CS 599 A1: Assignment 4

Due: Thursday, April 3, 2025

Total: 100 pts

Ankush Das

- This assignment is due on the above date and it must be submitted electronically on Gradescope.
- There are two parts to this assignment: a written part and a programming part.
- For the written part, please use the template provided on the course webpage to typeset your assignment and include your name and BU ID in the Author section (above).
- The written part must be submitted as a PDF in the Gradescope assignment called "Assignment 4".
- The programming part must be submitted in the Gradescope assignment called "Assignment 4 Programming". You should hand in one file named `asgn4.rast` containing the solutions to the coding assignment.
- You will be provided a `tex` file, named `asgn4.tex`. It contains an environment called `solution`. Please enter your solutions inside these environments.

Programming with Session Types

In this assignment, you will be using the Rast (short for Resource-Aware Session Types) programming language to implement session-typed processes. Appendix A provides a description of the language.

Repository Set Up

First, go to the following Bitbucket Repository: `https://bitbucket.org/fpfenning/rast/src/master/rast/` that contains the Rast source code. Next, follow the steps below:

- Install MLton: Rast is implemented in SML and MLton is a compiler for SML. Follow the steps here: `http://mlton.org/`. For Mac users, simply running `brew install mlton` in your terminal would work too.
- Next, clone the Rast repository. The easiest way to do this is via HTTPS. Issue the following command: `git clone https://bitbucket.org/fpfenning/rast.git`
- Finally, we need to build the source and generate the Rast executable. Navigate to the `src` directory: `cd rast/rast/src` and run `make`.
- To check if the `rast` executable is working correctly, run the following command on your terminal: `./rast ../examples/arith.rast`. If you see `% success` at the end, then you have set up Rast correctly.

1 Odd and Even Numbers [45 pts]

In the previous homework, we looked at binary numbers. In this homework, we will look at unary natural numbers represented as

```
type nat = +{succ : nat, zero : 1}
```

We also introduce the type of even and odd natural numbers defined mutually recursively as follows:

```
type even = +{succ : odd, zero : 1}
type odd = +{succ : even}
```

An even number can either be 0 or the successor of an odd number. Similarly, an odd number can only be a successor of an even number.

As a simple property, an even number can only be typed as `even`. For example, consider the process `two` that represents the number 2. The process is written as

```
decl two : . |- (x : nat)
proc x <- two = x.succ; x.succ; x.zero; close x
```

We can also write the same process with an `even` type:

```
decl two' : . |- (x : even)
proc x <- two' = x.succ; x.succ; x.zero; close x
```

But if we try to write the same process with an `odd` type, we will encounter a type error.

```
decl two'' : . |- (x : odd)
proc x <- two'' = x.succ; x.succ; x.zero; close x
```

You are welcome to write the above programs in a `.rast` file and confirm the observations above. This serves as a proof that 2 is an even number and 2 is not an odd number.

In the following set of problems, you will be writing programs that effectively serve as proofs about even and odd numbers. You are welcome to define and use helper processes. Use the file asgn4.rast provided in the assignment and write the following programs in that file. Make sure to run them through the Rast type checker using the command `./rast asgn4.rast` to make sure that the program is type correct.

Problem 1 (10 pts) *Define a process called* `double` *that has the following signature:*

```
decl double: (x : odd) |- (y : even)
```

The process doubles the input odd number channel x to produce y. This program serves as a proof that the double of an odd number is an even number.

Problem 2 (10 pts) *Define a process called* `plus` *that has the following signature:*

```
decl plus: (x : odd) (y : even) |- (z : odd)
```

The process uses an odd channel x and an even channel y and adds them to produce z. This program serves as a proof that the sum of an odd and even number is odd.

Problem 3 (15 pts) *Define a process called* `even_or_odd` *that has the following signature:*

```
decl even_or_odd: (x : nat) |- (y : +{E : even, O : odd})
```

The process uses a natural number channel x as input. It determines whether x is even or odd. If x is even, the process sends the label E on y and then produces the input number x on channel y. On the other hand, if x is odd, the process sends the label O on y and then produces the input number x on channel y. In both cases, the input number is produced on the output. This program serves as a proof that every natural number is either odd or even.

Problem 4 (10 pts) *Test the correctness of the* `even_or_odd` *process. Define two processes that represent numbers two and three respectively with the following signature:*

```
decl two : . |- (c : nat)
decl three : . |- (c : nat)
```

Now, define the process below by first spawning a process `two` *and then calling* `even_or_odd` *on it.*

```
decl two_is_even : . |- (c : +{E : even, O : odd})
```

Similarly, define the process below by doing the same for process `three`*.*

```
decl three_is_odd : . |- (c : +{E : even, O : odd})
```

Now, execute these 2 processes as follows:

```
exec two_is_even
exec three_is_odd
```

If you have implemented this correctly, the output should be:

```
exec two_is_even
c = E ; succ ; succ ; zero ; close
exec three_is_odd
c = O ; succ ; succ ; succ ; zero ; close
```

2 Text Editor [25 pts]

We will implement a prototype text editor using session types. This editor supports the following operations:

- moveL: moving the cursor one step left
- moveR: moving the cursor one step right
- ins: inserting a character at the cursor and moving the cursor one step right
- del: deleting a character at the cursor and moving the cursor one step left

The type to represent these operations is called cmd (akin to issuing commands to the editor):

```
type cmd = &{ moveL : cmd,
              moveR : cmd,
              ins : char -o cmd,
              del : +{none : cmd,
                      some : char * cmd}}
```

Here, the type char represents characters. However, your program is parametric in this type: you would never need to know the type definition of char. Your program would work for any definition of the char type. (This is also referred to as polymorphism)

We will implement this editor using two stacks: one that represents the characters on the left of the cursor and the other represents the characters on the right of the cursor. The type of stack is defined as (slight modification: it does not terminate in the none branch):

```
type stack = &{ ins : char -o stack,
                del : +{none : stack,
                        some : char * stack}}
```

Problem 5 (25 pts) *Write a process called* `editor` *with the following signature:*

```
decl editor : (l : stack) (r : stack) |- (c : cmd)
```

3 Proving With Session Types [30 pts]

Problem 6 (30 pts) *Complete the remaining cases (spawning, forwarding, and* **1***) of progress and preservation for session types.*

[Preservation Theorem] Consider a well-typed configuration $\mathcal{C}$ such that $\Delta_1 \vDash \mathcal{C} :: \Delta_2$. If $\mathcal{C} \mapsto \mathcal{C}'$, then $\Delta_1 \vDash \mathcal{C}' :: \Delta_2$.

[Progress Theorem] Consider a well-typed configuration $\mathcal{C}$ such that $\Delta_1 \vDash \mathcal{C} :: \Delta_2$. Either $\mathcal{C} \mapsto \mathcal{C}'$ or $\mathcal{C}$ poised*.*

A Syntax, Type System, and Semantics for Session Types

Syntax

$$\begin{array}{rl}\text{Expressions} & P ::= x.\mathsf{k}\,;\,P \mid \mathsf{case}\ x\ (\ell \Rightarrow P_\ell)_{\ell\in L} \mid y \leftarrow \mathsf{recv}\ x\,;\,P \mid \mathsf{send}\ x\ y\,;\,P \mid \mathsf{wait}\ x\,;\,P \mid \mathsf{close}\ x \\ & \quad\mid\ x \leftrightarrow y \mid x \leftarrow f\ \overline{y}\,;\,P \\ \text{Types} & A, B ::= \oplus\{\ell : A_\ell\}_{\ell\in L} \mid \&\{\ell : A_\ell\}_{\ell\in L} \mid A \otimes B \mid A \multimap B \mid \mathbf{1}\end{array}$$

Type System

$$\frac{(k \in L) \qquad \Delta \vdash P :: (x : A_k)}{\Delta \vdash (x.k\,;\,P) :: (x : \oplus\{\ell : A_\ell\}_{\ell\in L})}\ \oplus\text{R} \qquad \frac{(\forall \ell \in L) \qquad \Delta, x : A_\ell \vdash Q_\ell :: (z : C)}{\Delta, x : \oplus\{\ell : A_\ell\}_{\ell\in L} \vdash (\mathsf{case}\ x\ (\ell \Rightarrow Q_\ell)_{\ell\in L}) :: (z : C)}\ \oplus\text{L}$$

$$\frac{(\forall \ell \in L) \qquad \Delta \vdash P_\ell :: (x : A_\ell)}{\Delta \vdash (\mathsf{case}\ x\ (\ell \Rightarrow P_\ell)_{\ell\in L}) :: (x : \&\{\ell : A_\ell\}_{\ell\in L})}\ \&\text{R} \qquad \frac{(k \in L) \qquad \Delta, x : A_k \vdash Q :: (z : C)}{\Delta, x : \&\{\ell : A_\ell\}_{\ell\in L} \vdash (x.k\,;\,Q) :: (z : C)}\ \&\text{L}$$

$$\frac{\Delta \vdash P :: (x : B)}{\Delta, y : A \vdash (\mathsf{send}\ x\ y\,;\,P) :: (x : A \otimes B)}\ \otimes\text{R} \qquad \frac{\Delta, y : A, x : B \vdash Q :: (z : C)}{\Delta, x : A \otimes B \vdash (y \leftarrow \mathsf{recv}\ x\,;\,Q) :: (z : C)}\ \otimes\text{L}$$

$$\frac{\Delta, y : A \vdash P :: (x : B)}{\Delta \vdash (y \leftarrow \mathsf{recv}\ x\,;\,P) :: (x : A \multimap B)}\ \multimap\text{R} \qquad \frac{\Delta, x : B \vdash Q :: (z : C)}{\Delta, x : A \multimap B, y : A \vdash (\mathsf{send}\ x\ y\,;\,Q) :: (z : C)}\ \multimap\text{L}$$

$$\frac{}{\cdot \vdash (\mathsf{close}\ x) :: (x : \mathbf{1})}\ \mathbf{1}\text{R} \qquad \frac{\Delta \vdash Q :: (z : C)}{\Delta, x : \mathbf{1} \vdash (\mathsf{wait}\ x\,;\,Q) :: (z : C)}\ \mathbf{1}\text{L} \qquad \frac{}{x : A \vdash (y \leftrightarrow x) :: (y : A)}\ \mathsf{id}$$

$$\frac{\mathsf{decl}\ f : \overline{y' : A'} \vdash (x : A) \in \Sigma \qquad \Delta, x : A \vdash Q :: (z : C)}{\Delta, \overline{y : A'} \vdash (x \leftarrow f\ \overline{y}\,;\,Q) :: (z : C)}\ \mathsf{def}$$

Semantics

$(\oplus S)$	$\mathsf{proc}(c, c.k\,;\,P) \mapsto \mathsf{proc}(c', [c'/c]P), \mathsf{msg}(c, c.k\,;\,c \leftrightarrow c')$	$(c'$ fresh$)$
$(\oplus C)$	$\mathsf{msg}(c, c.k\,;\,c \leftrightarrow c'), \mathsf{proc}(d, \mathsf{case}\ c\ (\ell \Rightarrow Q_\ell)_{\ell\in L}) \mapsto \mathsf{proc}(d, [c'/c]Q_k)$	
$(\&S)$	$\mathsf{proc}(d, c.k\,;\,Q) \mapsto \mathsf{msg}(c', c.k\,;\,c' \leftrightarrow c), \mathsf{proc}(d, [c'/c]Q)$	$(c'$ fresh$)$
$(\&C)$	$\mathsf{proc}(c, \mathsf{case}\ c\ (\ell \Rightarrow Q_\ell)_{\ell\in L}), \mathsf{msg}(c', c.k\,;\,c' \leftrightarrow c) \mapsto \mathsf{proc}(c', [c'/c]Q_k)$	
$(\otimes S)$	$\mathsf{proc}(c, \mathsf{send}\ c\ e\,;\,P) \mapsto \mathsf{proc}(c', [c'/c]P), \mathsf{msg}(c, \mathsf{send}\ c\ e\,;\,c \leftrightarrow c')$	$(c'$ fresh$)$
$(\otimes C)$	$\mathsf{msg}(c, \mathsf{send}\ c\ e\,;\,c \leftrightarrow c'), \mathsf{proc}(d, x \leftarrow \mathsf{recv}\ c\,;\,Q) \mapsto \mathsf{proc}(d, [c', e/c, x]Q)$	
$(\multimap S)$	$\mathsf{proc}(d, \mathsf{send}\ c\ e\,;\,Q) \mapsto \mathsf{msg}(c', \mathsf{send}\ c\ e\,;\,c' \leftrightarrow c), \mathsf{proc}(d, [c'/c]Q)$	$(c'$ fresh$)$
$(\multimap C)$	$\mathsf{proc}(c, x \leftarrow \mathsf{recv}\ c), \mathsf{msg}(c', \mathsf{send}\ c\ e\,;\,c' \leftrightarrow c) \mapsto \mathsf{proc}(c', [c', d/c, x]P)$	
$(\mathbf{1}S)$	$\mathsf{proc}(c, \mathsf{close}\ c) \mapsto \mathsf{msg}(c, \mathsf{close}\ c)$	
$(\mathbf{1}C)$	$\mathsf{msg}(c, \mathsf{close}\ c), \mathsf{proc}(d, \mathsf{wait}\ c\,;\,Q) \mapsto \mathsf{proc}(d, Q)$	
$(\mathsf{def}C)$	$\mathsf{proc}(d, x \leftarrow P_x\ \overline{y}\,;\,Q_x) \mapsto \mathsf{proc}(c, [c/x]P_x), \mathsf{proc}(d, [c/x]Q_x)$	$(c$ fresh$)$
(id^+C)	$\mathsf{msg}(d, M), \mathsf{proc}(c, c \leftrightarrow d) \mapsto \mathsf{msg}(c, [c/d]M)$	
(id^-C)	$\mathsf{proc}(c, c \leftrightarrow d), \mathsf{msg}(e, M_c) \mapsto \mathsf{msg}(e, [d/c]M_c)$	