Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

D. Basin and P. Müller

# Formal Methods and Functional Programming

## Solutions of Exercise Sheet 10: Small Step Semantics

## Assignment 1

Consider the following case:

$$s_1 = \texttt{x:=0} \quad s_2 = (\texttt{while 1=1 do x:=x+1 end})$$

and the following derivation:

$$\langle s_1; s_2\ ,\ \sigma\rangle \ \rightarrow_1 \langle s_2\ ,\ \sigma[x \mapsto 0]\rangle \rightarrow_1^2 \langle s_2\ ,\ \sigma[x \mapsto 1]\rangle$$

and let $\sigma' = \sigma[x \mapsto 1]$. Therefore, this is one case where:

$$\langle s_1; s_2\ ,\ \sigma\rangle \rightarrow_1^* \langle s_2\ ,\ \sigma'\rangle$$

We want to show that it is not the case that $\langle s_1, \sigma\rangle \rightarrow \sigma'$. Assume the contrary. By the determinism of the semantics, we have that $\sigma'(x) = 0$, which contradicts the definition $\sigma' = \sigma[x \mapsto 1]$.

## Assignment 2

By induction on $k$, starting from 1.

The base case ($k = 1$) is equivalent to the first rule of the structural semantics for sequential composition, and therefore true.

Assume that the proposition holds for $k = n$, i.e. for all statements $p, q$ and states $\tau, \tau'$:

$$\langle p, \tau\rangle \rightarrow_1^n \tau' \ \Rightarrow\ \langle p; q\ ,\ \tau\rangle \rightarrow_1^n \langle q, \tau'\rangle \quad \text{IH}$$

Assume also that

$$\langle s_1, \sigma\rangle \rightarrow_1^{n+1} \sigma' \quad \text{A1}$$

We want to prove that

$$\langle s_1; s_2\ ,\ \sigma\rangle \rightarrow_1^{n+1} \langle s_2, \sigma'\rangle$$

From A1, we have that there is a configuration $\langle s_A, \sigma_A \rangle$ such that

$$\langle s_1, \sigma \rangle \rightarrow_1 \langle s_A, \sigma_A \rangle \rightarrow_1^n \sigma' \quad \text{A2}$$

which, by IH (instantiate: $\tau = \sigma_A$, $\tau' = \sigma'$, $p = s_A$ and $q = s_2$), becomes

$$\langle s_A; s_2\ , \sigma_A \rangle \rightarrow_1^n \langle s_2, \sigma' \rangle$$

and therefore what remains to be proven is:

$$\langle s_1; s_2, \sigma \rangle \rightarrow_1 \langle s_A; s_2\ , \sigma_A \rangle$$

This is proven by A2 and the second rule of the structural semantics for sequential composition.

# Assignment 3

You find a solution of this assignment in the literate Haskell file `simpi.lhs`.

# Assignment 4

To support coroutines, we augment the state with a number as follows:

$$\mathit{State}' = \mathit{State} \times \{-1, 0, 1\}$$

The meaning of this extra piece of state is as follows: 0 means that no coroutines are executing. 1 means that there are two coroutines executing and the left one is active. -1 means that there are two coroutines executing and the right one is active.

The semantics of all **IMP** statements remains the same. The extra piece of state that we introduced here is not touched. For example, the semantics of `skip` is given by:

$$\frac{}{\langle \texttt{skip}, (\sigma, n) \rangle \rightarrow_1 (\sigma, n)}$$

The semantics of `yield` is simple: it inverts the active coroutine:

$$\frac{}{\langle \texttt{yield}, (\sigma, n) \rangle \rightarrow_1 (\sigma, -n)}$$

Notice that if there are no coroutines, `yield` behaves like *skip*.

The first rule about `copar` says that the left subroutine is initially active:

$$\frac{}{\langle s_1 \texttt{ copar } s_2\ ,\ (\sigma, 0) \rangle \rightarrow_1 \langle s_1 \texttt{ copar } s_2\ ,\ (\sigma, 1) \rangle}$$

The second rule says that if the left coroutine is active, then it is executed:

$$\frac{\langle s_1, (\sigma, 1) \rangle \rightarrow_1 \langle s_1', (\sigma', n') \rangle}{\langle s_1 \texttt{ copar } s_2\ ,\ (\sigma, 1) \rangle \rightarrow_1 \langle s_1' \texttt{ copar } s_2\ ,\ (\sigma', n') \rangle}$$

Similarly, if the right coroutine is active, then it is executed:

$$\frac{\langle s_2, (\sigma, -1) \rangle \rightarrow_1 \langle s_2', (\sigma', n') \rangle}{\langle s_1 \texttt{ copar } s_2\ ,\ (\sigma, -1) \rangle \rightarrow_1 \langle s_1 \texttt{ copar } s_2'\ ,\ (\sigma', n') \rangle}$$

If the active coroutine terminates, then the whole construct terminates.

$$\frac{\langle s_1, (\sigma, 1)\rangle \rightarrow_1 (\sigma', 1)}{\langle s_1 \texttt{ copar } s_2\ ,\ (\sigma, 1)\rangle \rightarrow_1 (\sigma', 0)}$$

$$\frac{\langle s_2, (\sigma, -1)\rangle \rightarrow_1 (\sigma', -1)}{\langle s_1 \texttt{ copar } s_2\ ,\ (\sigma, -1)\rangle \rightarrow_1 (\sigma', 0)}$$

Notice in the last two rules that the `yield` statement was purposefully excluded. This is because when the last thing that a coroutine does is a `yield`, then the whole construct should not yet terminate, but first yield the control to the other coroutine. This is ensured by the final two rules:

$$\frac{}{\langle \texttt{yield copar } s_2\ ,\ (\sigma, 1)\rangle \rightarrow_1 \langle \texttt{skip copar } s_2\ ,\ (\sigma, -1)\rangle}$$

$$\frac{}{\langle s_1 \texttt{ copar yield}\ ,\ (\sigma, -1)\rangle \rightarrow_1 \langle s_1 \texttt{ copar skip}\ ,\ (\sigma, 1)\rangle}$$