) $S =$ `if (0 < n) then r ← n else r ← 0 - n`

$$\begin{aligned} \langle S, \sigma\rangle &\Rightarrow \langle \mathtt{r \leftarrow n}, \sigma\rangle \\ &\Rightarrow \left\langle \mathtt{skip}, \begin{bmatrix} \mathtt{n} \mapsto 3 \\ \mathtt{a} \mapsto 5 \\ \mathtt{r} \mapsto 3 \end{bmatrix} \right\rangle \end{aligned}$$

(c) $S =$ `b ← a + b; a ← b - a; b ← b - a`

$$\begin{aligned} \langle S, \sigma\rangle &\Rightarrow \left\langle \mathtt{a \leftarrow b - a;\ b \leftarrow b - a}, \begin{bmatrix} \mathtt{n} \mapsto 3 \\ \mathtt{a} \mapsto 5 \\ \mathtt{b} \mapsto 5 \end{bmatrix} \right\rangle \\ &\Rightarrow \left\langle \mathtt{b \leftarrow b - a}, \begin{bmatrix} \mathtt{n} \mapsto 3 \\ \mathtt{a} \mapsto 0 \\ \mathtt{b} \mapsto 5 \end{bmatrix} \right\rangle \\ &\Rightarrow \left\langle \mathtt{skip}, \begin{bmatrix} \mathtt{n} \mapsto 3 \\ \mathtt{a} \mapsto 0 \\ \mathtt{b} \mapsto 5 \end{bmatrix} \right\rangle \end{aligned}$$

(d) $S =$ `while (0 < n)  r ← n; n ← n - 2`

$$
\begin{aligned}
\langle S, \sigma \rangle &\Rightarrow \langle \texttt{r} \leftarrow \texttt{n; n} \leftarrow \texttt{n - 2; } S, \sigma \rangle \\
&\Rightarrow^2 \left\langle S, \begin{bmatrix} \texttt{n} \mapsto 1 \\ \texttt{a} \mapsto 5 \\ \texttt{r} \mapsto 3 \end{bmatrix} \right\rangle \\
&\Rightarrow \left\langle \texttt{r} \leftarrow \texttt{n; n} \leftarrow \texttt{n - 2; } S, \begin{bmatrix} \texttt{n} \mapsto 1 \\ \texttt{a} \mapsto 5 \\ \texttt{r} \mapsto 3 \end{bmatrix} \right\rangle \\
&\Rightarrow^2 \left\langle S, \begin{bmatrix} \texttt{n} \mapsto \texttt{-1} \\ \texttt{a} \mapsto 5 \\ \texttt{r} \mapsto 1 \end{bmatrix} \right\rangle \\
&\Rightarrow \left\langle \texttt{skip}, \begin{bmatrix} \texttt{n} \mapsto \texttt{-1} \\ \texttt{a} \mapsto 5 \\ \texttt{r} \mapsto 1 \end{bmatrix} \right\rangle
\end{aligned}
$$

(e) The last program does not terminate.

*** 7. Find an initial configuration (program and state) that do not give rise to any finite complete traces, but such that all infinite traces have no repeating configurations.
(This shows that deciding termination is more complex than simply detecting cycles.)

Solution

The following program suffices: the statement component of the configuration only takes two different values, but the state component changes without repeating. This is true for any initial state.

```
while (true) x ← x + 1
```

Properties of While The While language (and its expressions and boolean expressions) are also defined inductively. This means we can reason about all those objects inductively.

**** 8. On syntax, I mentioned that the associativity of sequential composition did not in fact matter, and chose to make it right associative. We'll first explore the choice, then the claim. This only leverages mathematical induction.

(a) Identify the next configuration for the following two programs, in some abstract state σ, justifying it fully using the rules of the semantics for While:

1. `x ← y; (z ← x; y ← z)`, and
2. `(x ← y; z ← x); y ← z`.

(b) Show that, if $\langle S_1; S_2, \sigma \rangle \Rightarrow^k \langle \texttt{skip}, \sigma'' \rangle$, then there must exist some state σ' and some natural number $k_1 \leq k$ such that $\langle S_1, \sigma \rangle \Rightarrow^{k_1} \langle \texttt{skip}, \sigma' \rangle$ and $\langle S_2, \sigma' \rangle \Rightarrow^{k-k_1} \langle \texttt{skip}, \sigma'' \rangle$.
(In other words, there is a suffix of the complete execution trace for $S_1; S_2$ in state σ that is an execution trace for S_2 in some state σ' that happens to be the terminal state when executing S_1 in $sigma$.)
This proof is by mathematical induction on k. You'll want to isolate the *first* transition in the inductive step of the proof, and do the appropriate case analysis.

(c) Show that, if $\langle S_1, \sigma\rangle \Rightarrow^k \left\langle \texttt{skip}, \sigma'\right\rangle$, then $\langle S_1; S_2, \sigma\rangle \Rightarrow^k \left\langle S_2, \sigma'\right\rangle$.
As before, this proof is by mathematical induction on k, isolating the first transition.

(d) Show that, if $\langle S_1; (S_2; S_3), \sigma\rangle \Rightarrow^* \left\langle \texttt{skip}, \sigma_3\right\rangle$, then $\langle (S_1; S_2); S_3, \sigma\rangle \Rightarrow^* \left\langle \texttt{skip}, \sigma_3\right\rangle$.
(The converse also holds, but the proof is roughly the same, and quite tedious.)

Solution

(a) The point here is to have you reflect on what it takes to identify the next configuration (which will look very similar in linear notation but would be different if we drew out the AST).

1. In the first case, the sequence rule where the first statement executes in one step gives us the next configuration right away.
Since we have $\left\langle \texttt{x} \leftarrow \texttt{y}, \sigma\right\rangle \Rightarrow \left\langle \texttt{skip}, \sigma'\right\rangle$, we also have $\left\langle \texttt{x} \leftarrow \texttt{y};\ (\texttt{z} \leftarrow \texttt{x};\ \texttt{y} \leftarrow \texttt{z}), \sigma\right\rangle \Rightarrow \left\langle \texttt{z} \leftarrow \texttt{x};\ \texttt{y} \leftarrow \texttt{x}, \sigma'\right\rangle$.
2. In the second case, the reason why this (same) transition is possible requires two levels of justification.
Since we have $\left\langle \texttt{x} \leftarrow \texttt{y}, \sigma\right\rangle \Rightarrow \left\langle \texttt{skip}, \sigma'\right\rangle$, we also have $\left\langle \texttt{x} \leftarrow \texttt{y};\ \texttt{z} \leftarrow \texttt{x}, \sigma\right\rangle \Rightarrow \left\langle \texttt{z} \leftarrow \texttt{x}, \sigma'\right\rangle$. and therefore we have $\left\langle (\texttt{x} \leftarrow \texttt{y};\ \texttt{z} \leftarrow \texttt{x});\ \texttt{y} \leftarrow \texttt{z}, \sigma\right\rangle \Rightarrow \left\langle \texttt{z} \leftarrow \texttt{x};\ \texttt{y} \leftarrow \texttt{z}, \sigma'\right\rangle$.

Now imagine if an entire n-line program is parsed into a left-associative way: justifying its first transition take $n-1$ reasoning steps.

(b) By mathematical induction over the length k of the trace.

Case $k = 0$: $\langle S_1, \sigma\rangle = \left\langle \texttt{skip}, \sigma''\right\rangle$, and we therefore have $\langle S_1; S_2, \sigma\rangle = \left\langle \texttt{skip}; S_2, \sigma\right\rangle \Rightarrow \langle S_2, \sigma\rangle$ and we conclude with $k_1 = 0$ and $\sigma' = \sigma$.

Case $k = n+1$: $\langle S_1, \sigma\rangle \Rightarrow \left\langle S_1', \sigma'\right\rangle \Rightarrow^n \left\langle \texttt{skip}, \sigma''\right\rangle$

Case $S_1' = \texttt{skip}$: Then we have $\langle S_1, \sigma\rangle \Rightarrow^1 \left\langle \texttt{skip}, \sigma'\right\rangle$, and therefore also $\langle S_1; S_2, \sigma\rangle \Rightarrow^1 \left\langle S_2, \sigma'\right\rangle \Rightarrow^n \left\langle \texttt{skip}, \sigma''\right\rangle$, and we can conclude with $k_1 = 1$ and σ' as defined.

Case $S_1' \neq \texttt{skip}$: Then we have $\langle S_1, \sigma\rangle \Rightarrow^1 \left\langle S_1', \sigma'\right\rangle$, and therefore also $\langle S_1; S_2, \sigma\rangle \Rightarrow^1 \left\langle S_1'; S_2, \sigma'\right\rangle \Rightarrow^n \left\langle \texttt{ski[}, \sigma''\right\rangle$. By induction hypothesis, there exists a state σ_1 and some integer $k_1' \leq n$ such that $\left\langle S_1', \sigma'\right\rangle \Rightarrow^{k_1'} \left\langle \texttt{skip}, \sigma_1\right\rangle$ and $\langle S_2, \sigma_1\rangle \Rightarrow^{n-k_1'} \left\langle \texttt{skip}, \sigma''\right\rangle$. Therefore, we have $\langle S_1, \sigma\rangle \Rightarrow^{k_1'+1} \left\langle \texttt{skip}, \sigma_1\right\rangle$, $\langle S_2, \sigma_1\rangle \Rightarrow^{n-k_1'-1} \left\langle \texttt{skip}, \sigma''\right\rangle$ and $k_1' + 1 \leq n + 1$, and we conclude.

(c) By mathematical induction on k.

Case $k = 0$: By definition of $\Rightarrow$.

Case $k = n+1$: $\langle S_1, \sigma\rangle \Rightarrow^1 \left\langle S_1', \sigma''\right\rangle \Rightarrow^n \left\langle \texttt{skip}, \sigma'\right\rangle$

Case $S_1' = \texttt{skip}$: Then $n = 0$ and we conclude by definition of $\Rightarrow$.

Case $S_1' \neq \texttt{skip}$: Then we have $\langle S_1; S_2, \sigma\rangle \Rightarrow \left\langle S_1'; S_2, \sigma''\right\rangle$ (by definition of $\Rightarrow$) and $\left\langle S_1'; S_2, \sigma''\right\rangle \Rightarrow^n \left\langle S_2, \sigma'\right\rangle$ (by induction hypothesis), and we conclude.

(d) Use (b) to deconstruct the trace into its three discrete components. Then use (c) to reconstruct the trace with the other associativity. It obviously also works the other way.