Lecture Notes on
Ordered Proofs as Concurrent Programs

15-317: Constructive Logic
Frank Pfenning

Lecture 24
November 30, 2017

1 Introduction

In this lecture we begin with a summary of the correspondence between proofs and programs for subsingleton logic, carry out some new examples, and then consider how the interpretation might be generalized to the case of ordered logic with more than one antecedent.

2 Concurrent Subsingleton Programs

Types	A	$::=$	$\oplus\{l_i : A_i\}_{i \in I}$	internal choice	
		$\mid$	$\&\{l_i : A_i\}_{i \in I}$	external choice	
		$\mid$	$\mathbf{1}$	termination	
Processes	P, Q	$::=$	$\leftrightarrow$	forward	id
		$\mid$	$(P \mid Q)$	compose	cut
		$\mid$	$\mathsf{R}.l_k \; ; P$	send label right	$\oplus R_k$
		$\mid$	$\mathsf{caseL}(l_i \Rightarrow Q_i)_{i \in I}$	receive label left	$\oplus L$
		$\mid$	$\mathsf{caseR}(l_i \Rightarrow P_i)_{i \in I}$	receive label right	$\& R$
		$\mid$	$\mathsf{L}.l_k \; ; Q$	send label left	$\& L_k$
		$\mid$	closeR	close and notify right	$\mathbf{1}R$
		$\mid$	$\mathsf{waitL} \; ; Q$	wait on close left	$\mathbf{1}L$

We also allow mutually recursive type definitions $\alpha = A$ which must be *contractive*, that is, A must be of the form $\oplus\{\ldots\}$, $\&\{\ldots\}$, or $\mathbf{1}$. We treat a type name as equal to its definition and will therefore silently replace it. The usual manner of making this more explicit is to use types of the form $\mu\alpha.A$, but we forego this exercise here.

Similarly, we allow mutually recursive process definitions of variables X as processes P in the form $\omega \vdash X = P : A$. Collectively, these constitute the program $\mathcal{P}$. We fix a global program $\mathcal{P}$ so that the typing judgment, formally, is $\omega \vdash_{\mathcal{P}} P : A$ where we assume that $\omega \vdash_{\mathcal{P}} Q : A$ for every definition $\omega \vdash X = Q : A$ in $\mathcal{P}$. Since $\mathcal{P}$ does not change in any typing derivation, we omit this subscript in the rules.

$$\frac{}{A \vdash \leftrightarrow : A}\ \mathsf{id}_A \qquad \frac{\omega \vdash P : A \quad A \vdash Q : C}{\omega \vdash (P \mid Q) : C}\ \mathsf{cut}_A$$

$$\frac{\omega \vdash P : A_k \quad (k \in I)}{\omega \vdash (\mathsf{R}.l_k \ ; P) : \oplus\{l_i : A_i\}_{i \in I}}\ \oplus R_k \qquad \frac{A_i \vdash Q_i : C \quad (\text{for all } i \in I)}{\oplus\{l_i : A_i\}_{i \in I} \vdash \mathsf{caseL}(l_i \Rightarrow Q_i)_{i \in I} : C}\ \oplus L$$

$$\frac{\omega \vdash P_i : A_i \quad (\text{for all } i \in I)}{\omega \vdash \mathsf{caseR}(l_i \Rightarrow P_i)_{i \in I} : \&\{l_i : A_i\}_{i \in I}}\ \& R \qquad \frac{A_k \vdash Q : C \quad (k \in I)}{\&\{l_i : A_i\}_{i \in I} \vdash (\mathsf{L}.l_k \ ; Q) : C}\ \& L_k$$

$$\frac{}{\cdot \vdash \mathsf{closeR} : \mathbf{1}}\ \mathbf{1}R \qquad \frac{\cdot \vdash Q : C}{\mathbf{1} \vdash \mathsf{waitL} \ ; Q : C}\ \mathbf{1}L$$

$$\frac{(\omega \vdash X = P : A) \in \mathcal{P}}{\omega \vdash X : A}\ X$$

For the synchronous operational semantics presented via ordered inference, we use ephemeral propositions $\mathsf{proc}(P)$ which expresses the current state of an executing process P. We also import the process definitions

$X = P$ as persistent propositions $\underline{\mathsf{def}}(X, P)$.

$$\frac{\leftrightarrow}{\cdot}\ \mathsf{fwd} \qquad \frac{(P \mid Q)}{P \mid Q}\ \mathsf{cmp}$$

$$\frac{(\mathsf{R}.l_k\ ;\ P) \mid (\mathsf{caseL}(l_i \Rightarrow Q_i)_{i\in I})}{P \mid Q_k}\ \oplus C$$

$$\frac{(\mathsf{caseR}(l_i \Rightarrow P_i)_{i\in I}) \mid (\mathsf{L}.l_k\ ;\ Q)}{P_k \mid Q}\ \& C$$

$$\frac{\mathsf{closeR} \mid (\mathsf{waitL}\ ;\ Q)}{Q}\ \mathbf{1}C$$

$$\frac{X \quad \underline{\mathsf{def}}(X, P)}{P}\ \mathsf{def}$$

3 Computing with Binary Numbers

We return to the example of numbers in binary notation. A number such as $(11)_{10} = (1011)_2$ is represented as the ordered context

$$\mathsf{e} \cdot \mathsf{b1} \cdot \mathsf{b0} \cdot \mathsf{b1} \cdot \mathsf{b1}$$

Increment can be specified using ordered inference by adding an inc proposition on the right and the following rules of inference

$$\frac{\mathsf{b0}\ \ \mathsf{inc}}{\mathsf{b1}} \qquad \frac{\mathsf{b1}\ \ \mathsf{inc}}{\mathsf{inc}\ \ \mathsf{b0}} \qquad \frac{\mathsf{e}\ \ \mathsf{inc}}{\mathsf{e}\ \ \mathsf{b1}}$$

For example, we may make the following inferences

$$\cfrac{\cfrac{\cfrac{\mathsf{e} \cdot \mathsf{b1} \cdot \mathsf{b0} \cdot \mathsf{b1} \cdot \mathsf{b1} \cdot \mathsf{inc}}{\mathsf{e} \cdot \mathsf{b1} \cdot \mathsf{b0} \cdot \mathsf{b1} \cdot \mathsf{inc} \cdot \mathsf{b0}}}{\mathsf{e} \cdot \mathsf{b1} \cdot \mathsf{b0} \cdot \mathsf{inc} \cdot \mathsf{b0} \cdot \mathsf{b0}}}{\mathsf{e} \cdot \mathsf{b1} \cdot \mathsf{b1} \cdot \mathsf{b0} \cdot \mathsf{b0}}$$

In this lecture we are interested in formulating this kind of computation via message-passing concurrency. This means we have to identify which of the propositions e, b0, b1 and inc are to be viewed as *messages* and which are to be viewed as *processes*. This choice is not uniquely determined, but can lead to very different styles of programs.

For want of a better name, we will call our two styles *functional* and *object-oriented*. In the functional style, e, b0, and b1 are messages and inc is a process. In the object-oriented style e, b0, and b1 are processes and inc is a message.

4 Quasi-Functional Increment

In the quasi-functional version, inc is implemented as a process *increment* that receives a stream of bits from the left representing the number n and produces a stream of bits on the right representing $n + 1$.

First, streams of bits (type *bin*) are represented very similarly to words from the previous lecture.

$$\begin{array}{l} \mathit{bin} = \oplus\{\mathsf{b0} : \mathit{bin}, \mathsf{b1} : \mathit{bin}, \mathsf{e} : \mathbf{1}\} \\ \mathit{bin} \vdash \mathit{incr} : \mathit{bin} \end{array}$$

The type of *incr* expresses that it transforms one number into another. The type dictates that *incr* will have to start by reading from the left.

$$\begin{array}{lll} \mathit{incr} = \mathsf{caseL} & (& \mathsf{b0} \Rightarrow \ldots \\ & \mid & \mathsf{b1} \Rightarrow \ldots \\ & \mid & \mathsf{e} \Rightarrow \ldots) \end{array}$$

In the case we receive b0 we have to output b1 *and we are done*. What we mean here by "being done" is that from then on, the remaining input bits are passed on unchanged. We can implement this with an identity process, or with forwarding, where the latter is more efficient and also more concise.

$$\begin{array}{lll} \mathit{incr} = \mathsf{caseL} & (& \mathsf{b0} \Rightarrow \mathsf{R.b1} \; ; \leftrightarrow \\ & \mid & \mathsf{b1} \Rightarrow \ldots \\ & \mid & \mathsf{e} \Rightarrow \ldots) \end{array}$$

In the case we receive b1 we have to send b0, but because of the required carry we still have to incr the remainder of the input stream of bits. This turns into a recursive call to *incr*.

$$\begin{array}{lll} \mathit{incr} = \mathsf{caseL} & (& \mathsf{b0} \Rightarrow \mathsf{R.b1} \; ; \leftrightarrow \\ & \mid & \mathsf{b1} \Rightarrow \mathsf{R.b0} \; ; \mathit{incr} \\ & \mid & \mathsf{e} \Rightarrow \ldots) \end{array}$$

When the input stream is empty (which represents the integer 0), we have to output the integer 1, which is b1 followed by e. After that, both input and output stream have type **1**, so we can terminate by forwarding.

$$\begin{array}{l} \mathit{incr} = \mathsf{caseL}\ (\ \mathsf{b0} \Rightarrow \mathsf{R.b1}\ ;\ \leftrightarrow \\ \qquad\qquad\quad \mid \mathsf{b1} \Rightarrow \mathsf{R.b0}\ ;\ \mathit{incr} \\ \qquad\qquad\quad \mid \mathsf{e} \Rightarrow \mathsf{R.b1}\ ;\ \mathsf{R.e}\ ;\ \leftrightarrow) \end{array}$$

5 Quasi-Object-Oriented Increment

The other possibility of interpreting our logical specification as message-passing concurrent computation is to turn e, b0 and b1 into processes, and inc into a message. A number such as

$$\mathsf{e} \cdot \mathsf{b1} \cdot \mathsf{b0} \cdot \mathsf{b1} \cdot \mathsf{b1}$$

then constitutes a configuration of five processes

$$\mathit{emp} \mid \mathit{bit1} \mid \mathit{bit0} \mid \mathit{bit1} \mid \mathit{bit1}$$

We can think of each of these processes as an "object", where we can send a message to the rightmost object only. Because we can only increment it, we think of each of these objects as a counter. At the moment, the only message we can send is an increment message inc, so we have

$$\begin{array}{l} \mathit{counter} = \&\{\mathsf{inc} : \mathit{counter}\} \\ \cdot \vdash \mathit{emp} : \mathit{counter} \\ \mathit{counter} \vdash \mathit{bit0} : \mathit{counter} \\ \mathit{counter} \vdash \mathit{bit1} : \mathit{counter} \end{array}$$

Let's start with the *bit0* process: it simply absorbes the *inc* message and turns into *bit1*:

$$\mathit{bit0} = \mathsf{caseR}\ (\mathsf{inc} \Rightarrow \mathit{bit1})$$

The *bit1* process has to turn into a *bit0* process, but is also has to send an increment message to its left, representing the carry.

$$\mathit{bit1} = \mathsf{caseR}\ (\mathsf{inc} \Rightarrow \mathsf{L.inc}\ ;\ \mathit{bit0})$$

Finally, *emp* does not have to send on any message, but spawn a new process and continue as *bit1*:

$$\mathit{emp} = \mathsf{caseR}\ (\mathsf{inc} \Rightarrow \mathit{empty} \mid \mathit{bit1})$$

6 Converting Between Styles

Here is a summary of the types and code so far.

Increment in quasi-functional style.

$bin = \oplus\{\mathsf{b0} : bin, \mathsf{b1} : bin, \mathsf{e} : \mathbf{1}\}$
$bin \vdash incr : bin$

$$\begin{aligned} incr = \mathsf{caseL}\ (\ & \mathsf{b0} \Rightarrow \mathsf{R.b1}\ ;\ \leftrightarrow \\ \mid\ & \mathsf{b1} \Rightarrow \mathsf{R.b0}\ ;\ incr \\ \mid\ & \mathsf{e} \Rightarrow \mathsf{R.b1}\ ;\ \mathsf{R.e}\ ;\ \leftrightarrow) \end{aligned}$$

There is concurrency here in that multiple increment processes can be active at the same time, processing the incoming bits in a pipeline.

A counter in quasi-object-oriented style.

$counter = \&\{\mathsf{inc} : counter\}$
$\cdot \vdash emp : counter$
$counter \vdash bit0 : counter$
$counter \vdash bit1 : counter$

$bit0 = \mathsf{caseR}\ (\mathsf{inc} \Rightarrow bit1)$
$bit1 = \mathsf{caseR}\ (\mathsf{inc} \Rightarrow \mathsf{L.inc}\ ;\ bit0)$
$emp = \mathsf{caseR}\ (\mathsf{inc} \Rightarrow empty \mid bit1)$

Here, concurrency is embodied in multiple increment messages being in flight at the same time as they flow through the network of processes from right to left.

Conversions between representations. To implement conversions between these representation means to implement to processes that mediate between the types.

$counter \vdash value : bin$
$bin \vdash toctr : counter$

Extracting the value of a counter. First, *value* which extracts the value from the counter to its left. In order to implement this, we need to add a new kind of message to the *counter* interface, let's call is *val*.

$$\begin{array}{l} \mathit{counter} = \&\{\mathsf{inc} : \mathit{counter}, \\ \qquad\qquad\ \ \mathsf{val} : \mathit{bin}\} \end{array}$$

We then extend the implementations of *bit0*, *bit1* and *emp* to account for this new kind of message.

$$\begin{array}{l} \mathit{bit0} = \mathsf{caseR}\ (\ \mathsf{inc} \Rightarrow \mathit{bit1} \\ \qquad\qquad\qquad \mid \mathsf{val} \Rightarrow \mathsf{R.b0}\ ;\ \mathsf{L.val}\ ;\ \leftrightarrow) \end{array}$$

$$\begin{array}{l} \mathit{bit1} = \mathsf{caseR}\ (\ \mathsf{inc} \Rightarrow \mathsf{L.inc}\ ;\ \mathit{bit0} \\ \qquad\qquad\qquad \mid \mathsf{val} \Rightarrow \mathsf{R.b1}\ ;\ \mathsf{L.val}\ ;\ \leftrightarrow) \end{array}$$

$$\begin{array}{l} \mathit{emp} = \mathsf{caseR}\ (\ \mathsf{inc} \Rightarrow \mathit{empty} \mid \mathit{bit1} \\ \qquad\qquad\qquad \mid \mathsf{val} \Rightarrow \mathsf{R.e}\ ;\ \leftrightarrow) \end{array}$$

Viewing a binary number as a counter. In our first implementation, a counter exists as a whole sequence of $\log_2(n+1)$ processes to hold the number n, each process holding one bit. In this implementation the counter actually receives a stream of bits to its left and behaves as a counter to its client.

Let's first look at the case when the counter receives an increment message inc. In that case we have to spawn a new increment *process* incr to increment the stream of bits coming from the left. These two fit together because

$$\frac{\mathit{bin} \vdash \mathit{incr} : \mathit{bin} \quad \mathit{bin} \vdash \mathit{toctr} : \mathit{counter}}{\mathit{bin} \vdash (\mathit{incr} \mid \mathit{toctr}) : \mathit{counter}}$$

$\mathit{bin} \vdash \mathit{toctr} : \mathit{counter}$

$$\begin{array}{l} \mathit{toctr} = \mathsf{caseR}\ (\ \mathsf{inc} \Rightarrow \mathit{incr} \mid \mathit{toctr} \\ \qquad\qquad\qquad \mid \mathsf{val} \Rightarrow \ldots) \end{array}$$

If the counter receives a val message it simply forwards, since it already holds the counter value as a stream to its left.

$\mathit{bin} \vdash \mathit{toctr} : \mathit{counter}$

$$\begin{array}{l} \mathit{toctr} = \mathsf{caseR}\ (\ \mathsf{inc} \Rightarrow \mathit{incr} \mid \mathit{toctr} \\ \qquad\qquad\qquad \mid \mathsf{val} \Rightarrow \leftrightarrow) \end{array}$$

7 From Subsingleton to Ordered Logic

The style of implementation we have discussed so far works well when the problem allows all the processes to be assembled in a straight line, communicating only with its immediate neighbors. But what if we want to implement a tree? Or operate on two streams such as adding two binary numbers?

In order to support more complex topologies of process, we generalize from subsingleton logic to ordered logic. The difference is only that we allow multiple antecedents. This is a big change since we immediately obtain four new connectives: over ($A \mathbin{/} B$), under ($A \setminus B$), fuse ($A \bullet B$), and twist ($A \circ B$).

Before we get to their operational meaning, let's reconsider the basic judgment. The first attempt is to generalize from

$$A \vdash P : B$$

to

$$A_1 \cdots A_n \vdash P : B$$

The problem now is: How can P address A_i if it wants to send or receive a message from it? For example, several of these types might be internal choice, and P could receive a label from any of them. In subsingleton logic, there was only (at most) a single process to the left, so this was unambiguous.

We could address this by saying, for example, that P received from the ith process, essentially numbering the antecedents. This quickly becomes unwieldy, both in practice and in theory. Or we might say that P can only communicate with, say, A_n or A_1, the extremal processes in the antecedents. However, this appears too restrictive. Instead, we uniquely label each antecedent as well as the succedent[1] with a *channel name*.

$$(x_1{:}A_1) \cdots (x_n{:}A_n) \vdash P :: (y{:}B)$$

We read this as

> *Process P provides a service of type B along channel y and uses channels x_i of type A_i.*

Since we are still in ordered logic, the order of the antecedents matter, and we will see later in which way. We abbreviate it as $\Omega \vdash P :: (y{:}B)$, overloading Ω to stand either for just an ordered sequence of antecedent or one where each antecedent is labeled.

We now generalize each of the rules from before.

[1]Not strictly necessary, since the conclusion remains a singleton, but convenient to correlate providers with their clients through a private shared channel.

Cut. Instead of simply writing $P \mid Q$, the two processes P and Q share a private channel.

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

As a point of notation, we subscript processes variables such as P or Q with *bound variables* if they are allowed to occur in them. In the process expression $x \leftarrow P_x\ ;\ Q_x$, the variable x is bound and occurs on both side, because it is a channel connecting the two processes. We almost maintain the invariant that all channel names in the antecedent and succedent are distinct, possibly renaming bound variable silently to maintain that.

Operationally, the process executing $(x \leftarrow P_x\ ;\ Q_x)$ continues as Q_x while spawning a new process P_x. This interpretation is meaningful since both $(x \leftarrow P_x\ ;\ Q_x)$ and Q_x offer service C along z. This asymmetry in the operational interpretation comes from the asymmetry of ordered logic (and intuitionistic logic in general) with multiple antecedents but at most one succedent.

In order to define the operational semantics, we write write $\mathsf{proc}(x, P)$ if the process P provides along channel x, which is to say it is typed as $\Omega \vdash P :: (x{:}A)$ for some Ω and A. This is useful to track communications. Then for cut we have the generalized rule of composition

$$\frac{\mathsf{proc}(z, x \leftarrow P_x\ ;\ Q_x)}{\mathsf{proc}(w, P_w) \quad \mathsf{proc}(z, Q_w)}\ \mathsf{cmp}^w$$

We write cmp^w to remind ourselves that the channel w must be globally "fresh": it is not allowed to occur anywhere else in the process configuration.

Identity. The identity rule could just be

$$\frac{}{y{:}A \vdash \leftrightarrow :: (x{:}A)}\ \mathsf{id}$$

based on the idea that x and y are known at this point in a proof so they don't need to be mentioned. Experience dictates that easily irecognizing whenever channels are used makes programs much more readable, so we write

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

and read is as x *is implemented by* y or x *forwards to* y.

There are various levels of detail in the operational semantics for describing identity in the presence of channel names. We cannot simply terminate the process, but we need to actively connect x with y. One way to do this is to globally identify them, which we can do in ordered inference by using equality (which we have not introduced yet).

$$\frac{\mathsf{proc}(x, x \leftarrow y)}{x = y}\ \mathsf{fwd}$$

Internal Choice. This should be straightforward: instead of sending a label "to the right", we send it along the channel the process provides.

$$\frac{\Omega \vdash P :: (x{:}A_k) \quad (k \in I)}{\Omega \vdash (x.l_k \ ; P) :: (x : \oplus\{l_i{:}A_i\}_{i\in I})}\ \oplus R_k$$

Conversely, for the left rule we just receive along a channel of the right type, rather than receiving from the right.

$$\frac{\Omega_L\ (x{:}A_i)\ \Omega_R \vdash Q_i :: (z{:}C) \quad (\forall i \in I)}{\Omega_L\ (x{:}\oplus\{l_i{:}A_i\}_{i\in I})\ \Omega_R \vdash \mathsf{case}\ x\ (l_i \Rightarrow Q_i)_{i\in I} :: (z{:}C)}\ \oplus L$$

Communication of the label goes through a channel. We only show the synchronous version:

$$\frac{\mathsf{proc}(x, x.l_k \ ; P) \quad \mathsf{proc}(z, \mathsf{case}\ x\ (l_i \Rightarrow Q_i)_{i\in I})}{\mathsf{proc}(x, P) \quad \mathsf{proc}(z, Q_k)}\ \oplus C$$

The fly in the ointment here is that these two processes may actually not be next to each other, because a client can not be next to all of its providers now that there is more than one.

One possible solution is to send messages (asynchronously) and allow them to be move past other messages and processes. This, however, does not seem a faithful representation of channel behavior, and a single communication could take many steps of exchange. A simpler solution is to retreat to *linear inference* where the order of the propositions no longer matters. We have used this, for example, to describe the spanning tree construction, Hamiltonian cycles, blocks world, etc. Now we reuse it for the operational semantics. Our earlier rules for cut and identity should also be reinterpreted in linear and not ordered inference.

External Choice. This is symmetric to internal choice and therefore boring and postponed.

Unit. The previous pattern generalizes nicely: instead of closeR and waitL we close and wait on a channel.

$$\frac{}{\cdot \vdash \mathsf{close}\ x :: (x{:}\mathbf{1})}\ \mathbf{1}R \qquad \frac{\Omega_L\ \Omega_R \vdash Q :: (z{:}C)}{\Omega_L\ (x{:}\mathbf{1})\ \Omega_R \vdash (\mathsf{wait}\ x\ ;\ Q) :: (z{:}C)}\ \mathbf{1}L$$

$$\frac{\mathsf{proc}(x, \mathsf{close}\ x) \quad \mathsf{proc}(z, \mathsf{wait}\ x\ ;\ Q)}{\mathsf{proc}(z, Q)}\ \mathbf{1}C$$

Fuse. The natural right rule for fuse has two premises.

$$\frac{\Omega_1 \vdash A \quad \Omega_2 \vdash B}{\Omega_1\ \Omega_2 \vdash A \bullet B}\ \bullet R$$

In order to avoid spawning a new process in this rule, we are looking for an sufficient one-premise version. We accomplish this by requiring that either of the two premises must be the identity. So either $\Omega_1 = A$ or $\Omega_2 = B$. These considerations yield:

$$\frac{\Omega \vdash B}{A\ \Omega \vdash A \bullet B}\ \bullet R^* \qquad \frac{\Omega \vdash A}{\Omega\ B \vdash A \bullet B}\ \bullet R^\dagger$$

Rather arbitrarily we pick the first, which yields the following pair of right and left rules for $A \bullet B$

$$\frac{\Omega \vdash B}{A\ \Omega \vdash A \bullet B}\ \bullet R^* \qquad \frac{\Omega_L\ A\ B\ \Omega_R \vdash C}{\Omega_L\ (A \bullet B)\ \Omega_R \vdash C}\ \bullet L$$

Again, we can ask which of the rules carries information, and here it is $\bullet R^*$ which sends. Filling in channel names, we see that once again a channel is sent and received.

$$\frac{\Omega \vdash P :: (x{:}B)}{(w{:}A)\ \Omega \vdash (\mathsf{send}\ x\ w\ ;\ P) :: (x{:}A \bullet B)}\ \bullet R^* \quad \frac{\Omega_L\ (y{:}A)\ (x{:}B)\ \Omega_R \vdash Q_y :: (z{:}C)}{\Omega_L\ (x{:}A \bullet B)\ \Omega_R \vdash (y \leftarrow \mathsf{recv}\ x\ ;\ Q_y) :: (z{:}C)}\ \bullet L$$

The computation rule implements the intended operational behavior directly.

$$\frac{\mathsf{proc}(x, \mathsf{send}\ x\ w\ ;\ P) \quad \mathsf{proc}(z, y \leftarrow \mathsf{recv}\ x\ ;\ Q_y)}{\mathsf{proc}(P) \quad \mathsf{proc}([w/y]Q_y)}\ \bullet C$$

8 Lists

With the constructs we have so far, we can now define the type list_A of lists with elements of some arbitrary type A. An "element" here is actually a channel. For example, a list of binary numbers from earlier in the lecture would be list_{bin}, a list of counters would be $\mathsf{list}_{counter}$.

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

If we are using a channel $l : \mathsf{list}_A$, the above type expresses that we either receive a cons label followed by a channel of type A and then another list, or we receive a nil label followed by an end message closing the channel.

As an example, we develop a process that takes two lists, l and k, and produces the result r of appending them. On each line we show the state of the type of the channels in the form $\Omega \vdash (r : C)$ as we fill in the process. We begin by receiving a label from l, just like the functional code would be a case of the structure of l.

```
$\mathsf{list}_A = \oplus\{\mathsf{cons} : A \bullet \mathsf{list}_A, \mathsf{nil} : \mathbf{1}\}$
$(l : \mathsf{list}_A)\ (k : \mathsf{list}_A) \vdash append : (r : \mathsf{list}_A)$
$append = \mathsf{case}\ l\ (\ \mathsf{nil} \Rightarrow \ldots$
                 $\mid \mathsf{cons} \Rightarrow \ldots)$
```

When the input list is empty, the result list r is simply k, which we accomplish just by forwarding. We just have to make sure to wait for the termination of l.

```
$\mathsf{list}_A = \oplus\{\mathsf{cons} : A \bullet \mathsf{list}_A, \mathsf{nil} : \mathbf{1}\}$
$(l : \mathsf{list}_A)\ (k : \mathsf{list}_A) \vdash append : (r : \mathsf{list}_A)$
$append = \mathsf{case}\ l\ (\ \mathsf{nil} \Rightarrow$          % $(l : \mathbf{1})\ (k : \mathsf{list}_A) \vdash (r : \mathsf{list}_A)$
                     $\mathsf{wait}\ l\ ;$                  % $(k : \mathsf{list}_A) \vdash (r : \mathsf{list}_A)$
                     $r \leftarrow k$
                 $\mid \mathsf{cons} \Rightarrow \ldots$             % $(l : A \bullet \mathsf{list}_A)\ (k : \mathsf{list}_A) \vdash (r : \mathsf{list}_A)$
```

In the case for cons, we have exposed the underlying ordered pair $A \bullet \mathsf{list}_A$. Checking against the left rule for $\bullet$

$$\frac{\Omega_L\ (y{:}A)\ (x{:}B)\ \Omega_R \vdash Q_y :: (z{:}C)}{\Omega_L\ (x{:}A \bullet B)\ \Omega_R \vdash (y \leftarrow \mathsf{recv}\ x\ ;\ Q_y) :: (z{:}C)}\ {\bullet}L$$

we see we can receive the element and it will be added to the antecedents in order

```
$\mathsf{list}_A = \oplus\{\mathsf{cons} : A \bullet \mathsf{list}_A, \mathsf{nil} : \mathbf{1}\}$
$(l : \mathsf{list}_A)\ (k : \mathsf{list}_A) \vdash \mathit{append} : (r : \mathsf{list}_A)$
$\mathit{append} = \mathsf{case}\ l$ ( nil $\Rightarrow$ wait $l$ ; $r \leftarrow k$
                 | cons $\Rightarrow$             % $(l : A \bullet \mathsf{list}_A)\ (k : \mathsf{list}_A) \vdash (r : \mathsf{list}_A)$
                      $x \leftarrow$ recv $l$        % $(x : A)\ (l : A \bullet \mathsf{list}_A)\ (k : \mathsf{list}_A) \vdash (r : \mathsf{list}_A)$
                      ...)
```

Next, we should send a cons label and then x along r: this is how much of the output list r we already know at this point. And we have to do this because we cannot recurse before the context has the right form again. Checking the form of the $\bullet R^*$ rule

$$\frac{\Omega \vdash P :: (x{:}B)}{(w{:}A)\ \Omega \vdash (\,\mathsf{send}\ x\ w\ ;\ P) :: (x{:}A \bullet B)}\ \bullet R^*$$

we see that it is possible to send x because it is indeed at the left end of the context.

```
$\mathsf{list}_A = \oplus\{\mathsf{cons} : A \bullet \mathsf{list}_A, \mathsf{nil} : \mathbf{1}\}$
$(l : \mathsf{list}_A)\ (k : \mathsf{list}_A) \vdash \mathit{append} : (r : \mathsf{list}_A)$
$\mathit{append} = \mathsf{case}\ l$ ( nil $\Rightarrow$ wait $l$ ; $r \leftarrow k$
                 | cons $\Rightarrow$             % $(l : A \bullet \mathsf{list}_A)\ (k : \mathsf{list}_A) \vdash (r : \mathsf{list}_A)$
                      $x \leftarrow$ recv $l$        % $(x : A)\ (l : \mathsf{list}_A)\ (k : \mathsf{list}_A) \vdash (r : \mathsf{list}_A)$
                      $r$.cons ;          % $(x : A)\ (l : \mathsf{list}_A)\ (k : \mathsf{list}_A) \vdash (r : A \bullet \mathsf{list}_A)$
                      send $r\ x$ ;       % $(l : \mathsf{list}_A)\ (k : \mathsf{list}_A) \vdash (r : \mathsf{list}_A)$
                      ...)
```

Now we can recurse, completing the program.

```
$\mathsf{list}_A = \oplus\{\mathsf{cons} : A \bullet \mathsf{list}_A, \mathsf{nil} : \mathbf{1}\}$
$(l : \mathsf{list}_A)\ (k : \mathsf{list}_A) \vdash \mathit{append} : (r : \mathsf{list}_A)$
$\mathit{append} = \mathsf{case}\ l$ ( nil $\Rightarrow$ wait $l$ ; $r \leftarrow k$
                 | cons $\Rightarrow$             % $(l : A \bullet \mathsf{list}_A)\ (k : \mathsf{list}_A) \vdash (r : \mathsf{list}_A)$
                      $x \leftarrow$ recv $l$        % $(x : A)\ (l : \mathsf{list}_A)\ (k : \mathsf{list}_A) \vdash (r : \mathsf{list}_A)$
                      $r$.cons ;          % $(x : A)\ (l : \mathsf{list}_A)\ (k : \mathsf{list}_A) \vdash (r : A \bullet \mathsf{list}_A)$
                      send $r\ x$ ;       % $(l : \mathsf{list}_A)\ (k : \mathsf{list}_A) \vdash (r : \mathsf{list}_A)$
                      $\mathit{append}$)
```

Interestingly, it appears[2] all terminating processes P with the type

$$(l : \mathsf{list}_A)\ (k : \mathsf{list}_A) \vdash P : (r : \mathsf{list}_A)$$

[2] a conjecture, at present...

will have to append l and k in this order, even if the details on how this is accomplished may differ. For example, let's imagine we want to read an element from k and send this on to r in a hypothetical process *exapp*:

```
$\mathsf{list}_A = \oplus\{\mathsf{cons} : A \bullet \mathsf{list}_A, \mathsf{nil} : \mathbf{1}\}$
$(l : \mathsf{list}_A)\,(k : \mathsf{list}_A) \vdash \mathit{exapp} : (r : \mathsf{list}_A)$
$\mathit{exapp} = \mathsf{case}\ k$ ( $\mathsf{nil} \Rightarrow \ldots$
                  | $\mathsf{cons} \Rightarrow$                     % $(l : \mathsf{list}_A)\ (k : A \bullet \mathsf{list}_A) \vdash (r : \mathsf{list}_A)$
                        $x \leftarrow \mathsf{recv}\ k$          % $(l : \mathsf{list}_A)\ (x : A)\ (k : \mathsf{list}_A) \vdash (r : \mathsf{list}_A)$
                        $\ldots$)
```

At this point we cannot send x along r because x is not at the left end of the context. So the constraints imposed by ordered logic significantly constrain the space of possibly implementations. If we worked in linear logic instead, where the order of hypotheses didn't matter, sending x here would be possible. The only guarantee we would get[3] is that r contains all the elements from l and k in some arbitrary order.

[3] a conjecture, at present. . .