CS 599 D1: Assignment 3

Due: Wednesday, March 20, 2024

Total: 50 pts

Ankush Das

- This assignment is due on the above date and it must be submitted electronically on Gradescope. Please create an account on Gradescope, if you haven't already done so.
- Please use the template provided on the course webpage to typeset your assignment and please include your name and BU ID in the Author section (above).
- Although it is not recommended, you can submit handwritten answers that are scanned as a PDF and clearly legible.
- You should hand in one file, named as ⟨first-name⟩_⟨last-name⟩_⟨BU-ID⟩_asgn3.pdf containing the solutions to the problems below.
- You will be provided a tex file, named asgn3.tex. It contains an environment called solution. Please enter your solutions inside these environments.

Session-Typed Programming

This assignment will be all about programming in the linear session-typed programming language that we have introduced in the course. The language is defined here for convenience.

Syntax

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

Type System

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

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

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

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

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

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

1 Binary Numbers [10 pts]

The first problem involves programming with binary numbers. Binary numbers are defined using the following type:

```
type bin = +{b1 : bin, b0 : bin, e : 1}
```

The numbers is represented such that the least significant bit is sent first. So, for example, the number $2 = (10)_2$ is represented by first sending $b0$, then $b1$, and finally e. Hence, the process is written as

```
decl three : . |- (x : bin)
proc x <- three = x.b0; x.b1; x.e; close x
```

Intuitively, the bits are sent in the reverse order, i.e., the rightmost bit is sent first and the leftmost bit is sent last, and then e is sent to indicate the number is completely transmitted. (You will see this representation turns out to be the most convenient!)

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

```
decl increment: (x : bin) |- (y : bin)
```

The process uses a binary number x as input and produces a binary number y such that $y = x + 1$.

2 Lists [40 pts]

We will get some more programming experience with lists. Recall the list type:

```
type listA = +{cons : A * listA, nil : 1}
type listB = +{cons : B * listB, nil : 1}
```

We already looked at standard list functions such as append and reverse. In this section, we will do higher-order programming like map and fold. To define a `map` process, we need a $\texttt{mapper}_{\texttt{AB}}$ type that performs the mapping from elements of type A to elements of type B.

```
type mapperAB = &{next : A -o B * mapperAB,
                  done : 1}
```

The $\texttt{mapper}_{\texttt{AB}}$ type can either receive the **next** message, followed by an element of type A and produces an element of type B. Or it can receive the **done** message and terminate.

Problem 2 (15 pts) *First, define a* `map` *process with the following signature:*

```
decl map : (a : listA), (m : mapperAB) |- (b : listB)
```

The process uses a list `a` : `listA` *and a mapper to produce a list* `b` : `listB`. *For each element in* `a`*, the* `map` *process sends the element of type A to the mapper, receives an element of type B which is then sent on the offered channel* `b`.

Next, we will complete this story for binary numbers. We will define a process that increments each element in a list of binary numbers. First, the type of list of binary numbers is defined as:

```
type binlist = +{cons : bin * binlist, nil : 1}
```

The mapper type will then be defined as follows:

```
type binmapper = &{next : bin -o bin * binmapper,
                   done : 1}
```

Problem 3 (15 pts) *Define a process called* `mapinc` *with the following signature:*

```
decl mapinc : . |- (m : binmapper)
```

This process receives binary numbers from m, increments them and sends them back to m. Remember that you have already defined the `increment` *process for binary numbers. Be sure to call it to actually increment the binary number!*

Problem 4 (10 pts) *Finally, define a process called* `listinc` *with the following signature:*

```
decl listinc : (a : binlist) |- (b : binlist)
```

This process uses a list `a : binlist` *and increments each element to produce list* `b : binlist`*. Instead of incrementing the elements on the list directly, use the* `map` *and* `mapinc` *processes you defined earlier! You can assume the following type for the* `map` *process:*

```
decl map : (a : binlist), (m : binmapper) |- (b : binlist)
```