Lecture Notes on
Queues and Stacks

15-816: Substructural Logics
Frank Pfenning

Lecture 9
Tuesday, September 27, 2016

In the last lecture we introduced lists with arbitrary elements and wrote ordered programs for *nil* (the empty list), *cons* (adding an element to the head of a list) and *append* to append two lists. The representation was in the form of an *internal choice*

$$\mathsf{list}_A = \oplus\{\mathsf{cons} : A \bullet \mathsf{list}_A, \mathsf{nil} : \mathbf{1}\}$$

We might think of this as the usual functional data structure of lists, but we should keep in mind that it is really just an interface specification for processes. It does not imply any particular representation.

Today, we will look at a data structure in which we can insert and delete channels of arbitrary type. The interface is different because it is in the form of an *external choice*, more in the style of object-oriented programming or signatures in module systems for functional languages.

1 Storing Channels

Here is our simple interface to a storage service for channels:

$$\begin{array}{ll}\mathsf{store}_A = \&\{ & \mathsf{ins} : A \setminus \mathsf{store}_A, \\ & \mathsf{del} : \oplus\{\mathsf{none} : \mathbf{1}, \mathsf{some} : A \bullet \mathsf{store}_A\}\}\end{array}$$

Using our operational interpretation, we can read this as follows:

> *A* store *for channels of type A offers a client a choice between insertion (label* ins*) and deletion (label* del*).*
> *When inserting, the clients sends a channel of type A which is added*

> *to the store.*
> *When deleting, the store responds with the label* none *if there are no elements in the store and terminates, or with the label* some*, followed by an element.*
> *When an element is actually inserted or deleted the provider of the storage service then waits for the next input (again, either an insertion or deletion).*

In this reading we have focused on the operations, and intentionally ignored the restrictions order might place on the use of the storage service. Hopefully, this will emerge as we write the code and analyze what the restrictions might mean.

First, we have to be able to create an empty store. We will write the code in stages, because I believe it is much harder to understand the final program than it is to follow its construction.

$$\begin{aligned}\mathsf{store}_A = \&\{\ & \mathsf{ins} : A \setminus \mathsf{store}_A, \\ & \mathsf{del} : \oplus\{\mathsf{none} : \mathbf{1}, \mathsf{some} : A \bullet \mathsf{store}_A\}\}\end{aligned}$$

First, the header of the process definition.

$$\begin{array}{l} \cdot \vdash \mathit{empty} :: (s : \mathsf{store}_A) \\ s \leftarrow \mathit{empty} = \ldots \end{array}$$

Because a store_A is an external choice, we begin with a case construct, branching on the received label.

$$\begin{array}{lll} \cdot \vdash \mathit{empty} :: (s : \mathsf{store}_A) & & \\ s \leftarrow \mathit{empty} = \mathsf{case}\ s\ (\mathsf{ins} \Rightarrow \ldots & \% & \cdot \vdash s : A \setminus \mathsf{store}_A \\ \qquad\qquad\qquad\qquad \mid \mathsf{del} \Rightarrow \ldots & \% & \cdot \vdash s : \oplus\{\mathsf{none} : \mathbf{1}, \mathsf{some} : A \bullet \mathsf{store}_A\} \\ \qquad\qquad\qquad\qquad) & & \end{array}$$

The case of deletion is actually easier: since this process represents an empty store, we send the label none and terminate.

$$\begin{array}{lll} \cdot \vdash \mathit{empty} :: (s : \mathsf{store}_A) & & \\ s \leftarrow \mathit{empty} = \mathsf{case}\ s\ (\mathsf{ins} \Rightarrow \ldots & \% & \cdot \vdash s : A \setminus \mathsf{store}_A \\ \qquad\qquad\qquad\qquad \mid \mathsf{del} \Rightarrow s.\mathsf{none}\ ;\ \mathsf{close}\ s) & & \end{array}$$

In the case of an insertion, the type dictates that we receive a channel of type A which we call x. It is added at the left end of the antecedents. Since they are actually none, both $A \setminus \mathsf{store}_A$ and $\mathsf{store}_A \mathbin{/} A$ would behave the same way here.

```
$\cdot \vdash empty :: (s : \mathsf{store}_A)$
$s \leftarrow empty = \mathsf{case}\ s\ (\mathsf{ins} \Rightarrow$                %  $\cdot \vdash s : A \setminus \mathsf{store}_A$
                            $x \leftarrow \mathsf{recv}\ s$ ;          %  $x{:}A \vdash s : \mathsf{store}_A$
                            ...
                    $\mid \mathsf{del} \Rightarrow s.\mathsf{none}$ ; $\mathsf{close}\ s)$
```

At this point it seems like we are stuck. We need to start a process implementing a store with *one* element, but so far we just writing the code for an empty store. We need to define a process *elem*

$$(x{:}A)\ (t{:}\mathsf{store}_A)\ \vdash elem :: (s : \mathsf{store}_A)$$

which holds an element $x{:}A$ and also another store $t{:}\mathsf{store}_A$ with further elements. In the singleton case, t will then be the empty store. Therefore, we first make a recursive call to create another empty store, calling it n for *none*.

```
$\cdot \vdash empty :: (s : \mathsf{store}_A)$
$s \leftarrow empty = \mathsf{case}\ s\ (\mathsf{ins} \Rightarrow x \leftarrow \mathsf{recv}\ s$ ;          %  $x{:}A \vdash s : \mathsf{store}_A$
                            $n \leftarrow empty$ ;            %  $(x{:}A)\ (n{:}\mathsf{store}_A) \vdash s : \mathsf{store}_A$
                            ...
                    $\mid \mathsf{del} \Rightarrow s.\mathsf{none}$ ; $\mathsf{close}\ s)$

$(x{:}A)\ (t{:}\mathsf{store}_A)\ \vdash elem :: (s : \mathsf{store}_A)$
$s \leftarrow elem \leftarrow x\ t = \ldots$
```

Postponing the definition of *elem* for now, we can invoke *elem* to create a singleton store with just x, calling the resulting channel e. This call will consume x and n, leaving e as the only antecedent.

```
$\cdot \vdash empty :: (s : \mathsf{store}_A)$
$s \leftarrow empty = \mathsf{case}\ s\ (\mathsf{ins} \Rightarrow x \leftarrow \mathsf{recv}\ s$ ;          %  $x{:}A \vdash s : \mathsf{store}_A$
                            $n \leftarrow empty$ ;            %  $(x{:}A)\ (n{:}\mathsf{store}_A) \vdash s : \mathsf{store}_A$
                            $e \leftarrow elem \leftarrow x\ n$ ;    %  $e{:}\mathsf{store}_A \vdash s : \mathsf{store}_A$
                            ...
                    $\mid \mathsf{del} \Rightarrow s.\mathsf{none}$ ; $\mathsf{close}\ s)$

$(x{:}A)\ (t{:}\mathsf{store}_A)\ \vdash elem :: (s : \mathsf{store}_A)$
$s \leftarrow elem \leftarrow x\ t = \ldots$
```

At this point we can implement s by e (the singleton store), which is just an application of the identity rule.

```
$\cdot \vdash \mathit{empty} :: (s : \mathsf{store}_A)$
$s \leftarrow \mathit{empty} = \mathsf{case}\ s\ (\mathsf{ins} \Rightarrow x \leftarrow \mathsf{recv}\ s\ ;$    % $(x{:}A) \vdash s : \mathsf{store}_A$
                                $n \leftarrow \mathit{empty}\ ;$              % $(x{:}A)\ (n{:}\mathsf{store}_A) \vdash s : \mathsf{store}_A$
                                $e \leftarrow \mathit{elem} \leftarrow x\ n$        % $e{:}\mathsf{store}_A \vdash s : \mathsf{store}_A$
                                $s \leftarrow e$
                            $\mid \mathsf{del} \Rightarrow s.\mathsf{none}\ ;\ \mathsf{close}\ s)$

$(x{:}A)\ (t{:}\mathsf{store}_A)\ \vdash \mathit{elem} :: (s : \mathsf{store}_A)$
$s \leftarrow \mathit{elem} \leftarrow x\ t = \ldots$
```

It remains to write the code for the process holding an element of the store. We suggest you reconstruct or at least read it line by line the way we developed the definition of *empty*, but we will not break it out explicitly into multiple steps. However, we will still give the types after each interaction. For easy reference, we repeat the type definition for store_A.

$$\begin{array}{l} \mathsf{store}_A = \&\{\ \mathsf{ins} : A \setminus \mathsf{store}_A, \\ \qquad\qquad \mathsf{del} : \oplus\{\mathsf{none} : \mathbf{1}, \mathsf{some} : A \bullet \mathsf{store}_A\}\} \end{array}$$

$(x{:}A)\ (t{:}\mathsf{store}_A)\ \vdash \mathit{elem} :: (s : \mathsf{store}_A)$

```
1  $s \leftarrow \mathit{elem} \leftarrow x\ t =$
2      case $s$ (ins $\Rightarrow y \leftarrow$ recv $s$ ;        % $(y{:}A)\ (x{:}A)\ (t{:}\mathsf{store}_A) \vdash s : \mathsf{store}_A$
3                  $t$.ins ;                    % $(y{:}A)\ (x{:}A)\ (t{:}A \setminus \mathsf{store}_A) \vdash s : \mathsf{store}_A$
4                  send $t\ x$ ;                % $(y{:}A)\ (t{:}\mathsf{store}_A) \vdash s : \mathsf{store}_A$
5                  $r \leftarrow \mathit{elem} \leftarrow y\ t$ ;     % $r{:}\mathsf{store}_A \vdash s : \mathsf{store}_A$
6                  $s \leftarrow r$
7             | del $\Rightarrow s$.some ;          % $(x{:}A)\ (t{:}\mathsf{store}_A) \vdash s : A \bullet \mathsf{store}_A$
8                  send $s\ x$ ;                % $t{:}\mathsf{store}_A \vdash s : \mathsf{store}_A$
9                  $s \leftarrow t$)
```

A few notes on this code. Look at the type at the end of the *previous* line to understand the next line.

- In line 2, we add $y{:}A$ at the left end of the context since $s : A \setminus \mathsf{store}_A$.
- In line 4, we can only pass x to t but not y, due restrictions of $\setminus L^*$.
- In line 5, y and t are in the correct order to call elem recursively.
- In line 8, we can pass x along s since it is at the left end of the context.

How does this code behave? Assume we have a store s holding elements x_1 and x_2 it would look like

$$\mathsf{proc}(s, s \leftarrow \mathit{elem} \leftarrow x_1\ t_1)\quad \mathsf{proc}(t_1, t_1 \leftarrow \mathit{elem} \leftarrow x_2\ t_2)\quad \mathsf{proc}(t_2, t_2 \leftarrow \mathit{empty})$$

where we have indicated the code executing in each process without unfolding the definition. If we insert an element along s (by sending ins and then a new y) then the process $s \leftarrow \mathit{elem} \leftarrow x_1\ t_1$ will insert x_1 along t_1 and then, in two steps, become $s \leftarrow \mathit{elem} \leftarrow y\ t_1$. Now the next process will pass x_2 along t_2 and hold on to x_1, and finally the process holding no element will spawn a new one (t_3) and itself hold on to x_2.

$$\begin{array}{r} \mathsf{proc}(s, s \leftarrow \mathit{elem} \leftarrow y\ t_1)\quad \mathsf{proc}(t_1, t_1 \leftarrow \mathit{elem} \leftarrow x_1\ t_2) \\ \mathsf{proc}(t_2, t_2 \leftarrow \mathit{elem} \leftarrow x_2\ t_3)\quad \mathsf{proc}(t_3, t_3 \leftarrow \mathit{empty}) \end{array}$$

If we next delete an element, we will get y back and the store will effectively revert to its original state, with some (internal) renaming.

$$\mathsf{proc}(s, s \leftarrow \mathit{elem} \leftarrow x_1\ t_2) \quad \mathsf{proc}(t_2, t_2 \leftarrow \mathit{elem} \leftarrow x_2\ t_3) \quad \mathsf{proc}(t_3, t_3 \leftarrow \mathit{empty})$$

In essence, the store behaves like a *stack*: the most recent element we have inserted will be the first one deleted. If you carefully look through the intermediate types in the *elem* process, it seems that this behavior is forced. We conjecture that any implementation of the store interface we have given will behave like a stack or might at some point not respond to further messages. We do not yet have the means to carry out such a proof. Some related prior work might provide hints on how this might be proved using parametricity [Rey83, CPPT13].[1]

[1]If I or someone else in the class can prove or refute this conjecture, we may return to it in a future lecture.

2 Tail Calls

Let's look again at the two pieces of code we have written.

$$\begin{array}{ll}\mathsf{store}_A = \&\{ & \mathsf{ins} : A \setminus \mathsf{store}_A, \\ & \mathsf{del} : \oplus\{\mathsf{none} : \mathbf{1}, \mathsf{some} : A \bullet \mathsf{store}_A\}\}\end{array}$$

$\cdot \vdash \textit{empty} :: (s : \mathsf{store}_A)$

```
1   $s \leftarrow \textit{empty} =$
2     case $s$ (ins $\Rightarrow$ $x \leftarrow$ recv $s$ ;        % $(x{:}A) \vdash s : \mathsf{store}_A$
3                  $n \leftarrow \textit{empty}$ ;           % $(x{:}A)\ (n{:}\mathsf{store}_A) \vdash s : \mathsf{store}_A$
4                  $e \leftarrow \textit{elem} \leftarrow x\ n$        % $e{:}\mathsf{store}_A \vdash s : \mathsf{store}_A$
5                  $s \leftarrow e$
6              | del $\Rightarrow$ $s$.none ; close $s$)
```

$(x{:}A)\ (t{:}\mathsf{store}_A)\ \vdash \textit{elem} :: (s : \mathsf{store}_A)$

```
7   $s \leftarrow \textit{elem} \leftarrow x\ t =$
8       case $s$ (ins $\Rightarrow$ $y \leftarrow$ recv $s$ ;        % $(y{:}A)\ (x{:}A)\ (t{:}\mathsf{store}_A) \vdash s : \mathsf{store}_A$
9                    $t$.ins ;                   % $(y{:}A)\ (x{:}A)\ (t{:}A \setminus \mathsf{store}_A) \vdash s : \mathsf{store}_A$
10                   send $t\ x$ ;               % $(y{:}A)\ (t{:}\mathsf{store}_A) \vdash s : \mathsf{store}_A$
11                   $r \leftarrow \textit{elem} \leftarrow y\ t$ ;      % $r{:}\mathsf{store}_A \vdash s : \mathsf{store}_A$
12                   $s \leftarrow r$
13             | del $\Rightarrow$ $s$.some ;            % $(x{:}A)\ (t{:}\mathsf{store}_A) \vdash s : A \bullet \mathsf{store}_A$
14                   send $s\ x$ ;               % $t{:}\mathsf{store}_A \vdash s : \mathsf{store}_A$
15                   $s \leftarrow t$)
```

empty starts two new processes, in lines 3 and 4 and then terminates in line 5 by forwarding. *elem* spawns only one new process, in line 11, and then terminates in line 12 by forwarding. Intuitively, spawning a new process and then immediately forwarding to this process is wasteful, especially if process creation is an expensive operation.

It would be nice if the process executing *empty* could effectively just continue by executing *elem*, and similarly, if *elem* could continue as the same process once x has been sent along t. This can be achieved if we treat *tail calls* specially. So instead of writing

```
4    $e \leftarrow \textit{elem} \leftarrow x\ n$
5    $s \leftarrow e$
```

we write

```
4    $s \leftarrow \textit{elem} \leftarrow x\ n$
```

and similarly in the definition of *elem*.

In general, we compress a cut in the form of a process invocation followed by an identity simply as a process invocation:

$y \leftarrow X \leftarrow y_1 \ldots y_n$
$x \leftarrow y$

becomes

$x \leftarrow X \leftarrow y_1 \ldots y_n$

This is analogous to the so-called *tail-call optimization* in functional languages where instead of f calling a function g and immediately returning its value, f just continues as g. This is often represented as saving stack space since it can be implemented as a jump instead of a call. Here, too, recursively defined processes executing a sequence of interactions can simply continue without spawning a new process and then forwarding the result immediately, thereby saving process invocations.

From now on, we will often silently use the compressed form. Of course, its purely logical meaning can be recovered by expanding it into a cut followed by an identity.

3 Analyzing Parallel Complexity

We can analyze various complexity measures of our implementations. For example, we can count the number of processes that execute. Any call (except for a tail call) will spawn a new process, and any forward and close will terminate a process. Looking at the code below we can see that inserting an element into a store will spawn exactly one new process, namely when we eventually insert the last element into the empty store. Deleting an element will terminate exactly one process: either the empty one, or the one holding the element we are returning. Therefore in a store with n elements there will be *exactly* $n+1$ processes.

$$\mathsf{store}_A = \&\{\ \mathsf{ins} : A \setminus \mathsf{store}_A,\ \mathsf{del} : \oplus\{\mathsf{none} : \mathbf{1}, \mathsf{some} : A \bullet \mathsf{store}_A\}\}$$

$\cdot \vdash \mathit{empty} :: (s : \mathsf{store}_A)$

```
$s \leftarrow \mathit{empty} =$
  case $s$ (ins $\Rightarrow$ $x \leftarrow$ recv $s$ ;        % $(x{:}A) \vdash s : \mathsf{store}_A$
                 $n \leftarrow \mathit{empty}$ ;          % $(x{:}A)\ (n{:}\mathsf{store}_A) \vdash s : \mathsf{store}_A$
                 $s \leftarrow \mathit{elem} \leftarrow x\ n$
           | del $\Rightarrow$ $s$.none ; close $s$)
```

$(x{:}A)\ (t{:}\mathsf{store}_A)\ \vdash \mathit{elem} :: (s : \mathsf{store}_A)$

```
$s \leftarrow \mathit{elem} \leftarrow x\ t =$
    case $s$ (ins $\Rightarrow$ $y \leftarrow$ recv $s$ ;        % $(y{:}A)\ (x{:}A)\ (t{:}\mathsf{store}_A) \vdash s : \mathsf{store}_A$
                   $t$.ins ;                  % $(y{:}A)\ (x{:}A)\ (t{:}A \setminus \mathsf{store}_A) \vdash s : \mathsf{store}_A$
                   send $t\ x$ ;              % $(y{:}A)\ (t{:}\mathsf{store}_A) \vdash s : \mathsf{store}_A$
                   $s \leftarrow \mathit{elem} \leftarrow y\ t$
             | del $\Rightarrow$ $s$.some ;           % $(x{:}A)\ (t{:}\mathsf{store}_A) \vdash s : A \bullet \mathsf{store}_A$
                   send $s\ x$ ;              % $t{:}\mathsf{store}_A \vdash s : \mathsf{store}_A$
                   $s \leftarrow t$)
```

Another interesting measure is the *reaction time* which is analogous to the *span* complexity measure for parallel programs. If we try to carry out two consecutive operations, how many steps must elapse between them, assuming maximal parallelism? Here it is convenient to count every interaction as a step and no other costs.

Looking at the code for *elem* we see that there are only two interactions along channel t until the *elem* process can interact again along s after it has received ins and y. For *empty* there is only one spawn but no other interactions. Moreover, there is no delay for a deletion, since the process will respond immedidately along s.

In aggregate, when we store n elements consecutively, the constant reaction time means that there will be n elements building up the internal data structure simultaneously. No matter how many insertions and deletions we carry out, the reaction time (measured in total system interactions assuming maximal parallelism) is always constant.

On the other hand, if we count the total number of interactions of the system taking place (ignoring any question of parallelism) we see that for n insertions it will be $O(n^2)$, since each new element initiates a chain reaction that reaches to the end of the chain of elements. This is usually called the *work* performed by the algorithm.

4 Queues

As notes, our implementation so far ended up behaving like a stack, and we conjectured that the type of the interface itself forced this behavior. Can we modify the type to allow (and perhaps force) the behavior of the store as a queue, where the first element we store is the first one we receive back? I encourage you to try to work this out before reading on . . .

The key idea is to change the type

$$\begin{aligned}\mathsf{store}_A = \&\{\ & \mathsf{ins} : A \setminus \mathsf{store}_A, \\ & \mathsf{del} : \oplus\{\mathsf{none} : \mathbf{1}, \mathsf{some} : A \bullet \mathsf{store}_A\}\}\end{aligned}$$

to

$$\begin{aligned}\mathsf{queue}_A = \&\{\ & \mathsf{ins} : \mathsf{queue}_A \;/\; A, \\ & \mathsf{del} : \oplus\{\mathsf{none} : \mathbf{1}, \mathsf{some} : A \bullet \mathsf{queue}_A\}\}\end{aligned}$$

We will not go through this in detail, but reading the following code and the type after each interaction should give you a sense for what this change entails.

$\cdot \vdash \mathit{empty} :: (s : \mathsf{queue}_A)$

```
1   $s \leftarrow \mathit{empty} =$
2     case $s$ (ins $\Rightarrow$ $x \leftarrow$ recv $s$ ;        % $x{:}A \vdash s : \mathsf{queue}_A$
3                  $n \leftarrow \mathit{empty}$ ;          % $(x{:}A)\ (n{:}\mathsf{queue}_A) \vdash s : \mathsf{queue}_A$
4                  $s \leftarrow \mathit{elem} \leftarrow x\ n$
5             | del $\Rightarrow$ $s$.none ; close $s$)
```

$(x{:}A)\ (t{:}\mathsf{queue}_A)\ \vdash \mathit{elem} :: (s : \mathsf{queue}_A)$

```
6   $s \leftarrow \mathit{elem} \leftarrow x\ t =$
7       case $s$ (ins $\Rightarrow$ $y \leftarrow$ recv $s$ ;      % $(x{:}A)\ (t{:}\mathsf{queue}_A)\ (y{:}A) \vdash s : \mathsf{queue}_A$
8                    $t$.ins ;                % $(x{:}A)\ (t{:}\mathsf{queue}_A \;/\; A)\ (y{:}A) \vdash s : \mathsf{queue}_A$
9                    send $t\ y$ ;            % $(x{:}A)\ (t{:}\mathsf{queue}_A) \vdash s : \mathsf{queue}_A$
10                   $s \leftarrow \mathit{elem} \leftarrow x\ t$
11            | del $\Rightarrow$ $s$.some ;          % $(x{:}A)\ (t{:}\mathsf{queue}_A) \vdash s : A \bullet \mathsf{queue}_A$
12                   send $s\ x$ ;            % $t{:}\mathsf{queue}_A \vdash s : \mathsf{queue}_A$
13                   $s \leftarrow t$)
```

The critical changes are in line 7 (where y is added to the *right end* of the antecedents instead of the left) and line 9 (where consequently y instead of x must be sent along t).

The complexity of all the operations remains the same, since the only difference is whether the current x or the new y is sent along t, but the implementation now behaves like a queue rather than a stack.

Exercises

Exercise 1 In this exercise we explore an alternative implementation of stacks. First, consider type of stacks (renamed from store_A in this lecture)

$$\begin{aligned}\mathsf{stack}_A = \&\{\ & \mathsf{ins} : A \setminus \mathsf{stack}_A, \\ & \mathsf{del} : \oplus\{\mathsf{none} : \mathbf{1}, \mathsf{some} : A \bullet \mathsf{stack}_A\}\}\end{aligned}$$

1. Provide definitions for

   $\cdot \vdash \mathit{stack\_new} :: (s : \mathsf{stack}_A)$
   $l{:}\mathsf{list}_A \vdash \mathit{stack} :: (s : \mathsf{stack}_A)$

   which represents the elements of the stack in a list. If you need auxiliary process definitions for lists, please state them clearly, including their type.

2. Repeat the analysis of Section 3:

   (a) How many processes execute for a stack with n elements?

   (b) What is the reaction time for an insertion or deletion given a stack with n elements?

   (c) What is the total work for each insertion or deletion given a stack with n elements?

Exercise 2 In this exercise we explore an alternative implementation of queues. First, recall the type of queues from Section 4.

$$\begin{aligned}\mathsf{queue}_A = \&\{\ & \mathsf{ins} : \mathsf{queue}_A \mathbin{/} A, \\ & \mathsf{del} : \oplus\{\mathsf{none} : \mathbf{1}, \mathsf{some} : A \bullet \mathsf{queue}_A\}\}\end{aligned}$$

1. Provide definitions for

   $\cdot \vdash \mathit{queue\_new} :: (s : \mathsf{queue}_A)$
   $l{:}\mathsf{list}_A \vdash \mathit{queue} :: (s : \mathsf{queue}_A)$

   which represents the elements of the queue in a list. If you need auxiliary process definitions for lists, please state them clearly, including their type.

2. Repeat the analysis of Section 3:

(a) How many processes execute for a queue with n elements?

(b) What is the reaction time for an insertion or deletion given a queue with n elements?

(c) What is the total work for each insertion or deletiion given a queue with n elements?

Exercise 3 In this exercise we will "turn around" Exercise 1. Write a process definition

$$s{:}\mathsf{stack}_A \vdash \mathsf{to\_list} :: (l{:}\mathsf{list}_A)$$

which converts a stack into a list. As far as you can tell, is the order of the elements that are sent along l fixed?

Exercise 4 Consider the standard functional programming technique of implementing a queue with two lists. Just briefly, we have an *input list in* to which we add elements when they are enqueued and an *output list out* from which we take elements when they are dequeued. When the output list becomes empty, we reverse the input list, adding each element in turn onto the output list. Initially, both lists are empty.

Explore if you can write such an implementation against the queue interface from Section 4. The implementation should have one of the two types

$$(\mathit{in}{:}\mathsf{list}_A)\ (\mathit{out}{:}\mathsf{list}_A) \vdash \mathit{queue2} :: (s : \mathsf{queue}_A)$$
$$(\mathit{out}{:}\mathsf{list}_A)\ (\mathit{in}{:}\mathsf{list}_A) \vdash \mathit{queue2} :: (s : \mathsf{queue}_A)$$

References

[CPPT13] Luís Caires, Jorge A. Pérez, Frank Pfenning, and Bernardo Toninho. Behavioral polymorphism and parametricity in session-based communication. In M.Felleisen and P.Gardner, editors, *Proceedings of the European Symposium on Programming (ESOP'13)*, pages 330–349, Rome, Italy, March 2013. Springer LNCS 7792.

[Rey83] John C. Reynolds. Types, abstraction, and parametric polymorphism. In R.E.A. Mason, editor, *Information Processing 83*, pages 513–523. Elsevier, September 1983.