Lecture Notes on Synchronous Message-Passing

15-317: Constructive Logic
Frank Pfenning

Lecture 25
Tuesday, April 25, 2023

1 Introduction

In the last lecture we introduced intuitionistic linear logic as a way to capture reasoning with resources. We presented the logic in the form of a sequent calculus rather than natural deduction because management of resources (logically, the antecedents) is more explicit and easier to reason about. Instead of local reductions and local expansions, we used cut reductions and identity expansion to test the right and left rules for all connectives for harmony. This directly feeds into the global properties of the admissibility of cut and identity, which we mentioned but did not formally state or prove.

Natural deduction gave us functional computation via the Curry-Howard correspondence. In this lecture we explore that the linear sequent calculus gives us in terms of computation. It turns out that it is *synchronous message-passing*. This observation was made most explicitly in work by Caires and Pfenning [2010] and Caires et al. [2016].

2 Proofs as Processes

In natural deduction, we interpret a derivation as a function from proofs of the hypotheses to a proof of the conclusion. Introduction rules are constructors, elimination rules are destructors, and local reduction (which is defined when a destructor meets a constructor) represents a step of computation.

In linear sequent calculus, we interpret a derivation as a *process*. The antecedents as well as the succedent are labeled with *channels* for the process to communicate along. We say a process *uses* (or *is a client to*) the channels on the left and *provides* the channel on the right.

$$\underbrace{a_1 : A_1, \ldots, a_n : A_n}_{\text{channels used}} \vdash \underbrace{c : C}_{\text{provided}}$$

Processes are connected by *private channels*, one side being the provider the other side the

client. Logically, this corresponds to a cut.

$$\frac{\begin{array}{cc} P & Q \\ \Delta_1 \vdash x : A & \Delta_2, x : A \vdash c : C \end{array}}{\Delta_1, \Delta_2 \vdash c : C}\ \mathsf{cut}$$

Here, x is the name of the private communication channel. To say it another way, cut corresponds to parallel composition of the two proofs with a private communication channel between them. We have labeled the proof P and Q in anticipation of this naming convention for processes.

The fact that cut is now essential in creating a configuration of communicating processes means that in this sequent calculus we take it as a first-class rule rather than one that is admissible.

3 Cut Reduction as Communication

Let's recall from the last lecture one of the two cases of cut reduction for alternative conjunction.

$$\frac{\dfrac{\begin{array}{cc} \mathcal{D}_1 & \mathcal{D}_2 \\ \Delta' \vdash A_1 & \Delta' \vdash A_2 \end{array}}{\Delta' \vdash A_1 \mathbin{\&} A_2}\ \&R \quad \dfrac{\begin{array}{c} \mathcal{E}_1 \\ \Delta_1, A_1 \vdash C \end{array}}{\Delta_1, A_1 \mathbin{\&} A_2 \vdash C}\ \&L_1}{\Delta', \Delta_1 \vdash C}\ \mathsf{cut}_{A_1 \& A_2}$$

$$\longrightarrow_R \quad \frac{\begin{array}{cc} \mathcal{D}_1 & \mathcal{E}_1 \\ \Delta' \vdash A_1 & \Delta_1, A_1 \vdash C \end{array}}{\Delta', \Delta_1 \vdash C}\ \mathsf{cut}_{A_1}$$

We now label the channels and rename the proofs from $\mathcal{D}$ and $\mathcal{E}$ to P and Q.

$$\frac{\dfrac{\begin{array}{cc} P_1 & P_2 \\ \Delta' \vdash x : A_1 & \Delta' \vdash x : A_2 \end{array}}{\Delta' \vdash x : A_1 \mathbin{\&} A_2}\ \&R \quad \dfrac{\begin{array}{c} Q_1 \\ \Delta_1, x : A_1 \vdash C \end{array}}{\Delta_1, x : A_1 \mathbin{\&} A_2 \vdash C}\ \&L_1}{\Delta', \Delta_1 \vdash c : C}\ \mathsf{cut}_{A_1 \& A_2}$$

$$\longrightarrow_R \quad \frac{\begin{array}{cc} P_1 & Q_1 \\ \Delta' \vdash x : A_1 & \Delta_1, x : A_1 \vdash C \end{array}}{\Delta', \Delta_1 \vdash c : C}\ \mathsf{cut}_{A_1}$$

If we think of the first premise of the cut as P and the second Q, how does the information flow here? The process P provides an *external choice* to the client Q. The choice is between A_1 and A_2, which are provided by P_1 and P_2, respectively. The client makes this choice. In particular Q starts with $\&L_1$, which chooses A_1. On the right-hand of the reduction we

see that processes P_1 and Q_1 now communicate along the same channel x, but this channel now has type A_1.

The information that is transmitted here is a single bit, represented, say, by either fst or snd. Our notation for these processes is

$$\begin{array}{lcl} P & = & (\mathbf{recv}\ x\ (\mathsf{fst} \Rightarrow P_1 \mid \mathsf{snd} \Rightarrow P_2)) \\ Q & = & (\mathbf{send}\ x\ \mathsf{fst}\ ;\ Q_1) \end{array}$$

The semicolon in Q the represents a kind of sequencing of actions: first we send fst along channel x and then we continue as Q_1. The process P receives either fst or snd along x and branches on what receives. This is rather like a case statement in ML.

One important difference to keep in mind, though, is that the channel x changes type. When P and Q run in parallel, x has type $A_1 \mathbin{\&} A_2$. After the client makes a choice of fst, x has type A_1. If the client had chosen snd instead, x would have type A_2 afterwards.

In order to formalize the dynamics we use *multiset rewriting* [Cervesato and Scedrov, 2009]. We represent a configuration of communicating processes as multiset of semantic objects $\mathsf{proc}\ P$. A rewriting rule matches the left-hand side of a rule against a subset of the configuration and replaces it by the right-hand side. In this case, the rules are:

$$\begin{array}{lcl} \mathsf{proc}\ (\mathbf{recv}\ a\ (\mathsf{fst} \Rightarrow P_1 \mid \mathsf{snd} \Rightarrow P_2)), \mathsf{proc}\ (\mathbf{send}\ a\ \mathsf{fst}\ ;\ Q_1) & \mapsto & \mathsf{proc}\ P_1, \mathsf{proc}\ Q_1 \\ \mathsf{proc}\ (\mathbf{recv}\ a\ (\mathsf{fst} \Rightarrow P_1 \mid \mathsf{snd} \Rightarrow P_2)), \mathsf{proc}\ (\mathbf{send}\ a\ \mathsf{snd}\ ;\ Q_2) & \mapsto & \mathsf{proc}\ P_2, \mathsf{proc}\ Q_2 \end{array}$$

Communication is *synchronous* in the sense that both sender and recipient step forward to a new process as part of the same transition. To make all communication *asynchronous* we need the linear semi-axiomatic sequent calculus (SAX) [DeYoung et al., 2020] instead of the linear sequent calculus from the last lecture.

We also annotate the sequents directly with the proof terms from above, just as we did for natural deduction. The logical rules then become typing rules. The judgment has the form

$$a_1 : A_1, \ldots, a_n : A_n \vdash P :: (c : C)$$

Here are the rules for external choice.

$$\frac{\Delta \vdash P_1 :: (a : A_1) \quad \Delta \vdash P_2 :: (a : A_2)}{\Delta \vdash \mathbf{recv}\ a\ (\mathsf{fst} \Rightarrow P_1 \mid \mathsf{snd} \Rightarrow P_2) :: (a : A_1 \mathbin{\&} A_2)}\ \mathbin{\&}R$$

$$\frac{\Delta, a : A_1 \vdash Q :: (c : C)}{\Delta, a : A_1 \mathbin{\&} A_2 \vdash \mathbf{send}\ a\ \mathsf{fst}\ ;\ Q :: (c : C)}\ \mathbin{\&}L_1 \qquad \frac{\Delta, a : A_2 \vdash Q :: (c : C)}{\Delta, a : A_1 \mathbin{\&} A_2 \vdash \mathbf{send}\ a\ \mathsf{snd}\ ;\ Q :: (d : C)}\ \mathbin{\&}L_2$$

Before we move on to the other connectives, we consider the cut itself.

4 Cut as Spawn

Cut, when seen as first-class rule, spawns a new process and allocates a fresh channel.

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

where

$$\mathsf{proc}\ (x{:}A \leftarrow P(x)\ ;\ Q(x)) \quad \mapsto \quad \mathsf{proc}\ P(a), \mathsf{proc}\ Q(a) \qquad (a \text{ fresh})$$

We see that $P(a)$ *provides* channel a and $Q(a)$ is a client to a. We follow the convention there that $P(x)$ corresponds to P with occurrences of x and $P(a)$ substituted a for x in $P(x)$.

5 Internal Choice

Internal choice $A \oplus B$ is just like external choice, except that the roles of provider and client are reversed.

$$\frac{\Delta \vdash P_1 :: (a : A_1)}{\Delta \vdash \mathbf{send}\ a\ \mathsf{fst}\ ;\ P_1 :: (a : A_1 \oplus A_2)}\ \oplus R_1 \qquad \frac{\Delta \vdash P_2 :: (a : A_2)}{\Delta \vdash \mathbf{send}\ a\ \mathsf{snd}\ ;\ P_2 :: (a : A_1 \oplus A_2)}\ \oplus R_2$$

$$\frac{\Delta, a : A_1 \vdash Q_1 :: (c : C) \quad \Delta, a : A_2 \vdash Q_2 :: (c : C)}{\Delta, a : A_1 \oplus A_1 \vdash \mathbf{recv}\ a\ (\mathsf{fst} \Rightarrow Q_1 \mid \mathsf{snd} \Rightarrow Q_2) :: (c : C)}\ \oplus L$$

Remarkably, the computation rules are exactly like the ones for external choice!

$$\begin{array}{lcl} \mathsf{proc}\ (\mathbf{send}\ a\ \mathsf{fst}\ ;\ P_1), \mathsf{proc}\ (\mathbf{recv}\ a\ (\mathsf{fst} \Rightarrow Q_1 \mid \mathsf{snd} \Rightarrow Q_2)) & \mapsto & \mathsf{proc}\ P_1, \mathsf{proc}\ Q_1 \\ \mathsf{proc}\ (\mathbf{send}\ a\ \mathsf{snd}\ ;\ P_2), \mathsf{proc}\ (\mathbf{recv}\ a\ (\mathsf{fst} \Rightarrow Q_1 \mid \mathsf{snd} \Rightarrow Q_2)) & \mapsto & \mathsf{proc}\ P_2, \mathsf{proc}\ Q_2 \end{array}$$

We have written them in a different order, and also swapped P and Q as names for the process expressions appearing in the rule.

6 Recursion

As in the step from natural deduction to functional programming, recursion is a bit of a fly in the ointment. In the case of natural deduction, we analyzed induction and the corresponding schema of primitive recursion.

For processes, this approach doesn't quite work because we often think of processes (like servers) as running indefinitely. On a technical level, recursive types are interpreted coinductively instead of inductively. For example, they would represent unbounded streams instead of finite lists. This takes an extensive amount of setup and theory, so instead we just allow arbitrary recursive types and arbitrary recursively defined processes. Many of these do not have a counterpart in logic, so we depart from a strict Curry-Howard isomorphism for the sake of brevity and simplicity.

Consider

$$\mathsf{bits} = \mathsf{bits} \oplus \mathsf{bits}$$

This represents an infinite stream of bits, where, say, the label fst represents the bit 0 and snd represents the bit 1. Such an encoding makes examples difficult to write, we allow general finite labeled sums instead of just the unlabeled binary one. This example above would be rewritten as

$$\mathsf{bits} = (\mathsf{b0} : \mathsf{bits}) \oplus (\mathsf{b1} : \mathsf{bits})$$

where b0 and b1 are labels that are exchanged via message passing. The usual unlabeled same can then be defined by $A \oplus B \triangleq (\mathsf{fst} : A) \oplus (\mathsf{snd} : B)$.

Let's write a transducer that negates every bit in a bit stream.

$\mathsf{bits} = (\mathsf{b0} : \mathsf{bits}) \oplus (\mathsf{b1} : \mathsf{bits})$

$x : \mathsf{bits} \vdash \mathit{neg}(y, x) :: (y : \mathsf{bits})$
$\mathit{neg}(y, x) =$
  $\mathbf{recv}\ x\ (\ \mathsf{b0} \Rightarrow \mathbf{send}\ y\ \mathsf{b1}\ ;\ \mathit{neg}(y, x)$
  $\quad\quad | \ \mathsf{b1} \Rightarrow \mathbf{send}\ y\ \mathsf{b0}\ ;\ \mathit{neg}(y, x)\ )$

Negation *neg* is a recursively defined process with two parameters that stand for channels. We always put the provided channel first in the list of parameters, just like the channel we send comes before the message.

We can compose two of these into a "network" of two processes. From endpoint to endpoint, it just copies bits from the input x to identical bits in the output z (with some internal delay).

$\mathit{neg2}(z, x) =$
  y:bits $\leftarrow \mathit{neg}(y, x)$;
  $\mathit{neg}(z, y)$

7 Unit and Termination

We assign processes to the unit rules as follows:

$$\frac{}{\cdot \vdash \mathbf{send}\ a\ \langle\rangle :: (a : 1)}\ 1R \qquad \frac{\Gamma \vdash Q :: (c : C)}{\Gamma, a : 1 \vdash \mathbf{recv}\ a\ (\langle\rangle \Rightarrow P) :: (c : C)}\ 1L$$

As expected, they take complementary action. What is new is the that process that sends the unit doesn't have a continuation process, which means that it terminates. So the only information communicated between these two is that the sending process has terminated.

$$\mathsf{proc}\ (\mathbf{send}\ a\ \langle\rangle), \mathsf{proc}\ (\mathbf{recv}\ a\ (\langle\rangle \Rightarrow Q)) \quad \mapsto \quad \mathsf{proc}\ Q$$

Now we can change our example: instead of an infinite stream of bits, we consider a potentially finite stream of bits terminated by the label e followed by the unit $\langle\rangle$. We can represent binary numbers in this form with the least significant bit arriving first.

$\mathsf{bin} = (\mathsf{b0} : \mathsf{bin}) \oplus (\mathsf{b1} : \mathsf{bin}) \oplus (\mathsf{e} : 1)$

For example, $6 = (110)_2$ would represented by the following sequences of messages on some channel.

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

We can make the following definitions of zero and successor processes.

$\cdot \vdash \mathit{zero}(z) :: (z : \mathsf{bin})$
$\mathit{zero}(z) =$
  $\mathbf{send}\ z\ e$;
  $\mathbf{send}\ z\ \langle\rangle$

This process will terminate after sending $\mathsf{e} \cdot \langle\rangle$.

$x : \mathsf{bin} \vdash \mathit{succ}(y, x) :: (y : \mathsf{bin})$
$\mathit{succ}(y, x) =$
  $\mathbf{recv}\ x\ (\ \mathsf{b0} \Rightarrow \mathbf{send}\ y\ \mathsf{b1}$;
    $\mathbf{fwd}\ y\ x$
  $\mid \mathsf{b1} \Rightarrow \mathbf{send}\ \mathsf{b0}\ y$;
    $\mathit{succ}(y, x)$
  $\mid \mathsf{e} \Rightarrow \mathbf{send}\ y\ \mathsf{b0}$;
    $\mathbf{send}\ y\ \mathsf{e}$;
    $\mathbf{fwd}\ y\ x\ )$

The new construct $\mathbf{fwd}\ c\ a$ is the proof term for the (general) identity.

$$\frac{}{a : A \vdash \mathbf{fwd}\ c\ a :: (c : C)}\ \mathsf{id}$$

It is called *forwarding* because it forwards any communication attempt on a to c and on c to a. In order to capture this concisely we can refactor our rules slightly, anticipating that for $c : A \multimap B$ and $c : A \otimes B$, the process terms will send a channel of type a along c.

Processes	P	::=	$x{:}A \leftarrow P(x)\ ;\ Q(x)$	cut/spawn
		$\mid$	$\mathbf{fwd}\ c\ a$	id/forward
		$\mid$	$\mathbf{send}\ a\ V\ ;\ \hat{P}$	send V along a, cont. as $\hat{P}$
		$\mid$	$\mathbf{recv}\ a\ K$	receive a V along a and pass it to K
Values	V	::=	k	labels ($\oplus$,&)
		$\mid$	b	channels ($\otimes$, $\multimap$)
		$\mid$	$\langle\rangle$	unit (1)
Continuations	K	::=	$(\ell \Rightarrow P_\ell)_{\ell \in L}$	labels ($\oplus$, &)
		$\mid$	$(y \Rightarrow P(y))$	channels ($\otimes$, $\multimap$)
		$\mid$	$(\langle\rangle \Rightarrow P)$	unit (1)
Optional Process	$\hat{P}$	::=	$\epsilon \mid P$	

$$\begin{array}{lcl}
\mathsf{proc}\ (\mathbf{send}\ c\ V\ ;\ \hat{P}), \mathsf{proc}\ (\mathbf{recv}\ c\ K) & \mapsto & \mathsf{proc}\ \hat{P}, \mathsf{proc}\ (V \rhd K) \\
\mathsf{proc}\ (\epsilon) & \mapsto & \cdot \\
\mathsf{proc}\ (\mathbf{send}\ a\ V\ ;\ \hat{P}), \mathsf{proc}\ (\mathsf{fwd}\ c\ a) & \mapsto & \mathsf{proc}\ (\mathbf{send}\ c\ V\ ;\ \hat{P}) \\
\mathsf{proc}\ (\mathsf{fwd}\ c\ a), \mathsf{proc}\ (\mathbf{send}\ c\ V\ ;\ \hat{P}) & \mapsto & \mathsf{proc}\ (\mathbf{send}\ a\ V\ ;\ \hat{P})
\end{array}$$

$$\begin{array}{rcll}
k & \rhd & (\ell \Rightarrow P_\ell)_{\ell \in L} & = P_k \quad (k \in L) \\
b & \rhd & (y \Rightarrow P(y)) & = P(b) \\
\langle\rangle & \rhd & (\langle\rangle \Rightarrow P) & = P
\end{array}$$

8 Multiplicative Connectives

Finally, we come to the multiplicative connectives $A \multimap B$ and $A \otimes B$. Viewed from the provider, the first receives a channel of type A while the second sends a channel of type A. In this, we slightly change the rules of our sequent calculus. It is certainly not essential (see [Caires and Pfenning, 2010]) but it makes the operational reading simpler.

$$\frac{\Delta, x : A \vdash P(x) :: (b : B)}{\Delta \vdash \mathbf{recv}\ b\ (x \Rightarrow P(x)) :: (b : A \multimap B)}\ \multimap R \qquad \frac{\Delta, b : B \vdash Q :: (c : C)}{\Delta, a : A, b : A \multimap B \vdash \mathbf{send}\ b\ a\ ;\ Q :: (c : C)}\ \multimap L$$

$$\frac{\Delta \vdash P :: (b : B)}{\Delta, a : A \vdash \mathbf{send}\ b\ a\ ;\ P :: (b : A \multimap B)}\ \otimes R \qquad \frac{\Delta, x : A, b : B \vdash P(x) :: (c : C)}{\Delta, b : A \otimes B \vdash \mathbf{recv}\ b\ (x \Rightarrow P(x)) :: (c : C)}\ \otimes L$$

The computation rules for these are already covered in the previous section.

9 An Implementation of a Queue Server

As a synthesizing example we use a queue with elements of some arbitrary type A. The interface is specified by the following type.

$$\begin{aligned}\mathsf{queue}_A = \quad & (\mathsf{enq} : A \multimap \mathsf{queue}_A) \\ \& \ & (\mathsf{deq} : (\mathsf{none} : 1) \oplus (\mathsf{some} : A \otimes \mathsf{queue}_A))\end{aligned}$$

We specify two processes: one for the empty queue, and one for the queue holding an element. For the queue holding an element we assume it also is the client to the remainder of the queue.

$$\cdot \vdash \mathit{empty}(q) :: (q : \mathsf{queue}_A)$$

$$\begin{aligned}&\mathit{empty}(q) = \\ &\quad \mathbf{recv}\ q\ (\ \mathsf{enq} \Rightarrow \mathbf{recv}\ q\ (x \Rightarrow t \leftarrow \mathit{empty}(t)\ ; \\ &\qquad\qquad\qquad\qquad\qquad\quad \mathit{elem}(q, x, t) \\ &\qquad\quad \mid \mathsf{deq} \Rightarrow \mathbf{send}\ q\ \mathbf{none}\ ;\ \mathbf{send}\ q\ \langle\rangle\ )\end{aligned}$$

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

$$\begin{aligned}&\mathit{elem}(q, x, t) = \\ &\quad \mathbf{recv}\ q\ (\ \mathsf{enq} \Rightarrow \mathbf{recv}\ q\ (y \Rightarrow \mathbf{send}\ t\ \mathsf{enq}\ ;\ \mathbf{send}\ t\ y\ ; \\ &\qquad\qquad\qquad\qquad\qquad\quad \mathit{elem}(q, x, t)\ ) \\ &\qquad\quad \mid \mathsf{deq} \Rightarrow \mathbf{send}\ q\ \mathsf{some}\ ;\ \mathbf{send}\ q\ x\ ; \\ &\qquad\qquad\quad \mathbf{fwd}\ q\ t\ )\end{aligned}$$

10 The Rast Language

The language for synchronous message passing we developed here has been implemented in the Rast language [Das and Pfenning, 2020b,c]. It has a number of additional features, specifically work and span analysis for parallel programs.

We did not state here the usual progress and preservation theorems for this language (in the presence of recursion). Corresponding theorems can be found in the literature (see, for example, [Das and Pfenning, 2020a]).

References

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.

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.

Iliano Cervesato and Andre Scedrov. Relating state-based and process-based concurrency through linear logic. *Information and Computation*, 207(10):1044–1077, October 2009.

Ankush Das and Frank Pfenning. Session types with arithmetic refinements. In I. Konnov and L. Kovács, editors, *31st International Conference on Concurrency Theory (CONCUR 2020)*, pages 13:1–13:18, Vienna, Austria, September 2020a. LIPIcs 171.

Ankush Das and Frank Pfenning. Rast: Resource-aware session types with arithmetic refinements. In Z. Ariola, editor, *5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020)*, pages 33:1–33:17. LIPIcs 167, June 2020b. System description.

Ankush Das and Frank Pfenning. Rast, a language for resource aware session types, 2020c. URL `https://bitbucket.org/fpfenning/rast/src/master/rast/`.

Henry DeYoung, Frank Pfenning, and Klaas Pruiksma. Semi-axiomatic sequent calculus. In Z. Ariola, editor, *5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020)*, pages 29:1–29:22, Paris, France, June 2020. LIPIcs 167.