MIT 6.887 FRAP – Problem Set 6

In this problem set, you are required to define a formal semantics for a given language for distributed systems, with message passing over an unreliable network. We also ask you to prove correctness of a particular example program that we provide.

Language Syntax

Constants	c	$\in$	$\mathbb{N}$
Variables	x	$\in$	Strings
PIDs	p	$\in$	$\mathbb{N}$
Arith	e	::=	$c \mid x \mid e == e$
Commands	i	::=	$x := e$
		$\mid$	$\mathsf{if}\ e\ \mathsf{then}\ \vec{i}\ \mathsf{else}\ \vec{i}$
		$\mid$	$\mathsf{while}\ e\ \vec{i}$
		$\mid$	$\mathsf{send}\ \vec{e}\ \mathsf{to}\ p$
		$\mid$	$\mathsf{receive}\ \vec{x}$
		$\mid$	$\mathsf{ret}\ e$
Program	g	::=	$\vec{i}$
System	s	::=	$\overrightarrow{(p, g)}$

Our core language is defined as above, where the starter code includes the formal syntax definition. For some syntactic class x, we write $\vec{x}$ for lists of x's. We have seen Arith expressions several times in class; we cut some unnecessary operations and instead add a Boolean equality ($e == e$) for implementing the example program below. You should keep our syntax definitions intact, but you are free to add any new syntactic features as new constructors, e.g. other binary operators.

Commands include typical imperative-language ingredients like assignments ($x := e$), branches ($\mathsf{if}\ e\ \mathsf{then}\ c\ \mathsf{else}\ c$), and iterations ($\mathsf{while}\ e\ c$). In addition to these features, we define commands for communication by message passing. ($\mathsf{send}\ \vec{e}\ \mathsf{to}\ p$) sends values $\vec{e}$ (where each e is evaluated) to the process that has p as its PID. ($\mathsf{receive}\ \vec{x}$) receives a message $\vec{c}$ from the network, assigning its components to $\vec{x}$, respectively, which must have the same length. A process terminates with a result code by ($\mathsf{ret}\ e$).

A program is defined as a list of commands, and a system is defined as a list of (PID, Program) pairs. We require (assume) any systems to avoid duplicate PIDs (process IDs).

Operational Semantics

Your first task is to formally define operational semantics for the language. You will almost certainly want to use a small-step semantics, since we are concerned here with just the sorts of concurrent, nondeterministic, nonterminating programs that motivate abandoning big-step semantics.

While semantics for imperative features are defined naturally, you may wonder how to define semantics for message passing *over an unreliable network*. We would like to give enough freedom to define such semantics, but your semantics should properly reflect the following *unreliability requirements*: 1) messages are sometimes sent to wrong recipients and 2) messages are sometimes not delivered in the order they are sent. To formalize these requirements, it may be helpful to replace the word "sometimes" with "nondeterministically." When you define your step relation for `system` by `Inductive`, it may contain some constructors specifically to model network unreliability. And feel free to add other ways in which your network is unreliable! That will make your job harder in the required program-correctness proof, but it would also make the result more realistic. (Some kinds of unreliability, like random corruption of messages, would render the theorem unprovable, so there are limits on how realistic you should get!)

Problem 1. Define operational semantics for the core language. Add a comment containing "`Problem 1`" right before your semantics definition.

A program to prove: simple echo client/server

```
Definition client_pid := 0.
Definition server_pid := 1.
Definition request_to_server := 1.
Definition response_from_server := 2.

Definition client := [
  "done" <- 0;
  while ("done" == 0) [
    send [Const request_to_server; Const n] to server_pid;
    receive ["code"; "res"];
    if_ "code" == response_from_server then [
      "done" <- 1
    ] else []
  ];
  ret "res"
]%cmd.

Definition server := [
  while (1) [
    receive ["code"; "n"];
    if_ "code" == request_to_server then [
      send [Const response_from_server; Var "n"] to client_pid
    ] else []
  ];
  ret 0
]%cmd.

Definition echo_client_server : pgms :=
  [(client_pid, client); (server_pid, server)].
```

Based on the language syntax and semantics, the next task is to implement a certain system and prove its correctness. The target system consists of two

programs, `client` and `server`, where implementations are given as above. Thanks to the notation mechanism in Coq, we can implement the system with a fancy syntax.

In our echo system, `client` first tries to send a message to `server` which contains the value `n`. When `server` receives the message, it sends a new message to `client` which contains the value from the message that `client` sent. After `client` receives the message from `server`, it returns the value from the message. Now here is the correctness theorem to prove.

Problem 2. Prove that, in this combined distributed system, *if* `client` *terminates, then it always returns* `n`. Add a comment containing "`Problem 2`" right before your main theorem.

A more naive design would fail to meet the spec, due to our unreliable network. As seen in the implementation, we had to send messages repeatedly and tag each message with the intended recipient. You may have to consider carefully why such tricks make the system correct. (Actually, some of them could be removed and the program would still be correct in the sense that we ask you to prove here, though the program could then fail to terminate even when the network seems to be cooperating decently well!)