CS6225 Spring 2025 IITM: Final Exam

May 10, 2025

1 Inter-process communication (30 points)

You will develop a small-step operational semantics for inter-process communication, and prove some theorems about it in Coq. This document presents the small-step operational semantics in Coq. Your job is to encode the same in Coq, and prove the theorems.

The language is an extension of the one seen in the operational semantics lecture.

$$
\begin{array}{rlll}
\text{Numbers} & n & \in & \mathbb{N} \\
\text{Variables} & x & \in & \text{Strings} \\
\text{ProcessId} & i & \in & \mathbb{N} \\
\text{Valuation} & v & ::= & x \mapsto \mathbb{N} \\
\text{Expressions} & e & ::= & n \mid x \mid e + e \mid e - e \mid e \times e \\
\text{Commands} & c & ::= & \text{skip} \mid x \leftarrow e \mid c; c \mid \text{if } e \text{ then } c \text{ else } c \mid \text{while } e \text{ do } c \\
 & & \mid & i!e \mid i?x \\
\text{Process} & p & ::= & (i, v, c) \\
\text{State} & s & ::= & [] \mid p :: s
\end{array}
$$

The state s is composed of a list of processes p, each of which is a triple (i, v, c) composed of process id, valuation and command. The command $i!e$ is *blocking send* operation, which sends the result of evaluating e to the process with the process id i. If there is no corresponding receive operation on the i, then the send blocks. The command $i?x$ is blocking recv operation, which receives a value n from the process i and extends the valuation with the variable x mapping to n.

We will use contextual small-step semantics to describe the operational semantics of this language. First, we define the evaluation context

$$\text{Evaluation contexts} \quad C \quad ::= \quad \square \mid C; e$$

and plugging evaluation contests:

$$
\begin{array}{rcl}
\square[c] & = & c \\
(C; c_2)[c] & = & C[c]; c_2
\end{array}
$$

Next, we define the basic step rules $(v, c) \rightarrow_0 (v', c')$, which is the same as what we had seen in the operational semantics lecture.

$$\frac{}{(v, x \leftarrow e) \rightarrow_0 (v[x \mapsto [\![e]\!]v, \text{skip})} \qquad \frac{}{(v, \text{skip}; c_2) \rightarrow_0 (v, c_2)}$$

$$\frac{[\![e]\!]v \neq 0}{(v, \text{if } e \text{ then } c_1 \text{ else } c_2) \rightarrow_0 (v, c_1)} \qquad \frac{[\![e]\!]v = 0}{(v, \text{if } e \text{ then } c_1 \text{ else } c_2) \rightarrow_0 (v, c_2)}$$

$$\frac{[\![e]\!]v \neq 0}{(v, \text{while } e \text{ do } c_1) \rightarrow_0 (v, c_1; \text{while } e \text{ do } c_1)} \qquad \frac{[\![e]\!]v = 0}{(v, \text{while } e \text{ do } c_1) \rightarrow_0 (v, \text{skip})}$$

Now, we now define the step rules $s \rightarrow s'$, that relates the states s and s':

$$\frac{s_1 \rightarrow s_2}{p :: s_1 \rightarrow p :: s_2} \quad \text{(Skip)}$$

$$\frac{}{p1 :: p2 :: s \rightarrow p2 :: p1 :: s} \quad \text{(Swap)}$$

$$\frac{(v, c) \rightarrow_0 (v', c')}{(i, v, C[c]) :: s \rightarrow (i, v', C[c']) :: s} \quad \text{(Reduce)}$$

$$\frac{}{\begin{array}{c}(i_1, v_1, C_1[i_2!e_1]) :: (i_2, v_2, C_2[i_1?x]) :: s \rightarrow \\ (i_1, v_1, C_1[\text{skip}]) :: (i_2, v_2[x \mapsto [\![e_1]\!]v_1], C_2[\text{skip}]) :: s\end{array}} \quad \text{(Comm)}$$

The Swap and the Skip rules allow you to permute the processes in the list. The Reduce rule reduces the first process in the list of processes. The Comm rule defines the rule for synchronous communiation. When the first process i_1 is willing to send to the second process i_2, and at the same time, the i_2 is willing to receive from i_1, then i_2 receives the message from i_1. Observe that in order to apply the Comm rule, the sender and the receiver need to be at the head of the list of processes in that order. This can be achieved with repeated application of Swap and Skip rules.

For example, consider the state $p :: r :: s :: []$, where $r = (i_2, v_2, C_2[i_2?x])$ and $s = (i_1, v_1, C_1[i_2!e_1])$. We can make the communication go through by,

$$\begin{array}{rl} & p :: r :: s :: [] \\ \xrightarrow{\text{Skip,Swap}} & p :: s :: r :: [] \\ \xrightarrow{\text{Swap}} & s :: p :: r :: [] \\ \xrightarrow{\text{Skip,Swap}} & s :: r :: p :: [] \end{array}$$

And now the communication rule can be applied.

1.1 Your Goal

Encode this operational semantics in Coq and prove the theorems in the given in the source file `final.v`. There is a challenge problem at the end which you may

attempt.

2 Inplace map in Pulse (30 points)

Here is the implementation of inplace map of an array in OCaml:

```
let map_inplace f a =
  for i = 0 to Array.length a - 1 do
    a.(i) <- f a.(i)
  done
```

Complete the specification (5 points) and the implementation (25 points) of the function in the file `InplaceMap.fst`. The specification must be the strongest one that fully captures the behaviour of the map inplace function.