Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

A. Lochbihler and P. Müller

Formal Methods and Functional Programming

Exercise Sheet 10: Big Step Semantics

Submission deadline: May 5th, 2014

Assignment 1 (Applying Big-Step Semantics)

Consider the following **IMP** statement s:

```
while n # 0 do
    (a := a+n;
    b := b*n);
    n := n-1
end
```

Let σ be a state such that $\sigma(\mathtt{a}) = 0$, $\sigma(\mathtt{b}) = 1$, and $\sigma(\mathtt{n}) = 2$. Prove using the natural semantics that there is some state σ' with $\sigma'(\mathtt{a}) = 3$, $\sigma'(\mathtt{b}) = 2$, and $\sigma'(\mathtt{n}) = 0$ such that $\langle s, \sigma \rangle \to \sigma'$.

Provide the complete derivation tree. Don't forget to write the names of the rules you apply explicitly, at each derivation step.

Assignment 2 (Reversing loop-unrolling)

(Note: in the lectures, you have seen the proof of this result in the other direction).

Prove that: For all σ, σ', b, s:

$$\begin{array}{l} \quad \vdash \langle \mathtt{if}\ b\ \mathtt{then}\ s; \mathtt{while}\ b\ \mathtt{do}\ s\ \mathtt{end}\ \mathtt{else}\ \mathtt{skip}\ \mathtt{end}, \sigma \rangle \to \sigma' \\ \Rightarrow \\ \quad \vdash \langle \mathtt{while}\ b\ \mathtt{do}\ s\ \mathtt{end}, \sigma \rangle \to \sigma' \end{array}$$

Assignment 3 (Execution only affects free variables)

Prove that:

$$\forall s, \sigma, \sigma', x. (\vdash \langle s, \sigma \rangle \to \sigma' \ \wedge\ x \notin \mathit{FV}(s) \quad \Rightarrow \quad \sigma'(x) = \sigma(x))$$

(Hint: the statement to prove is equivalent to the following (T ranges over derivation trees):

$$\forall T, s, \sigma, \sigma', x. (\ (\mathit{root}(T) = \langle s, \sigma \rangle \to \sigma') \ \wedge\ x \notin \mathit{FV}(s) \quad \Rightarrow \quad \sigma'(x) = \sigma(x))$$

Assignment 4 (Adding for-loop to IMP)

Consider an extension of the **IMP** programming language with a `for` statement:

$$\texttt{for } x := e_1 \texttt{ to } e_2 \texttt{ do } s \texttt{ end}$$

The execution of the statement should first evaluate e_1 and assign the result to variable x. Then, while the value of x is not equal to that of e_2, it should repeatedly execute s and increase x by one.

There are two possible interpretations of the above statement: (i) e_2 is evaluated once, (ii) e_2 is evaluated before each comparison to the value of x.

(a) Provide derivation rules in natural semantics for both (i) and (ii). For this question, do *not* the `while` construct (or any IMP extensions from your exercise session) in your derivation rules.

(b) Consider the case of semantics (ii) only. Show that, for all $x, e_1, e_2, s, \sigma, \sigma'$:

$$\vdash \langle \texttt{for } x := e_1 \texttt{ to } e_2 \texttt{ do } s \texttt{ end}, \sigma\rangle \rightarrow \sigma'$$
$$\Rightarrow$$
$$\vdash \langle x := e_1; \texttt{ while } x \# e_2 \texttt{ do } s; x := x+1 \texttt{ end }, \ \sigma\rangle \rightarrow \sigma'$$

Note: This exercise is a bit more involved (and the proof depends on how you formulate rules for for loops).

Assignment 5 (Big-Step IMP Interpreter)

In this assignment you will write a simple interpreter for **IMP** programs in the programming language Haskell. The interpreter is based on the big-step semantics of **IMP** that you have seen in the lecture, and reuses the implementation of the **IMP** syntax from assignment 4 of sheet 9. The following steps are necessary:

1. **Syntax.** Make sure you have an implementation of the **IMP** syntax together with evaluation functions of arithmetic and boolean expressions ready. You can either take your solution from assignment 4 of sheet 9, or use the master solution for that exercise provided on the website. You should implement a data type for IMP statements, too.

2. **States.** Implement (or reuse) a data type for states. Recall that states are functions from variable names to integers. Furthermore, implement a constant `allZeroState` that represents the all-zeros state (where every variable evaluates to zero); this state will be used as the initial state for our example programs. Finally, implement a function `updateState` that implements a state update, as defined in the lectures. That is, given a state σ, a variable x and a value v, it returns a new state representing $\sigma[x \mapsto v]$.

3. **Configurations.** In big-step semantics, a configuration is either final or non-final. A final configuration consists simply of a state. A non-final configuration consists of an **IMP** program (statement) and a state on which we start to run the given **IMP** program. Implement a data type for configurations.

4. **Transition function.** Implement a transition function `transNS` to implement the big-step semantics as a function from configurations to configurations. Recall that in big-step semantics, transitions always go from non-final configurations to final configurations. Hint: your implementation should probably define one case for every rule in big-step semantics.

5. **Interpreter.** Implement a function `run` that takes an **IMP** program, runs that program starting in the state `allZeroState` and returns the resulting final state. In our programs, by convention we use a variable with the name `"result"` to indicate the overall result of the program. Write a function `result` that executes a program using run, and then returns the value of the result variable in the last state.

6. **Tests.** Write the simple programs from below in your Haskell data type of the **IMP** syntax and define a constant for each of them. The expected output for all programs is indicated, and you can use the `result` function to see if your interpreter computes the correct result.

Test programs

```
n := 2;
b := 1;
while n # 0 do
    a := a+n;
    b := b*n;
    n := n-1
end;
result := a
```

Expected result is 3, which you already proved in assignment 1 of this sheet.

```
result := 1;
n := 10;
while n > 1 do
    result := result*n;
    n := n-1
end
```

Expected result is 3628800.

```
x := 15648;
y := 3;
z := 0;
v := 0;
while v < x do
  v := 1;
  i := 0;
  while i < y do
    v := v * (z+1);
    i := i+1
```

```
  end;
  if v <= x then
    z := z+1
  else
    skip
  end
end;
result := z
```

The expected result is 25. The program computes $\lfloor \sqrt[3]{15648} \rfloor$ (in general, the while loops compute $\lfloor \sqrt[y]{x} \rfloor$).

Submission

Please mail your solution of this assignment together with the test cases to your tutor. The email addresses of the tutors are:

Alex Summers	`alexander.summers@inf.ethz.ch`
Milos Novacek	`milos.novacek@inf.ethz.ch`
Uri Juhasz	`uri.juhasz@inf.ethz.ch`
Alex Viand	`vianda@student.ethz.ch`
Cyril Steimer	`csteimer@student.ethz.ch`

Assignment 6 (Adding Break statement to IMP)

Extend the big-step semantics of **IMP** to support a `break` statement. That is, define suitable extra syntax and derivation rules to define the big-step semantics of this syntax. The break statement should stop the execution of the inner-most surrounding while loop that is being executed, as in C/Java. Execution should then resume immediately after the loop. You may assume that `break` will never appear outside of a while loop.
Hint: You might want to change the definition of *states* to help you to define this extension.
Note: This exercise is a bit more involved.