Lecture Notes on Stacks and Queues

15-317: Constructive Logic
Frank Pfenning

Lecture 25
December 5, 2017

1 Introduction

We begin this section by writing some more examples on ordered lists and then implementations of lists and queues in an object-oriented style.

Recall the definition of ordered antecedents

$$\Omega ::= (c : A) \mid \epsilon \mid \Omega_1 \cdot \Omega_2$$

where '$\cdot$' is an associative operator with unit ϵ, and c is interpreted as a channel. The basic judgment

$$(c_1 : A_1) \cdots (c_n : A_n) \vdash P :: (c : A)$$

means that P is a process *using* channels c_i and *providing* channel c.

We summarize the language constructs so far in the following table, from the point of view of the provider of a service.

Type	**Provider**	**Continuation**	**Client**
$c : \oplus\{\ell : A_\ell\}_{\ell \in L}$	$c.k$	$c : A_k$	$\mathsf{case}\ c\ (\ell \Rightarrow Q_\ell)_{\ell \in L}$
$c : \&\{\ell : A_\ell\}_{\ell \in L}$	$\mathsf{case}\ c\ (\ell \Rightarrow Q_\ell)_{\ell \in L}$	$c : A_k$	$c.k$
$c : \mathbf{1}$	$\mathsf{close}\ c$	(*none*)	$\mathsf{wait}\ c\ ;\ Q$
$c : A \bullet B$	$\mathsf{send}\ c\ d$	$c : B$	$x \leftarrow \mathsf{recv}\ c\ ;\ Q_x$

2 List Constructors

Recall the type of ordered lists

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

Even though this session type describes a message interface rather than data layed out in memory, and it is ordered rather than composed of arbitrary pairs, the analogy to data types in functional language should be clear. For example, in the syntax of Standard ML, we might translate this type as

```
datatype 'a list =
   cons of 'a * 'a list
 | nil of unit
```

In a functional language, this give us `cons` and `nil` as constructors. Are there analogous *processes* here? Let's consider

$(x : A)\ (l : \mathsf{list}_A) \vdash \mathit{cons} :: (r : \mathsf{list}_A)$

When defining *cons*, x, l, and r are process parameters, which we indicate by

$r \leftarrow \mathit{cons} \leftarrow x\ l = \ldots$

where the body of the process definition refers to r, x, and l. The intent is that r represents the list with head x and tail l. It can announce the fact that is starts with a cons by sending this:

$$
\begin{array}{ll}
r \leftarrow \mathit{cons} \leftarrow x\ l = & \\
\quad r.\mathsf{cons}\ ; & \%\ (x : A)\ (l : \mathsf{list}_A) \vdash r : A \bullet \mathsf{list}_A \\
\quad \ldots &
\end{array}
$$

Now we have to send a channel of type A along r. Fortunately, the element $x : A$ is at the left end of the antecedents so the ordering restriction on $\bullet R^*$ is satisfied and we can send it.

$$
\begin{array}{ll}
r \leftarrow \mathit{cons} \leftarrow x\ l = & \\
\quad r.\mathsf{cons}\ ; & \%\ (x : A)\ (l : \mathsf{list}_A) \vdash r : A \bullet \mathsf{list}_A \\
\quad \mathsf{send}\ r\ x\ ; & \%\ (l : \mathsf{list}_A) \vdash r : \mathsf{list}_A \\
\quad \ldots &
\end{array}
$$

Now that we have sent x, we express that the remainder of the list r is l by forwarding l to r. Fortunately, the types are arranged in the right way.

$$
\begin{array}{ll}
r \leftarrow \mathit{cons} \leftarrow x\ l = & \\
\quad r.\mathsf{cons}\ ; & \%\ (x : A)\ (l : \mathsf{list}_A) \vdash r : A \bullet \mathsf{list}_A \\
\quad \mathsf{send}\ r\ x\ ; & \%\ (l : \mathsf{list}_A) \vdash r : \mathsf{list}_A \\
\quad r \leftarrow l &
\end{array}
$$

The *nil* constructor for a process representing the empty list works analogously.

$\vdash \mathit{nil} :: (r : \mathsf{list}_A)$
$r \leftarrow \mathit{nil} = r.\mathsf{nil}\ ;\ \mathsf{close}\ r$

3 Following the Types

Next, we explore the prescriptive power of types. Which mystery processes would have type

$(l : \mathsf{list}_A)\ (x : A) \vdash \mathit{myst} :: (r : \mathsf{list}_A)$

This is almost the type of *cons*, except that the order of the antecedents is reversed. If we were working in linear logic, where $A \bullet B$ is symmetric, *cons* would in fact satisfy this type, but not in ordered logic because after sending the cons label

```
r ← myst ← l x =
    r.cons ;                    % (l : list_A) (x : A) ⊢ r : A • list_A
    ...
```

we cannot send x because it is not at the left end of the antecedents.

Intuitively, if the list l is "virtually" in the context with its elements in order, then maybe we should be able to add x at the end. This means we actually have to read the elements from l, similar to our implementation of *append*.

```
(l : list_A) (x : A) ⊢ myst :: (r : list_A)
r ← myst ← l x =
    case l ( cons ⇒ y ← recv l ;        % (y : A) (l : list_A) (x : A) ⊢ (r : list_A)
        ...)
```

At this point we can send cons and then y along r, but not x, which will only be available once the list l has been been transferred to r in its entirety. Once we have sent y, we can recurse, passing the same l and x back to *myst*.

```
(l : list_A) (x : A) ⊢ myst :: (r : list_A)
r ← myst ← l x =
    case l ( cons ⇒ y ← recv l ;        % (y : A) (l : list_A) (x : A) ⊢ (r : list_A)
                    r.cons ; send r y ;  % (l : list_A) (x : A) ⊢ r : list_A
                    r ← myst ← l x
           | nil ⇒ ...)
```

In the case of nil we can wait for l to close and then send cons, x, nil, and then close.

$$
\begin{array}{l}
(l : \mathsf{list}_A)\ (x : A) \vdash \mathit{myst} :: (r : \mathsf{list}_A) \\
r \leftarrow \mathit{myst} \leftarrow l\ x = \\
\quad \mathsf{case}\ l\ (\mathsf{cons} \Rightarrow\ y \leftarrow \mathsf{recv}\ l\ ; \qquad \% \ (y : A)\ (l : \mathsf{list}_A)\ (x : A) \vdash (r : \mathsf{list}_A) \\
\qquad\qquad\qquad\ \ r.\mathsf{cons}\ ;\ \mathsf{send}\ r\ y\ ; \quad \% \ (l : \mathsf{list}_A)\ (x : A) \vdash r : \mathsf{list}_A \\
\qquad\qquad\qquad\ \ r \leftarrow \mathit{myst} \leftarrow l\ x \\
\qquad\quad \mid \mathsf{nil} \Rightarrow\ \ \mathsf{wait}\ l\ ; \qquad \% \ (x : A) \vdash r : \mathsf{list}_A \\
\qquad\qquad\qquad\ \ r.\mathsf{cons}\ ;\ \mathsf{send}\ r\ x \quad \% \ \vdash r : \mathsf{list}_A \\
\qquad\qquad\qquad\ \ r.\mathsf{nil}\ ;\ \mathsf{close}\ r)
\end{array}
$$

We can also use our *nil* and *cons* in several places instead falling back on sending messages directly.

$$
\begin{array}{l}
(l : \mathsf{list}_A)\ (x : A) \vdash \mathit{myst} :: (r : \mathsf{list}_A) \\
r \leftarrow \mathit{myst} \leftarrow l\ x = \\
\quad \mathsf{case}\ l\ (\mathsf{cons} \Rightarrow\ y \leftarrow \mathsf{recv}\ l\ ; \qquad \% \ (y : A)\ (l : \mathsf{list}_A)\ (x : A) \vdash r : \mathsf{list}_A \\
\qquad\qquad\qquad\ \ s \leftarrow \mathit{myst} \leftarrow l\ x\ ; \quad \% \ (y : A)\ (s : \mathsf{list}_A) \vdash r : \mathsf{list}_A \\
\qquad\qquad\qquad\ \ r \leftarrow \mathit{cons} \leftarrow y\ s \\
\qquad\quad \mid \mathsf{nil} \Rightarrow\ \ \mathsf{wait}\ l\ ; \qquad \% \ (x : A) \vdash r : \mathsf{list}_A \\
\qquad\qquad\qquad\ \ n \leftarrow \mathit{nil}\ ; \qquad \% \ (x : A)\ (n : \mathsf{list}_A) \vdash r : \mathsf{list}_A \\
\qquad\qquad\qquad\ \ r \leftarrow \mathit{cons} \leftarrow x\ n)
\end{array}
$$

Note how, for example, the recursive call to *myst* consumes $(l : \mathsf{list}_A)\ (x : A)$ from the antecedents and replace it by $(s : \mathsf{list}_A)$. This stems from the form of the ordered cut rule

$$\frac{\Omega \vdash P :: (x : A) \quad \Omega_L \cdot (x : A) \cdot \Omega_R \vdash Q :: (z : C)}{\Omega_L \cdot \Omega \cdot \Omega_R \vdash (x \leftarrow P\ ;\ Q) :: (z : C)}\ \mathsf{cut}$$

where here $\Omega_L = (y : A)$, $\Omega = (l : \mathsf{list}_A)\ (x : A)$, and $\Omega_R = \epsilon$.

Also, in the case of nil where we spawn a new *nil* process, no antecedents are consumed, so the new channel n can go anywhere in the context. We will need it to the right of $x : A$ so we can call *cons* next to construct a singleton list.

So we see there are different programs, with different behavior (for example, the placement of the recursive call), and yet in the end the observable behavior along r should be the same: the elements of l followed by x.

4 Ordered Implications

Now we move on from the quasi-functional style a quasi-object-oriented example: implementing a store with insert and delete operations. Rather than an internal choice ($\oplus$) like lists, the interface will present the client with a choice between insertion and deletion in the form of an external choice ($\&$).

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

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

In the first line we see that the provider may receive and ins message, and then it must receive the channel to insert into the store. We model this with $A \setminus B$, which we have not yet introduced. First, the right rule:

$$\frac{(y{:}A)\ \Omega \vdash P_y :: (x : B)}{\Omega \vdash (y \leftarrow \mathsf{recv}\ x\ ;\ P_y) :: (x : A \setminus B)}\ \setminus R$$

Operationally, a provider of $x : A \setminus B$ receives a channel of type A, adds it to the left end of the antecedents, and continues with $x : B$.

A client process using a channel $x : A \setminus B$ must therefore send a channel of type A. Moreover, due to ordering constraints, this channel must be immediately to the left of x.

$$\frac{\Omega_L\ (x : B)\ \Omega_R \vdash Q :: (z : C)}{\Omega_L\ (w : A)\ (x : A \setminus B)\ \Omega_R \vdash (\mathsf{send}\ x\ w\ ;\ Q) :: (z : C)}\ \setminus L^*$$

The following computation rule implements the cut reduction of $\setminus R$ and $\setminus L^*$.

$$\frac{\mathsf{proc}(x, y \leftarrow \mathsf{recv}\ x\ ;\ P_y) \quad \mathsf{proc}(z, \mathsf{send}\ x\ w\ ;\ Q)}{\mathsf{proc}(x, [w/y]P_y) \quad \mathsf{proc}(z, Q)}\ \setminus C$$

Note that the operational reading here is *identical* for $B\ /\ A$; the difference is entirely in the restrictions about where $w{:}B$ or $y{:}B$ are to be found. In the $/R$ rule it will be added to the right of the antecedents and in the $/L^*$ rule it must be immediately to the right of the receiving channel.

$$\frac{\Omega\ (y{:}A) \vdash P_y :: (x : B)}{\Omega \vdash (y \leftarrow \mathsf{recv}\ x\ ;\ P_y) :: (x : B\ /\ A)}\ /R$$

$$\frac{\Omega_L\ (x : B)\ \Omega_R \vdash Q :: (z : C)}{\Omega_L\ (x : B\ /\ A)\ (w : A)\ \Omega_R \vdash (\mathsf{send}\ x\ w\ ;\ Q) :: (z : C)}\ /L^*$$

5 Implementing a Store

Recall the desired type of a store interface, using $A \setminus B$:

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

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.

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

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

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

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

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

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 $\mathsf{store}_A \,/\, A$ would behave the same way here.

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

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*

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

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*.

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

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

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.

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

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

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

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

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

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

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.

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

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

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.

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

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.

6 Tail Calls

If we look at lines 5 and 6

$$\begin{array}{ll} r \leftarrow \mathit{elem} \leftarrow y\ t\ ; & \% \quad r{:}\mathsf{store}_A \vdash s : \mathsf{store}_A \\ s \leftarrow r & \end{array}$$

we are starting a new process, providing along a new channel r and then forward this to s. Instead, we can simply continue in the current process, executing *elem*, which is written as

$$s \leftarrow \mathit{elem} \leftarrow y\ t\ ;$$

The examples now simplify very slightly.

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

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

$$\begin{array}{lll}
(x{:}A)\ (t{:}\mathsf{store}_A)\ \vdash \mathit{elem} :: (s : \mathsf{store}_A) & & \\
s \leftarrow \mathit{elem} \leftarrow x\ t = & & \\
\quad \mathsf{case}\ s\ (\mathsf{ins} \Rightarrow y \leftarrow \mathsf{recv}\ s\ ; & \% & (y{:}A)\ (x{:}A)\ (t{:}\mathsf{store}_A) \vdash s : \mathsf{store}_A \\
\qquad\qquad\qquad\quad t.\mathsf{ins}\ ; & \% & (y{:}A)\ (x{:}A)\ (t{:}A \setminus \mathsf{store}_A) \vdash s : \mathsf{store}_A \\
\qquad\qquad\qquad\quad \mathsf{send}\ t\ x\ ; & \% & (y{:}A)\ (t{:}\mathsf{store}_A) \vdash s : \mathsf{store}_A \\
\qquad\qquad\qquad\quad s \leftarrow \mathit{elem} \leftarrow y\ t & & \\
\qquad\qquad\quad \mid \mathsf{del} \Rightarrow s.\mathsf{some}\ ; & \% & (x{:}A)\ (t{:}\mathsf{store}_A) \vdash s : A \bullet \mathsf{store}_A \\
\qquad\qquad\qquad\quad \mathsf{send}\ s\ x\ ; & \% & t{:}\mathsf{store}_A \vdash s : \mathsf{store}_A \\
\qquad\qquad\qquad\quad s \leftarrow t) & &
\end{array}$$

7 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

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

to

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

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 ← empty =
2     case s (ins ⇒ x ← recv s ;          %  x:A ⊢ s : queue_A
3                   n ← empty ;           %  (x:A) (n:queue_A) ⊢ s : queue_A
4                   s ← elem ← x n
5             | del ⇒ s.none ; close s)
```

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

```
6   s ← elem ← x t =
7       case s (ins ⇒ y ← recv s ;        %  (x:A) (t:queue_A) (y:A) ⊢ s : queue_A
8                     t.ins ;             %  (x:A) (t:queue_A / A) (y:A) ⊢ s : queue_A
9                     send t y ;          %  (x:A) (t:queue_A) ⊢ s : queue_A
10                    s ← elem ← x t
11                | del ⇒ s.some ;         %  (x:A) (t:queue_A) ⊢ s : A • queue_A
12                    send s x ;          %  t:queue_A ⊢ s : queue_A
13                    s ← 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.

8 Summary, and Linear Logic

We complete the table from the beginning of the lecture that summarizes the computational interpretation of ordered logic.

Type	Provider	Continuation	Client
$c : \oplus\{\ell : A_\ell\}_{\ell \in L}$	$c.k$	$c : A_k$	case c $(\ell \Rightarrow Q_\ell)_{\ell \in L}$
$c : \&\{\ell : A_\ell\}_{\ell \in L}$	case c $(\ell \Rightarrow Q_\ell)_{\ell \in L}$	$c : A_k$	$c.k$
$c : \mathbf{1}$	close c	(*none*)	wait c ; Q
$c : A \bullet B$	send c d	$c : B$	$x \leftarrow$ recv c ; Q_x
$c : A \setminus B$	$x \leftarrow$ recv c ; P_x	$c : B$	send c d
$c : B \,/\, A$	$x \leftarrow$ recv c ; P_x	$c : B$	send c d

The difference between the last two rows are the places of x and d among the antecedents.

In the case of linear logic, the last two lines collapse, using $A \multimap B$ in a unified notation, and the multiplicative conjunction $A \bullet B$ no longer is subject to any ordering constraint and is written as $A \otimes B$. The programming constructs do not change.

Type	Provider	Continuation	Client
$c : \oplus\{\ell : A_\ell\}_{\ell \in L}$	$c.k$	$c : A_k$	case c $(\ell \Rightarrow Q_\ell)_{\ell \in L}$
$c : \&\{\ell : A_\ell\}_{\ell \in L}$	case c $(\ell \Rightarrow Q_\ell)_{\ell \in L}$	$c : A_k$	$c.k$
$c : \mathbf{1}$	close c	(*none*)	wait c ; Q
$c : A \otimes B$	send c d	$c : B$	$x \leftarrow$ recv c ; Q_x
$c : A \multimap B$	$x \leftarrow$ recv c ; P_x	$c : B$	send c d