Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

A. Lochbihler and P. Müller

# Formal Methods and Functional Programming

## Exercise Sheet 11: Small Step Semantics

Submission deadline: May 12th, 2014

## Assignment 1 (Applying Small-Step Semantics)

Consider the following **IMP** statement $s$:

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

Let $\sigma$ be a state such that $\sigma(\mathtt{a}) = 0$, $\sigma(\mathtt{b}) = 1$, and $\sigma(\mathtt{n}) = 2$. Prove using the structural operational semantics that there is a state $\sigma'$ with $\sigma'(\mathtt{a}) = 3$, $\sigma'(\mathtt{b}) = 2$, and $\sigma'(\mathtt{n}) = 0$ such that $\langle s, \sigma \rangle \rightarrow_1^* \sigma'$. Provide the complete derivation sequence.

## Assignment 2 (k-step Execution Extension)

Let $s_1$ and $s_2$ be statements, $\sigma$ and $\sigma'$ states, and $k$ a positive integer. Prove that if $\langle s_1, \sigma \rangle \rightarrow_1^k \sigma'$ then $\langle s_1; s_2, \sigma \rangle \rightarrow_1^k \langle s_2, \sigma' \rangle$.

## Assignment 3 (Proof of Equivalence Lemmas)

Prove the following equivalence lemmas from lecture slides 153, 154.

(L1) $\forall \sigma, \sigma', s \cdot\ \vdash \langle s, \sigma \rangle \rightarrow \sigma' \ \Rightarrow\ \langle s, \sigma \rangle \rightarrow_1^* \sigma'$

(L2) $\forall \sigma, \sigma', s, k \cdot\ \langle s, \sigma \rangle \rightarrow_1^k \sigma' \ \Rightarrow \vdash \langle s, \sigma \rangle \rightarrow \sigma'$

## Assignment 4 (Small-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 *small*-step semantics of **IMP** that you have seen in the lecture. The following three ingredients can be taken from the previous assignments (your own or the master solutions):

1. **Syntax.** An implementation of the **IMP** syntax together with evaluation functions for arithmetic and boolean expressions.

2. **States.** A data type for states, a constant `allZeroState` that represents the all-zeros state (where every variable evaluates to zero) and a function `substState` that implements a state update. That is, given a state $\sigma$, a variable $x$ and a value $v$, it returns a new state representing $\sigma[x \mapsto v]$.

3. **Configurations.** A data type for configurations. 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 and a state on which we start to run the given **IMP** program.

Furthermore, you need to implement the following new features:

1. **Transition function.** Implement a transition function `transSOS` for structural semantics as a function from configurations to configurations. Your implementation should have exactly one case for every rule in structural semantics.

2. **Interpreter.** Implement a function `run` that takes an **IMP** program, runs that program starting in the all-zeros state and returns the final state. In our programs, by convention we use the 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.

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 (same as on previous sheet)

```
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 5 (Adding revert-if statement to IMP)

Consider the IMP extension
`revert` $s$ `if` $b$
which should have the following semantics: statement $s$ is executed, but the effects of executing $s$ must be reverted if boolean expression $b$ is true in the state after the execution of $s$. For example the following program has no effect on the state:

```
revert x:=0 if x=0
```

while the following is equivalent to `x:=0`

```
revert x:=0 if x!=0
```

(This construct is related to a very simple form of transaction management / conflict resolution as used e.g., by databases.)
Hint I: You might want to consider changing the definition of states.
Hint II: Ideally, your solution should support the possibility of `revert` statements being nested. But you might find it easier to consider the non-nested case (just only `revert` statement) first.
**Note**: This exercise is a bit more involved.