Lecture Notes on Message-Passing Concurrency

15-814: Types and Programming Languages
Frank Pfenning

Lecture 21
November 15, 2018

1 Introduction

In the last lecture we have seen the sequent calculus as a calculus of proof search for natural deduction. The "informative" proof term assignment decomposed the computation into smaller steps. Today, we will take a leap and provide an interpretation of the sequent calculus based on processes that execute concurrently and pass messages between each other.

2 Destinations

Let's reconsider for a moment the informative proof terms assigned to the sequent calculus, just looking at identity and cut.

$$\frac{}{\Gamma, x : A \Vdash x : A}\ \mathsf{id} \qquad \frac{\Gamma \Vdash M : A \quad \Gamma, x : A \Vdash N : C}{\Gamma \Vdash \mathbf{let}\ x : A = M\ \mathbf{in}\ N : C}\ \mathsf{cut}$$

We can almost give this a store semantics, maybe simplifying the S machine, if we think of every variable standing in for a location at runtime. The only missing piece is that there is no destination for the result of the computation. We can fix that by also naming the right-hand side (statically with a variable, and dynamically with a destination):

$$\frac{}{\Gamma, x : A \Vdash ?? :: (y : A)}\ \mathsf{id} \qquad \frac{\Gamma \Vdash ?? :: (x : A) \quad \Gamma, x : A \Vdash ?? :: (z : C)}{\Gamma \Vdash ?? :: (z : C)}\ \mathsf{cut}$$

The proof term for the identity should copy x to y, which is also the operational interpretation of a destination expression in the S machine.

$$\frac{}{\Gamma, x : A \Vdash (y \leftarrow x) :: (y : A)}\ \mathsf{id}$$

The cut rule creates a new destination for x then runs M to fill it and N to use it.

$$\frac{\Gamma \Vdash M :: (x : A) \quad \Gamma, x : A \Vdash N :: (z : C)}{\Gamma \Vdash \mathbf{let}\ x : A = M\ \mathbf{in}\ N :: (z : C)}\ \mathsf{cut}$$

If M and N run in sequentially, this fits within the model for a functional language we have introduced so far. If M and N run in parallel, then this is the behavior of a future [Hal85]. We can develop the dynamics of the remaining proof terms under this interpretation. The proof terms represent a kind of low-level language for instructions of the S machine.

Instead of pursuing this further, we make a deceptively small change in the sequent calculus to obtain an alternate interpretation as message passing.

3 Linearity

The key reinterpretation of the judgment

$$x_1 : A_1, \ldots, x_n : A_n \Vdash P :: (z : C)$$

is that the x_i and z are *channels for communication* and P is a process. We say P *provides* channel z and *uses* channels x_i. The propositions A_i and C describe a protocol for interaction along the channel x_i and x, respectively.

The first fundamental invariant we want to preserve throughout computation is:

> ***Linearity:*** *Every channel has exactly one provider and exactly one client.*

The second one enables us to identify processes with the channels they provide:

> ***Uniqueness:*** *Every process provides exactly one channel.*

It is possible to relax both of these, but in this lecture we are concerned with the core of the computational interpretation of the (linear) sequent calculus.

Let's reconsider identity and cut in light of these invariants.

$$\frac{}{\Gamma, y : A \Vdash ?? :: (x : A)}\ \mathsf{id}$$

The process $??$ is obligated to *provide* a service following protocol A along x. It also *uses* a channel y of the same type A. One way to fulfill its obligation is to *forward* between x and y and terminate. We can also say that this process *identifies* x and y so that further communication along x will go to the provider of y, and further communication along x will go the client of y. We write this as $x \leftarrow y$ and read it as "x *is implemented by* y".

Since this process terminates by forwarding, it cannot be using any other channels. If it did, those channels would be left without a client, violating linearity! So our final rule is

$$\frac{}{y : A \Vdash (x \leftarrow y) :: (x : A)}\ \mathsf{id}$$

Let's move on to cut, not yet committing to the process expression/proof term for it.

$$\frac{\Gamma \Vdash P :: (x : A) \quad \Gamma, x : A \Vdash Q :: (z : C)}{\Gamma \Vdash ?? :: (z : C)}\ \mathsf{cut}$$

We can observe a few things about this rule. Since channels must be distinct, x is not already declared in Γ. Moreover, P provides a service of type A along x and Q is the client. Also, whatever $??$ turns out to be, it provides along z, the same channel as Q. So $??$ *spawns* a new process P that provides along a fresh channel x and continues with Q. We write this as

$$x \leftarrow P\ ;\ Q$$

Both P and Q depend on x, P being the provider and Q being the client. Before we can complete the rule, we should consider Γ. In the current form, every channel in Γ suddenly would have two clients, namely P and Q. This violates linearity, so instead we need to "split up" the context: some of the channels should be used by P and others by Q. We use the notation Δ_1, Δ_2 for joining two contexts with no overlapping names. Then we have

$$\frac{\Delta_1 \Vdash P :: (x : A) \quad \Delta_2, x : A \Vdash Q :: (z : C)}{\Delta_1, \Delta_2 \Vdash (x \leftarrow P\ ;\ Q) :: (z : C)}\ \mathsf{cut}$$

We use Δ as our notation for contexts of channels that should be used *linearly*, that is, with exactly one provider and exactly one client.

In summary, we have

$$\frac{}{y : A \Vdash (x \leftarrow y) :: (x : A)}\ \mathsf{id}$$

$$\frac{\Delta_1 \Vdash P :: (x : A) \quad \Delta_2, x : A \Vdash Q :: (z : C)}{\Delta_1, \Delta_2 \Vdash (x \leftarrow P\ ;\ Q) :: (z : C)}\ \mathsf{cut}$$

4 Intuitionistic Linear Logic

The sequent calculus we have started derives from *intuitionistic linear logic* [GL87, CCP03]. It is "intuitionistic" because the right-hand side of the sequents are singletons, thereby maintaining our uniqueness invariant. Classical linear logic [Gir87] has symmetric sequents, which has some advantages and some disadvantages for our purposes.

All of the rules we will provide in the remainder of this lecture are indeed also logical rules when one ignores the process expressions. In linear logic, a sequent $\Delta \Vdash A$ expresses that A can be proved from Δ using each antecedent in Δ *exactly once*. Often, this is explained by thinking of the antecedents as abstract *resources* that must be consumed in a proof.

In order to recover the usual expressive power of logic (in our case, intuitionistic logic), linear logic adds a modality $!A$. Only antecedents of this form may be reused or discarded in a proof. We do not develop this modality in this lecture, but might return to it in one of the remaining lectures.

5 Internal Choice

As a first *logical connective* we consider a form of disjunction, written in linear logic as $A \oplus B$. From the computational perspective, a provider of $x : A \oplus B$ should send either l or r. If the provider sends l, communication should the continue following the type A; if it sends r it should continue following B.

$$\frac{\Delta \Vdash P :: (x : A)}{\Delta \Vdash (x.l\ ;\ P) :: (x : A \oplus B)}\ \oplus R_1 \qquad \frac{\Delta \Vdash P :: (x : B)}{\Delta \Vdash (x.r\ ;\ P) :: (x : A \oplus B)}\ \oplus R_2$$

The proposition $A \oplus B$ is called *internal choice* because the provider decides whether to choose A (by sending l) or B (by sending r). Conversely, the

client must be ready to receive either l or r and then continue communication at type A or B, respectively.

$$\frac{\Delta, x : A \Vdash Q :: (z : C) \quad \Delta, x : B \Vdash R :: (z : C)}{\Delta, x : A \oplus B \Vdash (\mathbf{case}\ x\ \{l \Rightarrow Q \mid r \Rightarrow R\}) :: (z : C)}\ \oplus L$$

At this point you might, and probably *should* object: didn't we say that each antecedent in Δ should be used exactly once in a proof? Or, computational, each channel in Δ should have exactly one client? Here, it looks as if Δ is duplicated to that each channel has two clients: Q and R.

Thinking about the operational semantics clarifies why the rule must be as shown. Imagine the provider of $(x.l\ ;\ P) :: (x : A \oplus B)$ sends l to a client of x, say $\mathbf{case}\ x\ \{l \Rightarrow Q \mid r \Rightarrow R\}$. The provider continues with $P :: (x : A)$ and the client continues with Q. Now each channel used by the original client is used by Q, precisely because we have propagated all of Δ to the first branch. If the provider sends r, then the continuation R is the one that will use all these channels. So linearity is preserved in both cases. If we had split Δ into two, linearity could in fact have been violated because in each case some of the providers could be left without clients.

To formally describe the dynamics we use semantic objects of the form $\mathsf{proc}\ P\ c$ which means that process P executes providing along channel c. Just as in destination-passing style, we do not explicit record the channels that P uses—they simply occur (free) in P. In the S machine we also needed memory cells $!\mathsf{cell}\ d\ v$ and continuations $\mathsf{cont}\ d\ k\ d'$ which turn out not to be required here. In linear logic, every semantic object is in fact a process.

The possible interactions for internal choice then are described by the following two rules:

$$(\oplus C_1) \quad \mathsf{proc}\ (c.l\ ;\ P)\ c, \mathsf{proc}\ (\mathbf{case}\ c\ \{l \Rightarrow Q \mid r \Rightarrow R\})\ d \ \mapsto\ \mathsf{proc}\ P\ c, \mathsf{proc}\ Q\ d$$

$$(\oplus C_2) \quad \mathsf{proc}\ (c.r\ ;\ P)\ c, \mathsf{proc}\ (\mathbf{case}\ c\ \{l \Rightarrow Q \mid r \Rightarrow R\})\ d \ \mapsto\ \mathsf{proc}\ P\ c, \mathsf{proc}\ R\ d$$

Returning to identity and cut, we get the following rules, writing out formally what we described informally.

$$(\mathsf{id}C) \quad \mathsf{proc}\ P\ d, \mathsf{proc}\ (c \leftarrow d)\ c \ \mapsto\ \mathsf{proc}\ ([c/d]P)\ c$$

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

6 An Example: Bit Streams

Already in the fragment with identity, cut, and internal choice, we can write some interesting programs provided we have recursion, both at the

level of types and the level of processes. We add this here intuitively, to be formalized later.

Consider a type for a processes sending an infinite stream of bits 0 and 1.

$$\mathit{bits} = \oplus\{\mathsf{b0} : \mathit{bits}, \mathsf{b1} : \mathit{bits}\}$$

For simplicity, we consider this as an equality (so-called *equirecursive types*) rather then an isomorphism (*isorecursive types*), which allows us to avoid sending **fold** or **unfold** messages. We use here the generalized form of internal choice

$$\oplus\{\ell : A_\ell\}_{\ell \in L}$$

for a finite set of labels L. We have $A \oplus B = \oplus\{l : A, r : B\}$, so this is the same idea as behind disjoint sums.

We can write a process (a form of transducer) that receives a bit stream along some channel x it uses and sends a bit stream along the channel y is provides, negating every bit.

$$x : \mathit{bits} \Vdash \mathit{neg} :: (y : \mathit{bits})$$
$$\mathit{neg} = \ldots$$

The first thing *neg* has to do is to receive one bit along x, which corresponds

$$\begin{array}{l}\mathit{neg} = \mathbf{case}\ x\ (\ \mathsf{b0} \Rightarrow \ldots \\ \qquad\qquad\quad \mid \mathsf{b1} \Rightarrow \ldots\ )\end{array}$$

If we receive b0 we output b1 along y and recurse (to process the remaining stream); if we receive b1 we output b0 and recurse.

$$\begin{array}{l}\mathit{neg} = \mathbf{case}\ x\ (\ \mathsf{b0} \Rightarrow y.\mathsf{b1}\ ;\ \mathit{neg} \\ \qquad\qquad\quad \mid \mathsf{b1} \Rightarrow y.\mathsf{b0}\ ;\ \mathit{neg}\ )\end{array}$$

What about a process *and* that takes the conjunction of the two bits from corresponding streams? In each phase we have to read the two bits from the two channels, output one bit, and recurse.

$$x : \mathit{bits}, y : \mathit{bits} \Vdash \mathit{and} :: (z : \mathit{bits})$$

$$\begin{array}{l}\mathit{and} = \mathbf{case}\ x\ (\ \mathsf{b0} \Rightarrow \mathbf{case}\ y\ (\ \mathsf{b0} \Rightarrow z.\mathsf{b0}\ ;\ \mathit{and} \\ \qquad\qquad\qquad\qquad\qquad\qquad \mid \mathsf{b1} \Rightarrow z.\mathsf{b0}\ ;\ \mathit{and}\ ) \\ \qquad\qquad\quad \mid \mathsf{b1} \Rightarrow \mathbf{case}\ y\ (\ \mathsf{b0} \Rightarrow z.\mathsf{b0}\ ;\ \mathit{and} \\ \qquad\qquad\qquad\qquad\qquad\qquad \mid \mathsf{b1} \Rightarrow z.\mathsf{b1}\ ;\ \mathit{and}\ )\ )\end{array}$$

An interesting twist here is that we already know, after receiving b0 that the output will also be b0, so we can output it right away. We just need to be careful to still consume one bit along channel y, or the two input streams fall out of synch.

$$x : \mathit{bits}, y : \mathit{bits} \Vdash \mathit{and} :: (z : \mathit{bits})$$

$$\begin{array}{l}\mathit{and} = \mathbf{case}\ x\ (\ \mathsf{b0} \Rightarrow z.\mathsf{b0}\ ;\ \mathbf{case}\ y\ (\ \mathsf{b0} \Rightarrow \mathit{and} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathsf{b1} \Rightarrow \mathit{and}\ ) \\ \qquad\qquad \mid \mathsf{b1} \Rightarrow \mathbf{case}\ y\ (\ \mathsf{b0} \Rightarrow z.\mathsf{b0}\ ;\ \mathit{and} \\ \qquad\qquad\qquad\qquad\qquad\qquad \mid \mathsf{b1} \Rightarrow z.\mathsf{b1}\ ;\ \mathit{and}\ )\ )\end{array}$$

As a final example along similar lines we consider a process *compress* that compresses consecutives zeros into just one zero. The case where we see a b1 is easy: we just output it and recurse.

$$x : \mathit{bits} \Vdash \mathit{compress} :: (y : \mathit{bits})$$

$$\begin{array}{l}\mathit{compress} = \mathbf{case}\ x\ (\ \mathsf{b0} \Rightarrow \ldots \\ \qquad\qquad\qquad\qquad \mid \mathsf{b1} \Rightarrow y.\mathsf{b1}\ ;\ \mathit{compress}\ )\end{array}$$

When we see a b0 we don't know how many b0's are still to come. So we can output the first b0, but then we need to continue to ignore all following b0's until we see a b1. We need another process definition ignore for this purpose.

$$x : \mathit{bits} \Vdash \mathit{compress} :: (y : \mathit{bits})$$
$$x : \mathit{bits} \Vdash \mathit{ignore} :: (y : \mathit{bits})$$

$$\begin{array}{l}\mathit{compress} = \mathbf{case}\ x\ (\ \mathsf{b0} \Rightarrow \ldots \\ \qquad\qquad\qquad\qquad \mid \mathsf{b1} \Rightarrow y.\mathsf{b1}\ ;\ \mathit{compress}\ )\end{array}$$

$$\begin{array}{l}\mathit{ignore} = \mathbf{case}\ x\ (\ \mathsf{b0} \Rightarrow \mathit{ignore} \\ \qquad\qquad\qquad\qquad \mid \mathsf{b1} \Rightarrow y.\mathsf{b1}\ ;\ \mathit{compress}\ )\end{array}$$

At this point it only remains to fill the call to *ignore* after an output of the first b0 seen.

$$x : \mathit{bits} \Vdash \mathit{compress} :: (y : \mathit{bits})$$
$$x : \mathit{bits} \Vdash \mathit{ignore} :: (y : \mathit{bits})$$

$$\begin{array}{l}\mathit{compress} = \mathbf{case}\ x\ (\ \mathsf{b0} \Rightarrow y.\mathsf{b0}\ ;\ \mathit{ignore} \\ \qquad\qquad\qquad\qquad \mid \mathsf{b1} \Rightarrow y.\mathsf{b1}\ ;\ \mathit{compress}\ )\end{array}$$

$$\begin{array}{l}\mathit{ignore} = \mathbf{case}\ x\ (\ \mathsf{b0} \Rightarrow \mathit{ignore} \\ \qquad\qquad\qquad\qquad \mid \mathsf{b1} \Rightarrow y.\mathsf{b1}\ ;\ \mathit{compress}\ )\end{array}$$

7 Ending a Session

Viewed as types, the propositions of linear logic are called *session types*, as pioneered by Honda [Hon93]. The logical origins of session types had been in the air (see, for example, Gay and Vasconcelos [GV10]) but wasn't formally spelled out and proved until 2010 [CP10, CPT16]. The concept of a *session* is a sequence of interactions between two processes (for us, provider and client) as specified by a session type.

In the examples so far, all sessions are infinite, which is common and expected in the theory of processes. But we should also have a way to end a session after finitely many interactions. This is the role played by the type 1. As a propositions, it means the "empty" resource (or the absence of resources). Computationally, a provider of $x : 1$ can end a session and terminate, while a client waits for the session to be ended. We can also think of this as *closing* a channel of communication. To preserve our linearity invariant, the process that ends the session cannot use any other channels.

$$\frac{}{\cdot \Vdash \mathbf{close}\, x :: (x : 1)}\ 1R \qquad \frac{\Delta \Vdash Q :: (z : C)}{\Delta, x : 1 \Vdash (\mathbf{wait}\, x\ ;\ Q) :: (z : C)}\ 1L$$

The reduction:

$$(1C) \quad \mathsf{proc}\ (\mathbf{close}\, c)\ c, \mathsf{proc}\ (\mathbf{wait}\ c\ ;\ Q)\ d \quad \mapsto \quad \mathsf{proc}\ Q\ d$$

A few words on our conventions:

- Even though the semantic objects in a configuration are unordered, we always write the provider of a channel to the left of its client.
- We use P for providers, and Q for clients.
- We use x, y, z for expression variables that stand for channels, while c, d, e are used for channels as they exist as processes execute. This is the same distinction we make between variables in a functional program and destinations or memory addresses at runtime.

8 An Example: Binary Numbers

As another example we use numbers in binary form represented as a sequence of messages. This is almost like bit streams, but they can be terminated by ϵ, which represents 0 as the empty string of bits.

$$\mathit{bin} = \oplus\{\mathsf{b0} : \mathit{bin}, \mathsf{b1} : \mathit{bin}, \epsilon : 1\}$$

A process *zero* producing the representation of 0 is easy. After sending the label ϵ we have to end the session by closing the channel because the type of x at this point in the session has become 1.

$$\begin{array}{l} \cdot \Vdash \mathit{zero} :: (x : \mathit{bin}) \\ \mathit{zero} = x.\epsilon \; ; \; \mathbf{close}\, x \end{array}$$

A process that computes the successor of a binary number is more complicated.

$$\begin{array}{l} x : \mathit{bin} \Vdash \mathit{succ} :: (y : \mathit{bin}) \\ \mathit{succ} = \mathbf{case}\; x\; (\; \mathsf{b0} \Rightarrow \ldots \\ \qquad\qquad\quad \mid \mathsf{b1} \Rightarrow \ldots \\ \qquad\qquad\quad \mid \epsilon \Rightarrow \ldots\;) \end{array}$$

Let's start with the last case. The label ϵ represents 0, so we have to send along y the representation of 1, which is $\mathsf{b1}$ followed by ϵ.

$$\begin{array}{l} x : \mathit{bin} \Vdash \mathit{succ} :: (y : \mathit{bin}) \\ \mathit{succ} = \mathbf{case}\; x\; (\; \mathsf{b0} \Rightarrow \ldots \\ \qquad\qquad\quad \mid \mathsf{b1} \Rightarrow \ldots \\ \qquad\qquad\quad \mid \epsilon \Rightarrow y.\mathsf{b1} \; ; \; y.\epsilon \; ; \ldots) \end{array}$$

At this point we have $x : 1$ in the context and the successor process must provide $y : 1$. We could accomplish this by forwarding $y \leftarrow x$ or by waiting for x to close and then close y. Let's use the latter version.

$$\begin{array}{l} x : \mathit{bin} \Vdash \mathit{succ} :: (y : \mathit{bin}) \\ \mathit{succ} = \mathbf{case}\; x\; (\; \mathsf{b0} \Rightarrow \ldots \\ \qquad\qquad\quad \mid \mathsf{b1} \Rightarrow \ldots \\ \qquad\qquad\quad \mid \epsilon \Rightarrow y.\mathsf{b1} \; ; \; y.\epsilon \; ; \\ \qquad\qquad\qquad\quad \mathbf{wait}\; x \; ; \; \mathbf{close}\; y\;) \end{array}$$

In the case of $\mathsf{b0}$, *succ* just outputs $\mathsf{b1}$. Then the remaining string of bits is unchanged, so we just forward.

$$\begin{array}{l} x : \mathit{bin} \Vdash \mathit{succ} :: (y : \mathit{bin}) \\ \mathit{succ} = \mathbf{case}\; x\; (\; \mathsf{b0} \Rightarrow y.\mathsf{b1} \; ; \; y \leftarrow x \\ \qquad\qquad\quad \mid \mathsf{b1} \Rightarrow \ldots \\ \qquad\qquad\quad \mid \epsilon \Rightarrow y.\mathsf{b1} \; ; \; y.\epsilon \; ; \\ \qquad\qquad\qquad\quad \mathbf{wait}\; x \; ; \; \mathbf{close}\; y\;) \end{array}$$

When the first (lowest) bit of the input is b1 we have to output b0, but we still need to take care of the carry bit. We can do this simply by calling *succ* recursively.

$$
\begin{array}{l}
x : \mathit{bin} \Vdash \mathit{succ} :: (y : \mathit{bin}) \\
\mathit{succ} = \mathbf{case}\ x\ (\ \mathsf{b0} \Rightarrow y.\mathsf{b1}\ ;\ y \leftarrow x \\
\qquad\qquad\quad \mid \mathsf{b1} \Rightarrow y.\mathsf{b0}\ ;\ \mathit{succ} \\
\qquad\qquad\quad \mid \epsilon \Rightarrow y.\mathsf{b1}\ ;\ y.\epsilon\ ; \\
\qquad\qquad\qquad\quad \mathbf{wait}\ x\ ;\ \mathbf{close}\ y\ )
\end{array}
$$

9 External Choice

In internal choice $A \oplus B$ it is the provider who gets to choose. External choice $A \mathbin{\&} B$ let's the client choose, which means the provider has to be ready with two different branches.

$$\frac{\Delta \Vdash P_1 :: (x : A) \quad \Delta \Vdash P_2 :: (x : B)}{\Delta \Vdash (\mathbf{case}\ x\ \{l \Rightarrow P_1 \mid r \Rightarrow P_2\}) :: (x : A \mathbin{\&} B)}\ \&R$$

$$\frac{\Delta, x : A \Vdash Q :: (z : C)}{\Delta, x : A \mathbin{\&} B \Vdash x.l\ ;\ Q :: (z : C)}\ \&L_1 \qquad \frac{\Delta, x : B \Vdash Q :: (z : C)}{\Delta, x : A \mathbin{\&} B \Vdash x.r\ ;\ Q :: (z : C)}\ \&L_2$$

We see that internal choice and external choice are quite symmetric in the linear sequent calculus, while in natural deduction (and functional programming) they look much further apart. The transition rules follow the pattern of internal choice, with the role of provider and client swapped.

$$
\begin{array}{lll}
(\&C_1) & \mathsf{proc}\ (\mathbf{case}\ c\ \{l \Rightarrow P_1 \mid r \Rightarrow P_2\})\ c, \mathsf{proc}\ (c.l\ ;\ Q)\ d & \mapsto & \mathsf{proc}\ P_1\ c, \mathsf{proc}\ Q\ d \\
(\&C_2) & \mathsf{proc}\ (\mathbf{case}\ c\ \{l \Rightarrow P_1 \mid r \Rightarrow P_2\})\ c, \mathsf{proc}\ (c.r\ ;\ Q)\ d & \mapsto & \mathsf{proc}\ P_2\ c, \mathsf{proc}\ Q\ d
\end{array}
$$

With external choice we can implement a *counter* that can take two labels: increment (inc) that increments its internal value and val after which it streams the bits making up the current value of the counter. In the latter case, the counter is also destroyed, so with this interface we can request its value only once.

$$\mathit{ctr} = \&\{\mathsf{inc} : \mathit{ctr}, \mathsf{val} : \mathit{bin}\}$$

We implement the counter as a process that holds a binary number as an internal data structure, which is implemented as a process of type *bin*.

$y : \mathit{bin} \Vdash \mathit{counter} :: (x : \mathit{ctr})$

We say y represents an internal data structure because *counter* is the *only* client of it (by linearity), so no other process can access it.

The counter distinguishes cases based on the label received along x. After all, it is an external choice so we need to know what the client requests.

$$
\begin{array}{l}
y : \mathit{bin} \Vdash \mathit{counter} :: (x : \mathit{ctr}) \\
\mathit{counter} = \mathbf{case}\ x\ (\ \mathsf{inc} \Rightarrow \ldots \\
\qquad\qquad\qquad\quad \mid \mathsf{val} \Rightarrow \ldots\ )
\end{array}
$$

We increment such a counter by using the *succ* process from the previous example. We can do this by spawning a new successor process without actually receiving anything from the stream y.

$$
\begin{array}{l}
y : \mathit{bin} \Vdash \mathit{counter} :: (x : \mathit{ctr}) \\
\mathit{counter} = \mathbf{case}\ x\ (\ \mathsf{inc} \Rightarrow y' \leftarrow \mathit{succ} \leftarrow y\ ; \\
\qquad\qquad\qquad\qquad\quad \ldots \\
\qquad\qquad\qquad\quad \mid \mathsf{val} \Rightarrow \ldots\ )
\end{array}
$$

In order to spawn a new process and not become confused with different variables called y, we use the notation

$$x \leftarrow f \leftarrow y_1, \ldots, y_n\ ;\ P$$

for a cut, passing channels $y_1, \ldots, y_n$ to process f that provides along the fresh channel x that can be used in P. Note that due to linearity, $y_1, \ldots, y_n$ will no longer be available since now the freshly spawned instance of f is their client.

We use this same notation for the recursive call to *counter*.

$$
\begin{array}{l}
y : \mathit{bin} \Vdash \mathit{counter} :: (x : \mathit{ctr}) \\
\mathit{counter} = \mathbf{case}\ x\ (\ \mathsf{inc} \Rightarrow y' \leftarrow \mathit{succ} \leftarrow y\ ; \\
\qquad\qquad\qquad\qquad\quad x \leftarrow \mathit{counter} \leftarrow y' \\
\qquad\qquad\qquad\quad \mid \mathsf{val} \Rightarrow \ldots\ )
\end{array}
$$

Just so we don't make a mess of the bound variables, we define *counter* as depending on two channels, x and y. When *counter* is called, x will be created fresh since a new process is spanned, and y will be passed to this this new process.

$$
\begin{array}{l}
y : \mathit{bin} \Vdash \mathit{counter} :: (x : \mathit{ctr}) \\
x \leftarrow \mathit{counter} \leftarrow y = \\
\quad \mathbf{case}\ x\ (\ \mathsf{inc} \Rightarrow y' \leftarrow \mathit{succ} \leftarrow y\ ; \\
\qquad\qquad\qquad\quad x \leftarrow \mathit{counter} \leftarrow y' \\
\qquad\quad \mid \mathsf{val} \Rightarrow \ldots\ )
\end{array}
$$

Now we can fill in the last case, where we just forward to the privately held binary number, thereby terminating and communicating the number back to the client.

$$
\begin{array}{l}
y : \mathit{bin} \Vdash \mathit{counter} :: (x : \mathit{ctr}) \\
\mathit{counter} = \mathbf{case}\ x\ (\ \mathsf{inc} \Rightarrow y' \leftarrow \mathit{succ} \leftarrow y\ ; \\
\qquad\qquad\qquad\qquad\quad x \leftarrow \mathit{counter} \leftarrow y' \\
\qquad\qquad\qquad \mid \mathsf{val} \Rightarrow x \leftarrow y\ )
\end{array}
$$

To assure you that this is type-correct, we see that the type of counter, after seeing the val becomes *bin*, which is exactly the type of y.

We can create a new counter with some initial value by calling this process and passing it a process holding the initial value. For example,

$$
\begin{array}{l}
z \leftarrow \mathit{zero}\ ; \\
c \leftarrow \mathit{counter} \leftarrow z\ ; \\
P
\end{array}
$$

creates a new counter c that can be used in P. The channel z, on the other hand, is not accessible there because it has been passed to the instance of *counter*.

More formally, if we see a type declaration and a definition

$$
\begin{array}{l}
y_1 : B_1, \ldots, y_n : B_n \Vdash f :: (x : A) \\
x \leftarrow f \leftarrow y_1, \ldots, y_n = P
\end{array}
$$

then we check

$$y_1 : B_1, \ldots, y_n : B_n \Vdash P :: (x : A)$$

and every call

$$x' \leftarrow f \leftarrow y'_1, \ldots, y'_n$$

is executed as

$$x' \leftarrow [x'/x, y'_1/y_1, \ldots, y'_n/y_n]P$$

A tail call (which has no continuation)

$$x \leftarrow f \leftarrow y'_1, \ldots, y'_n$$

is syntactically expanded into a call, followed by a forward

$$x' \leftarrow f \leftarrow y'_1, \ldots y'_n \ ;$$
$$x \leftarrow x'$$

We can now rewrite the earlier definitions in this style, for consistency. We only show this for the processes on binary numbers.

$$\mathit{bin} = \oplus\{\mathsf{b0} : \mathit{bin}, \mathsf{b1} : \mathit{bin}, \epsilon : 1\}$$

$$\cdot \Vdash \mathit{zero} :: (x : \mathit{bin})$$

$$x \leftarrow \mathit{zero} = x.\epsilon \ ; \ \mathbf{close}\, x$$

$$x : \mathit{bin} \Vdash \mathit{succ} :: (y : \mathit{bin})$$

$$\begin{array}{l} y \leftarrow \mathit{succ} \leftarrow x = \\ \quad \mathbf{case}\ x\ (\ \mathsf{b0} \Rightarrow y.\mathsf{b1}\ ;\ y \leftarrow x \\ \qquad\quad \mid \mathsf{b1} \Rightarrow y.\mathsf{b0}\ ;\ \mathit{succ} \\ \qquad\quad \mid \epsilon \Rightarrow y.\mathsf{b1}\ ;\ y.\epsilon\ ; \\ \qquad\qquad\quad \mathbf{wait}\ x\ ;\ \mathbf{close}\ y\ ) \end{array}$$

Taking stock, we see that *external choice* provides an object-oriented style of concurrent programming where we send messages to objects and may (or may not) receive replies. In contrast, *internal choice* looks more like functional programming, done concurrently: instead of representing data in memory, they are represented via messages. However, nonterminating process such as transducers make perfect sense because we care about the interactive behavior of processes and not just a final value.

References

[CCP03] Bor-Yuh Evan Chang, Kaustuv Chaudhuri, and Frank Pfenning. A judgmental analysis of linear logic. Technical Report CMU-CS-03-131R, Carnegie Mellon University, Department of Computer Science, December 2003.

[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.

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[GL87] Jean-Yves Girard and Yves Lafont. Linear logic and lazy computation. In H. Ehrig, R. Kowalski, G. Levi, and U. Montanari, editors, *Proceedings of the International Joint Conference on Theory and Practice of Software Development*, volume 2, pages 52–66, Pisa, Italy, March 1987. Springer-Verlag LNCS 250.

[GV10] Simon J. Gay and Vasco T. Vasconcelos. Linear type theory for asynchronous session types. *Journal of Functional Programming*, 20(1):19–50, January 2010.

[Hal85] Robert H. Halstead. Multilisp: A language for parallel symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–539, October 1985.

[Hon93] Kohei Honda. Types for dyadic interaction. In *4th International Conference on Concurrency Theory*, CONCUR'93, pages 509–523. Springer LNCS 715, 1993.