From Singleton to Linear Logic
Frank Pfenning
OPLSS 2019

Date Performed: June 18th 2019
Students: J.W.N. Paulus
R. Gurdeep Singh
H. C. A. Tavante

4 Lecture 4: Linear logic $\otimes$ session types

4.1 Connectives of Linear Logic

Judgement

$$\underbrace{\Gamma}_{A_1,\ldots,A_n} \vdash A$$

Id:

$$\frac{}{A \vdash A}\ ID$$

Cut-rule:

$$\frac{\Gamma_1 \vdash A \qquad \Gamma_2, A \vdash C}{\Gamma_1, \Gamma_2 \vdash C}\ CUT$$

$\oplus$ Right and Left rules (Internal choice)

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \oplus B}\ \oplus R_1 \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \oplus B}\ \oplus R_2 \qquad \frac{\Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma, A \oplus B \vdash C}\ \oplus L$$

& Right and Left rules (External choice)

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \& B}\ \& R \qquad \frac{\Gamma, A \vdash C}{\Gamma, A \& B \vdash C}\ \& L_1 \qquad \frac{\Gamma, B \vdash C}{\Gamma, A \& B \vdash C}\ \& L_2$$

$\otimes$ Right and Left rules (Tensor; dual choice)

$$\frac{\Gamma_1 \vdash A \quad \Gamma_2 \vdash B}{\Gamma_1, \Gamma_2 \vdash A \otimes B}\ \otimes R \qquad \frac{\Gamma, A, B \vdash C}{\Gamma, A \otimes B \vdash C}\ \otimes L$$

$\multimap$ Right and Left rules (Also kown as Linear implication or "Loly")

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \multimap B} \multimap R \qquad \frac{\Gamma_1 \vdash A \quad \Gamma_2, B \vdash C}{\Gamma_1, \Gamma_2, A \multimap B \vdash C} \multimap L$$

No resources at all (the dot $\bullet$ means there are no resources):

$$\frac{}{\bullet \vdash 1}\ 1R \qquad \frac{\Gamma \vdash C}{\Gamma, 1 \vdash C}\ 1L$$

4.2 ???

4.3 Implementing a queue

We will implement a queue in linear logic. Professor Pfenning says that he wants to do this in an object oriented way.

4.3.1 Type

Our queue object will feature an enqueue method (enQ) and a dequeue (deQ) method. We start by defining the type of the queue. Because the user of the queue determines how it is used, the type will start with an external choice (&). When an enqueue is requested, we "provide to read on a channel of type a and then return to being a queue" ($A \multimap queue_A$). For the dequeue there are two options.

- The queue is either empty ($none$). In this case, we terminate the queue by "returning" type 1.
- The queue contains data ($some$) so we can return it. We will send an A and become a queue again ($A \otimes queue_A$).

Depending on its state, the queue internally ($\oplus$) decides what to return. We can thus write the following type $queue_A$ for a queue with elements of type A:

$$\begin{aligned} queue_A = \&\{&enQ : A \multimap queue_A \\ &deQ : \oplus\{none : 1, \quad some : A \otimes queue_A\}\} \end{aligned}$$

A note on destruction In the foregoing type, we used type 1 when a dequeue was requested on an empty queue. This ensures that the queue terminates and gets out of the way. A downside of this is that once we popped all elements, we cannot use the queue any longer[1]. We could also have chosen to let the none return $queue_A$, to dealocate the queue we would then need an additional *destruct* method/message. But this more tricky. The destruct operation must "use up" all elements in the queue at that moment and then return with type 1. In our setup we take the opportunity to terminate the queue once it is empty.

[1] because in linear types everything must be exactly one time

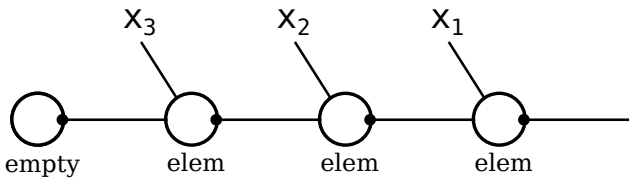

Figure 1: Visualization of the processes (circles) involved in implementing a queue

4.3.2 Implementation

Our queue effectively has two behaviours, an empty-behaviour and a non-empty behaviour. Figure 1 visualizes the structure of the queue process. There is a process that handles the empty queue, and there is an *elem* process for each element in the queue. Both kinds of processes accept the queue interface. Each of the *elem* processes have an "input" of type A that contains the value they hold. To add a new value the *empty* node will be replaced with an *elem* node that has (a new) *empty* as next process.
The type of *elem* reflects that it uses a stored value x of type A and a "tail" queue t, its interface it that of $queue_A$. So, we have:

$$x : A, t : queue_A \vdash elem :: (q : queue_A)$$

The implementation is shown below, the blue text at the end of each line indicates the type we have at the moment.

$$\begin{array}{lrl}
q \leftarrow elem \leftarrow t, x = \text{CASE } q(enQ \Rightarrow & y \leftarrow \text{RECV } q; & (x : A, t : queue_A \vdash q : A \multimap queue_A) \\
& t.enQ; & (x : A, t : queue_A, y : A \vdash q : queue_A) \\
& \text{SEND } t, y; & (x : A, y : A, t : A \multimap queue_A \vdash q : queue_A) \\
& q \leftarrow elem \leftarrow t, x & (x : A, t : queue_A \vdash q : queue_A) \\
 & & \\
|deQ \Rightarrow & q.some; & (x : A, t : queue_A \vdash q : A \otimes queue_A) \\
& \text{SEND } q, x; & (x : A, t : queue_A \vdash q : queue_A) \\
& q \leftarrow t) & (t : queue_A \vdash q : queue_A)
\end{array}$$

Empty has the following type

$$\vdash elem :: (q : queue_A)$$

The implementation is shown below again:

$$
\begin{array}{lrl}
q \leftarrow \mathit{empty} = \textsc{case}\ q(\mathit{enQ} \Rightarrow & y \leftarrow \textsc{recv}\ q; & (\vdash q : A \multimap \mathit{queue}_A) \\
& e \leftarrow \mathit{empty}; & (y : A \vdash q : \mathit{queue}_A) \\
& q \leftarrow \mathit{elem} \leftarrow e, y & (x : A, e : \mathit{queue}_A \vdash q : \mathit{queue}_A) \\
\\
| \mathit{deQ} \Rightarrow & q.\mathit{none}; & \\
& \textsc{close}\ q & (\vdash q : 1)
\end{array}
$$

4.3.3 Other related advances

- Shared memeory instead of message passing
- parallel complexity $\longrightarrow$ temporal logic (ICFP'18)
- sequential complexity