Lecture Notes on Message-Passing Concurrency

15-814: Types and Programming Languages
Frank Pfenning

Lecture 24
Tuesday, December 1, 2020

1 Introduction

So far, we have viewed concurrency through the lens of shared memory. That's because there is a direct way of translating expressions into processes that make memory allocation as well as reading and writing explicit. The usual sequential dynamics can be recovered easily, as pointed out in Section L21.5 and Exercise L21.4, but concurrency is in fact most natural. Nevertheless computation is quite *pure*, not covering mutable references, but see Lecture 22 for an approach to adding this to a call-by-value language.

Communication in shared memory takes the form of writing to and reading from shared cells. However, there are many situations where processes that execute concurrently do not have a shared address space but need to communicate with each other via messages. And even if shared memory is available, message-passing is a useful, perhaps higher-level abstraction that may prevent certain kinds of errors. For example, the Go concurrency slogan[1] exhorts:

> *Do not communicate by sharing memory; instead, share memory by communicating.*

So, given our type-based approach, can we model message-passing concurrency? The answer to this rhetorical question is of course "Yes!" In fact, the whole approach to concurrency we have taken in this course originated by examining message-passing concurrency through the lens of *linear*

[1] https://golang.org/doc/effective_go.html

logic [CP10, CPT16]. The adaption to shared memory did not come until later [PP20].

There are different computational models for message-passing concurrency, for example, *actors* [Agh85] or the *π-calculus* [MPW92]. Our model resembles the *asynchronous π-calculus* [Bou92] in that a sender can proceed immediately, while a recipient blocks until a message is received. Unlike the asynchronous π-calculus our calculus enforces that message are received in the order they are sent, which is essential for type soundness (that is, preservation and progress).

2 Reinterpreting Process Typing

The key step towards message-passing concurrency from where we are is to reinterpret the typing judgment for processes. With shared-memory concurrency we have

$$\underbrace{x_1 : \tau_1, \ldots, x_n : \tau_n}_{\text{read from}} \Vdash P :: \underbrace{(y : \sigma)}_{\text{write to}}$$

where all variables x_i and y stand for *addresses* in shared memory. The process P reads from the x_i and writes to y. Linearity ensured that a terminating process is guaranteed to read all the x_i and write to y.

With message-passing concurrency, each variable stands for a *channel for bidirectional communication*,

$$\underbrace{x_1 : \tau_1, \ldots, x_n : \tau_n}_{\substack{\text{use} \\ \text{may send or recv}}} \Vdash P :: \underbrace{(y : \sigma)}_{\substack{\text{provide} \\ \text{may send or recv}}}$$

where the channel y on the right represents a service provided and the channels x_i on the left a service used. We say P is a *provider* for y and a *client* to all x_i. It is now the types σ and τ_i that determine whether messages are sent or received.

The rule to allocate or spawn (called *cut* under its logical interpretation) still has the same effect, except that it allocates a *fresh private channel* connecting a provider P to its client Q instead of a memory cell.

$$\frac{\Delta \Vdash P :: (x : \tau) \quad \Delta', x : \tau \Vdash Q :: (y : \sigma)}{\Delta, \Delta' \Vdash (x \leftarrow P \,;\, Q) :: (y : \sigma)}\ \mathsf{cut}$$

Because the channel x here is shared between P and Q, any type τ prescribes two complementary actions: one by the provider (say, a send) another one by the client (say, a corresponding receive).

The dynamics is as before, except we no longer record a distinguished destination since communication is bidirectional.

$$\mathsf{proc}\ (x \leftarrow P\ ;\ Q) \mapsto \mathsf{proc}\ ([c/x]P), \mathsf{proc}\ ([c/x]Q) \qquad (c \text{ fresh})$$

In Section 8 we return to the identity rule, which corresponded to moving a value from one cell to another but now *forwards* messages from one channel to another.

3 Tagged Sums Become Internal Choice

Taking the provider's perspective, constructs associated with positive types will *send* a message. Therefore, from the client's perspective, they will *receive*.

Intuitively, we think of a sequence of messages on a channel as beads on a string. While this image is correct, we have to be careful that multiple messages on a channel arrive in the order they were sent—otherwise, type safety might fail when consecutive messages have different type. The way we accomplish this is that every message (except one that closes a channel) carries a *continuation channel* for further communication. In an implementation, this could be achieved with an explicit message queue, which in our formulation is an emergent structure rather than a primitive.

The *tagged sum* $\sum_{i \in I}(i : \tau_i)$ corresponds to *internal choice* $\oplus_{i \in I}(i : \tau_i)$. A provider of a channel $x : \oplus_{i \in I}(i : \tau_i)$ will *send* a label $j \in I$ along x and a continuation channel of type τ_j. This is called *internal choice* because the *provider can choose* which label to send.

$$\frac{(j \in I)}{y : \tau_j \Vdash x.(j \cdot y) :: (x : \oplus_{i \in I}(i : \tau_i))}\ \oplus R^0$$

Correspondingly, the recipient will branch based on the label received.

$$\frac{\Delta, y_i : \tau_i \Vdash P_i :: (z : \sigma) \quad (\text{for all } i \in I)}{\Delta, x : \oplus_{i \in I}(i : \tau_i) \Vdash \mathsf{case}\ x\ (i \cdot y_i \Rightarrow P_i)_{i \in I} :: (z : \sigma)}\ \oplus L$$

These linear rules are exactly as before. In order to describe the changed dynamics, we need a second form of semantic object $\mathsf{msg}\ c\ V$ which means

that the small value V is a message on channel c. Then we have

$$\begin{array}{rlcl} & \mathsf{proc}\ (c.(j \cdot c')) & \mapsto & \mathsf{msg}\ c\ (j \cdot c') \\ \mathsf{msg}\ c\ (j \cdot c'), & \mathsf{proc}\ (\mathsf{case}\ c\ (i \cdot y_i \Rightarrow P_i)_{i \in I}) & \mapsto & \mathsf{proc}\ ([c'/y_j]P_j) \end{array}$$

Observe the significance of linearity here: in the second rule, the message is ephemeral so it is removed from the configuration so that now the next message can be received. Meanwhile, the continuation channel c' is substituted for y_j, which is the placeholder for the continuation channel.

4 Generalizing to Other Types

Before we write out our bit flipping pipeline once again (now in a message-passing interpretation), we can summarize the possible configurations and even the transition rules. Small values V are as before, except they are comprised of channels, not addresses. Continuation processes K are also as before (see the Lecture 23 Rule Sheet for reference).

$$\begin{array}{lrcll} \text{Processes} & P & ::= & x \leftarrow P\ ;\ Q & \text{spawn} \\ & & \mid & x.V & \text{send} \\ & & \mid & \mathsf{case}\ x\ K & \text{receive} \\ \\ \text{Configurations} & \mathcal{C} & ::= & \cdot \mid \mathcal{C}_1, \mathcal{C}_2 \mid \mathsf{proc}\ P \mid \mathsf{msg}\ c\ V \end{array}$$

The dynamics then, so far, consists of only three rules, where we have speculatively generalized, sending arbitrary V and passing them to arbitrary K when received.

$$\begin{array}{lcll} \mathsf{proc}\ (x \leftarrow P\ ;\ Q) & \mapsto & \mathsf{proc}\ ([c/x]P), \mathsf{proc}\ ([c/x]Q) & (c \text{ fresh}) \\ \mathsf{proc}\ (c.V) & \mapsto & \mathsf{msg}\ c\ V & (\text{send}) \\ \mathsf{msg}\ c\ V, \mathsf{proc}\ (\mathsf{case}\ c\ K) & \mapsto & \mathsf{proc}\ (V \triangleright K) & (\text{receive}) \end{array}$$

In fact, the last two rules will be sufficient for all positive and negative types!

A perhaps unexpected aspect of these rules is that a process that sends a message terminates. That works out because in programs we create a continuation channel c' and a very short-lived process that corresponds to just a message. This is actually quite similar to the asynchronous π-calculus where there is no explicit construct for sending a message, just parallel composition with a process that (intuitively) represents a message.

5 The Bit-Flipping Pipeline Revisited

In order to consider recursively defined processes, we allow global declarations of f in a fixed signature Σ in the form

$$x_1 : \tau_1, \ldots, x_n : \tau_n \Vdash f :: (y : \sigma)$$
$$y \leftarrow f\, x_1\, \ldots\, x_n = P$$

where the process expression P is typed with

$$x_1 : \tau_1, \ldots, x_n : \tau_n \Vdash P :: (y : \sigma)$$

We then add to the language of processes a *call*

$$d \leftarrow f\, c_1\, \ldots\, c_n$$

which is typed as follows (taking care to allow the concrete arguments to be different from the names of the parameters):

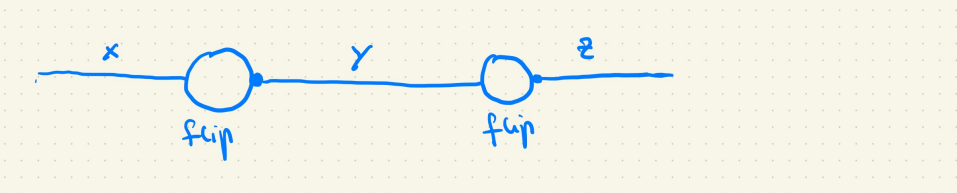
$$\frac{(x_1 : \tau_1, \ldots, x_n : \tau_n \Vdash f :: (y : \sigma)) \in \Sigma}{c_1 : \tau_1, \ldots, c_n : \tau_n \Vdash f :: (d : \sigma)}\ \mathsf{call}$$

In the concrete example of the bit-flipping pipeline we have

$$\mathit{bits} \cong (\mathbf{b0} : \mathit{bits}) + (\mathbf{b1} : \mathit{bits})$$

where we elide the fold constructor in the examples to simplify the code, as we have done in recent lectures.

Then the pipeline below



consists of two running processes, both executing *flip*. The channel y is a private channel connecting these two processes. We use a small dot to indicate the channel provided by a process (y for the process on the left and z for the process on the right).

x : *bits* $\Vdash$ *flip* :: (y : *bits*)
$y \leftarrow$ *flip* $x = \ldots$ *% to be written later*

x : *bits* $\Vdash$ *flip2* :: (z : *bits*)
$z \leftarrow$ *flip2* $x =$
  $y \leftarrow (y \leftarrow$ *flip* $x)$
  $z \leftarrow$ *flip* y

The first line in the definition of *flip2* creates a new channel y, which is provided by the first *flip* process and used by the second. Because this pattern is pervasive, we use a derived notation that combines a cut and abbreviate $x \leftarrow (x \leftarrow f\ y_1\ \ldots\ y_n)\ ;\ Q$ by $x \leftarrow f\ y_1\ \ldots\ y_n\ ;\ Q$.

x : *bits* $\Vdash$ *flip2* :: (z : *bits*)
$z \leftarrow$ *flip2* $x =$
  $y \leftarrow$ *flip* x
  $z \leftarrow$ *flip* y

We can quickly verify that this definition is *linear*: x is used in the first call to *flip* which provides y, used in the second call to *flip*.

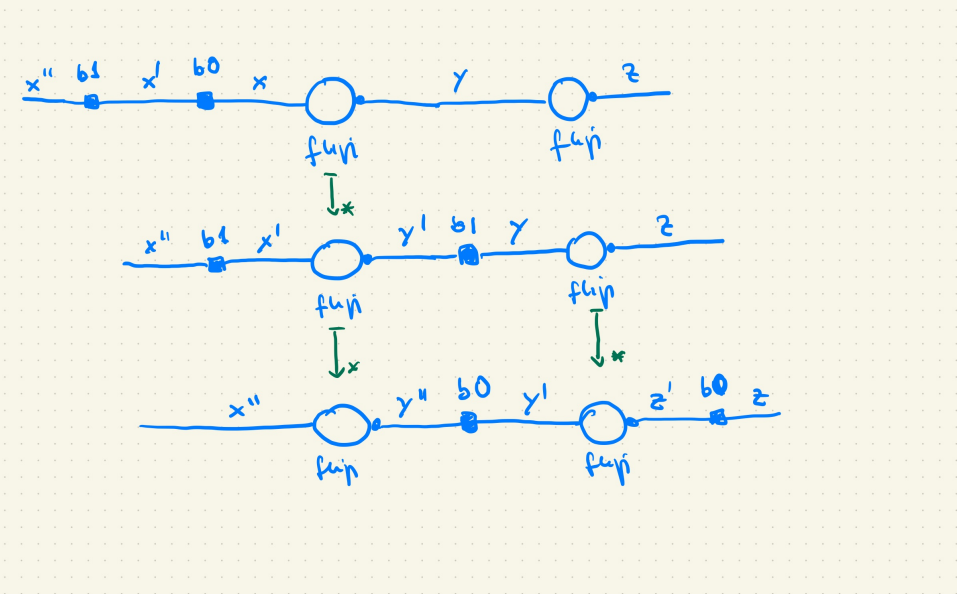
Now the code for *flip* itself receives a bit along channel x, together with a continuation channel x'.

$y \leftarrow$ *flip* $x =$
case x $(\ \mathbf{b0} \cdot x' \Rightarrow \ldots$
$\quad\ \mid \mathbf{b1} \cdot x' \Rightarrow \ldots$
$\quad\ )$

Before we can send the negated bit $\mathbf{b1}$ along y, we have to create the continuation channel for it. We obtain this from the recursive call, because after the interaction the *flip* process has continuation channels x' and y' on the two sides. The handling of $\mathbf{b1}$ is symmetric.

$y \leftarrow$ *flip* $x =$
case x $(\ \mathbf{b0} \cdot x' \Rightarrow y' \leftarrow$ *flip* $x'\ ;$
$\qquad\qquad\qquad\ \ y.(\mathbf{b1} \cdot y')$
$\quad\ \mid \mathbf{b1} \cdot x' \Rightarrow y' \leftarrow$ *flip* $x'\ ;$
$\qquad\qquad\qquad\ \ y.(\mathbf{b0} \cdot y')$
$\quad\ )$

In pictures:



The configuration shown can make the following transition:

$$\begin{aligned} & \mathsf{msg}\ x'\ (\mathsf{b1} \cdot x''), \mathsf{msg}\ x\ (\mathsf{b0} \cdot x'), \mathsf{proc}\ (y \leftarrow \mathit{flip}\ x), \mathsf{proc}\ (z \leftarrow \mathit{flip}\ y) \\ \mapsto^* & \mathsf{msg}\ x'\ (\mathsf{b1} \cdot x''), \mathsf{proc}\ (y' \leftarrow \mathit{flip}\ x'), \mathsf{msg}\ y\ (\mathsf{b1} \cdot y'), \mathsf{proc}\ (z \leftarrow \mathit{flip}\ y) \end{aligned}$$

Because we have concurrent language, the two messages to the two processes (in the middle of the picture) can proceed independently:

$$\begin{aligned} & \mathsf{msg}\ x'\ (\mathsf{b1} \cdot x''), \mathsf{proc}\ (y' \leftarrow \mathit{flip}\ x'), \mathsf{msg}\ y\ (\mathsf{b1} \cdot y'), \mathsf{proc}\ (z \leftarrow \mathit{flip}\ y) \\ \mapsto^* & \mathsf{proc}\ (y'' \leftarrow \mathit{flip}\ x''), \mathsf{msg}\ y'(\mathsf{b0} \cdot y''), \mathsf{proc}\ (z' \leftarrow \mathit{flip}\ y'), \mathsf{msg}\ z\ (\mathsf{b0} \cdot z') \end{aligned}$$

6 Lazy Records Become External Choice

In the process language with shared memory, lazy records $\&_{i \in I}(i : \tau_i)$ write a continuation $K = (i \cdot x_i \Rightarrow P_i)_{i \in I}$ to memory. Here, in the message passing setting, a process providing a channel $x : \&_{i \in I}(i : \tau_i)$ *receives* one of the labels $j \in I$ and continuation channel $y : \tau_j$. It is dual to $\oplus_{i \in I}(i : \tau_i)$ in the sense that it just reverses the role of provider and client. We call it *external choice* since the client can make the choice which label j to send.

As for internal choice, the typing rules remain the same.

$$\frac{\Delta \Vdash P_i :: (y_i : \tau_i) \quad (\text{for all } i \in I)}{\Delta \Vdash \mathsf{case}\ x\ (i \cdot y_i \Rightarrow P_i)_{i \in I} :: (x : \&_{i \in I}(i : \tau_i))}\ \&R$$

$$\frac{(j \in I)}{1em]x : \&_{i \in I}(i : \tau_i) \Vdash x.(j \cdot y) :: (y : \tau_j)}\ \&L^0$$

Dynamically, the external choice is already handled with the rules

$$\begin{array}{lcll} \mathsf{proc}\ (c.V) & \mapsto & \mathsf{msg}\ c\ V & \text{(send)} \\ \mathsf{proc}\ (\mathsf{case}\ c\ K), \mathsf{msg}\ c\ V & \mapsto & \mathsf{proc}\ (V \triangleright K) & \text{(receive)} \end{array}$$

even though the direction of the message flow has changed: it now goes to the provider from the client.

7 Queues Revisited

Before revisiting queues, let's summarize the message-passing interpretation of the various type constructors. Note that the statics is unchanged from Lecture 23 and the dynamics (excepting only the identity) is presented in Section 4. Furthermore, each provider action implies a matching complementary client action.

Type	**Provider Action**	**Continuation Channel**
$x : \oplus_{i \in I}(i : \tau_i)$	send label j	$x' : \tau_j$
$x : \tau_1 \otimes \tau_2$	send channel $y : \tau_1$	$x' : \tau_2$
$x : 1$	send $\langle\,\rangle$	*none*
$x : \rho\alpha.\,\tau$	send fold	$x' : [\rho\alpha.\,\tau/\alpha]\tau$
$x : \&_{i \in I}(i : \tau_i)$	receive label j	$x' : \tau_j$
$x : \tau_1 \multimap \tau_2$	receive channel $y : \tau_1$	$x' : \tau_2$
$x : \delta\alpha.\,\tau$	receive fold	$x' : [\delta\alpha.\,\tau/\alpha]\tau$

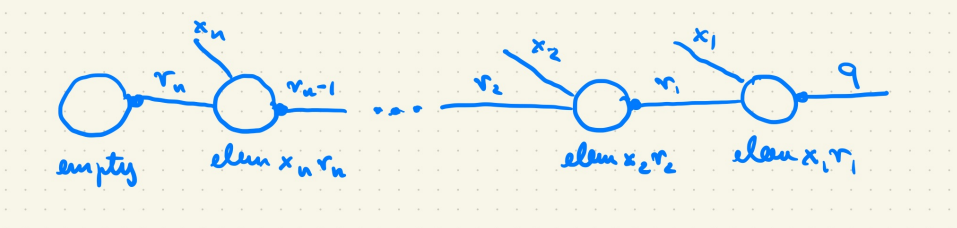
Here, the *corecursive types* $\delta\alpha.\,\tau$ is a lazy alternative to $\rho\alpha.\,\tau$ (see Exercise L20.3). Here reuses the fold constructor and pattern since it is dual to $\rho\alpha.\,\tau$.

A possible type for queues then is

$$\begin{array}{lll} \mathit{queue}\ \alpha \cong & (\mathbf{enq} : \alpha \multimap \mathit{queue}\ \alpha) & \\ \quad \& & (\mathbf{deq} : \quad (\mathbf{none} : 1) \\ & \qquad \oplus\ (\mathbf{some} : \alpha \otimes \mathit{queue}\ \alpha)) \end{array}$$

The elements of the queue are channels of type α. Compared the previous incarnation of queues (Exercise L22.3), we close the channel and terminate the providing process if there is an attempt to dequeue from the empty queue. This is expressed in the type 1. In the linear setting, we would otherwise need a separate choice for the client to deallocate the queue, and that would only be possible if the queue is empty. We didn't make this explicit here, but if we define *queue* explicitly it would be as a corecursive type $\mathit{queue} = \lambda\alpha.\, \delta q.\ \ldots$.

We implement the queue as a *bucket brigade*, with the first element at the head of the queue. A new element to be enqueued is then passed all the way to the back of the queue. A queue with elements $x_1, \ldots x_n$ can be depicted as follows.

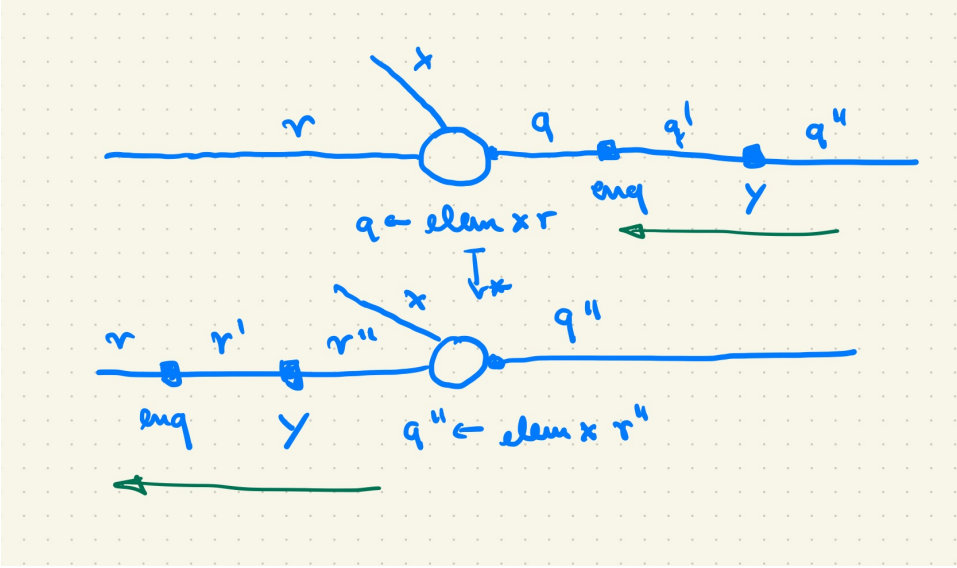


All the channels $r_1, \ldots, r_n$ have type *queue* α. We see we need two kinds of processes: one, *elem* x r that holds an element x, and *empty* marking the end of the queue.

We define the *elem* process.

$$
\begin{array}{l}
x : \alpha, r : \mathit{queue}\ \alpha \Vdash \mathit{elem} :: (q : \mathit{queue}\ \alpha) \\
q \leftarrow \mathit{elem}\ x\ r = \\
\quad \mathsf{case}\ q\ (\ \mathbf{enq} \cdot q' \Rightarrow \mathsf{case}\ q'\ (\langle y, q'' \rangle \Rightarrow r' \leftarrow r.(\mathbf{enq} \cdot r')\ ; \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad r'' \leftarrow r'.\langle y, r'' \rangle\ ; \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad q'' \leftarrow \mathit{elem}\ x\ r''\ ) \\
\qquad\qquad | \ \mathbf{deq} \cdot q' \Rightarrow q'' \leftarrow q''.\langle x, r \rangle\ ; \\
\qquad\qquad\qquad\qquad\quad q'.(\mathbf{some} \cdot q'')\ )
\end{array}
$$

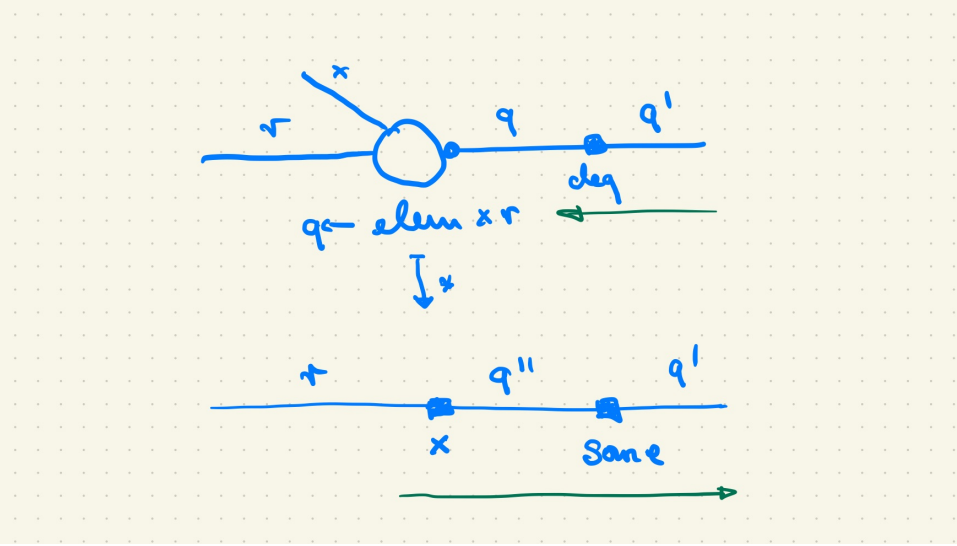
In case the client chooses to enqueue, the transition of the configuration can be depicted as follows.



As indicated by the green arrows, the messages here flow from right to left (from client to provider), unlike the bit flipping example where the flowed from left to right. As indicated in the table at the beginning of this section, messages of positive type flow from the provider to the client and message of negative type flow from the client to the provider.

In the case of the dequeue the provider has to respond to the client, so the direction of message flow changes. Moreover, the process holding the

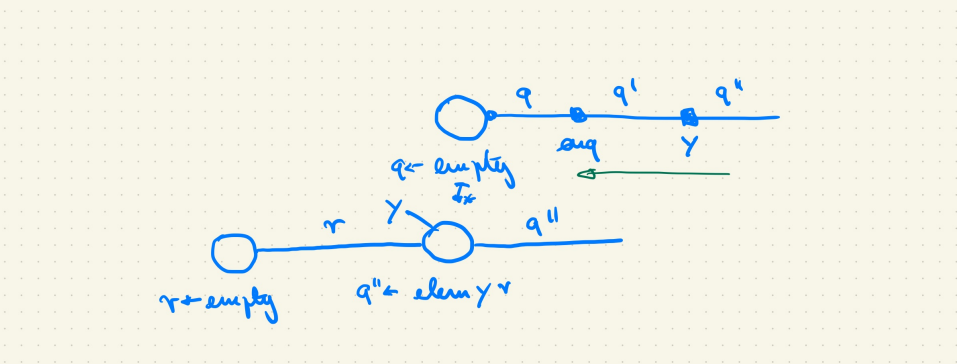
element x terminates, since that channel is returned to the client.



We did not have time to write this in lecture, but here is the code for the empty process.

$$
\begin{array}{l}
\cdot \Vdash \mathit{empty} :: (q : \mathit{queue}\ \alpha) \\
q \leftarrow \mathit{empty} = \\
\quad \mathsf{case}\ q\ (\ \mathbf{enq} \cdot q' \Rightarrow \mathsf{case}\ q'\ (\langle y, q'' \rangle \Rightarrow r \leftarrow \mathit{empty}\ ; \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad q'' \leftarrow \mathit{elem}\ y\ r\ ) \\
\qquad\quad \mid \mathbf{deq} \cdot q' \Rightarrow q'' \leftarrow q''.\langle\,\rangle\ ; \\
\qquad\qquad\qquad\quad q'.(\mathbf{none} \cdot q'')\ )
\end{array}
$$

The action of the enqueue operation for the empty queue can be depicted as follows:



8 Move Becomes Forwarding

The one construct have not discussed yet is $x \leftarrow y$. In the shared memory interpretation this either *copied* the contents of y to x (in the nonlinear version) or *moved* the contents of y to x (in the linear version). Here it *forwards messages* from one channel to another and terminates.

$$\begin{array}{lcll} \mathsf{msg}\ y\ V, \mathsf{proc}\ (x \leftarrow y) & \mapsto & \mathsf{msg}\ x\ V & \text{(positive types)} \\ \mathsf{proc}\ (x \leftarrow y), \mathsf{msg}\ x\ V & \mapsto & \mathsf{msg}\ y\ V & \text{(negative types)} \end{array}$$

9 Rule Summary

The small values V, continuations K and typing rules can be found in the Lecture 23 Rule Sheet and remain unchanged.

The statics for configurations consists of the following rules.

$$\frac{\Delta \Vdash P :: (c : \tau)}{\Delta', \Delta \Vdash \mathsf{proc}\ P :: (\Delta', c : \tau)}\ \mathsf{tp/proc} \qquad \frac{\Delta \Vdash c.V :: (d : \tau)}{\Delta', \Delta \Vdash \mathsf{msg}\ c\ V :: (\Delta', d : \tau)}\ \mathsf{tp/msg}$$

$$\frac{}{\Delta \Vdash (\cdot) :: \Delta}\ \mathsf{tp/empty} \qquad \frac{\Delta \Vdash \mathcal{C}_1 :: \Delta_1 \quad \Delta_1 \Vdash \mathcal{C}_2 :: \Delta_2}{\Delta \Vdash (\mathcal{C}_1, \mathcal{C}_2) :: \Delta_2}\ \mathsf{tp/join}$$

The dynamics has very few rules, since we have factored out the (unchanged) passing of a value V to a continuation K.

$$\begin{array}{lcll} \mathsf{proc}\ (x \leftarrow P\ ;\ Q) & \mapsto & \mathsf{proc}\ ([c/x]P), \mathsf{proc}\ ([c/x]Q) & \text{(spawn; } c \text{ fresh)} \\ \mathsf{proc}\ (c.V) & \mapsto & \mathsf{msg}\ c\ V & \text{(send)} \\ \mathsf{msg}\ c\ V, \mathsf{proc}\ (\mathsf{case}\ c\ K) & \mapsto & \mathsf{proc}\ (V \triangleright K) & \text{(receive)} \\ \mathsf{msg}\ d\ V, \mathsf{proc}\ (c \leftarrow d) & \mapsto & \mathsf{msg}\ c\ V & \text{(pos. forward)} \\ \mathsf{proc}\ (c \leftarrow d), \mathsf{msg}\ c\ V & \mapsto & \mathsf{msg}\ d\ V & \text{(neg. forward)} \end{array}$$

References

[Agh85] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. PhD thesis, Massachusetts Institute of Technology, June 1985.

[Bou92] Gérard Boudol. Asynchrony and the π-calculus. Rapport de Recherche 1702, INRIA, Sophia-Antipolis, 1992.

[CP10] Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *Proceedings of the 21st International Conference on Concurrency Theory (CONCUR 2010)*, pages 222–236, Paris, France, August 2010. Springer LNCS 6269.

[CPT16] Luís Caires, Frank Pfenning, and Bernardo Toninho. Linear logic propositions as session types. *Mathematical Structures in Computer Science*, 26(3):367–423, 2016. Special Issue on Behavioural Types.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes. *Information and Computation*, 100(1):1–77, September 1992. Parts I and II.

[PP20] Klaas Pruiksma and Frank Pfenning. Back to futures. *CoRR*, abs/2002.04607, February 2020.