Lecture Notes on
Linear Message Passing I

15-836: Substructural Logics
Frank Pfenning

Lecture 5
September 12, 2023

1 Introduction

In this lecture we start a new section of the course. We have studied proof systems for substructural logics and their properties, such as cut and identity elimination. We have also seen that substructural inference itself can express certain algorithms (e.g., for parsing) at a high level of abstraction. We can summarize this with the slogan "*computation is proof construction*". The final answer is a proof, or sometimes just the information of whether a proof exists or not.

Now we look at a connection where proofs themselves are programs, and computation proceeds by *proof reduction* rather than *proof construction*. The new slogan is "*computation is proof reduction*". This notion of computation inherits many desirable properties from logic and proof theory, but it is certainly not without its own set of challenges and difficulties. We will come back to these challenges in Lecture 7, once we have developed an intuitive understanding of the relationship.

Here is the basic table of correspondences:

Logic	**Programming**
Proposition	Type
Proof	Program
Reduction	Computation

The specifics of the correspondence are dramatically dependent on the following variables (and maybe more):

- **The logic.** Ordered logic is different from linear logic, which is be different from structural logic, and several of these come in intuitionistic as well as classical versions. Other examples are temporal logics, modal logics, epistemic logics, and so on, each with their opportunity for computational meaning.

- **The proof system.** Since proofs are programs, the specifics of each proof system determine the structure of programs. And different proof systems have different notion of reduction, which induce different forms of computation.

Such variations are not trivial, but fundamentally change the way we think about programs and their computation. For example, early work by Curry [1934] essentially assigned computational meaning to axiomatic proofs in Hilbert-style systems. Such computation is in the form of combinatory reduction which can be seen at the root of the APL programming language. Later work by Howard [1969] established a relationship between Gentzen's system of natural deduction [Gentzen, 1935, Prawitz, 1965] and Church's typed λ-calculus [Church, 1940]. Here, computation proceeds by substitution which is at the root of modern functional programming languages.

In today's lecture, we begin to establish a connection between linear logic [Girard, 1987, Girard and Lafont, 1987] presented as a sequent calculus, and message-passing processes. The propositions of linear logic express communication protocols, giving a post-hoc logical justification for *session types* [Honda and Tokoro, 1991, Honda, 1993, Honda et al., 1998]. The connection in this form was first established by Caires and Pfenning [2010] and followed up in various ways (e.g., [Wadler, 2012, Caires et al., 2016]).

But enough of the generalities. Let's get started! Instead of ordered logic which has been mostly our focus so far, we now move to linear logic because of the wider variety of programs it supports.

2 Cut as Process Composition

The first two fundamental ideas are the following:

- Proofs represent processes.
- Cut corresponds to the parallel composition of two processes with a private communication channel connecting them.

In order to see how processes are connected, exactly, we label antecedents as we did in Lecture 4. This removes ambiguity, for example, when we have more than one antecedent with a particular proposition A. In addition, we also label the *succedent* with a channel in order to clarify which of the antecedents in the other premise of a cut it is connected to. Without an explicit proof term, the sequent then would have the form

$$\underbrace{x_1 : A_1, \ldots, x_n : A_n}_{\text{channels used}} \quad \vdash \quad \underbrace{x : A}_{\text{channel provided}}$$

A process provides exactly one channel and may use multiple channels. All the variables x_i and x must be distinct.

We need cut as a primitive rule now because cut reduction induces computation. Without cut, there will be no computation! We call the process P (= the proof of the first premise) the *provider* (or server) and the the process Q (= the proof of the second premise) the *client*.

$$\frac{\begin{array}{cc} P & Q \\ \Delta \vdash (x : A) & \Delta', x : A \vdash (z : C) \end{array}}{\Delta, \Delta' \vdash (z : C)}\ \mathsf{cut}$$

The variable x represents the channel of communication between provider and client. This channel is *private* in the sense that P and Q are the only two endpoints, which is guaranteed by our convention about the uniqueness of variable names.

Because the variables in a sequent represent channels, we often just refer to them as channels, just like we might say "the integer x" when x is a variable standing for an integer.

3 Cut Reduction as Communication

The next question is how cut reduction corresponds to communication. We have seen that there are three different kinds of cut reduction:

1. *principal reductions*, where a right rule for a connective meets a corresponding left rule;
2. *identity reductions*, where one of the premises is an identity rule; and
3. *permuting reductions*, where an inference is applied to an antecedent or succedent not involved in the cut.

It turns out that only the first two are of interest *computationally* while the third represents a form of equality reasoning between processes.

We start with principal reductions, using *internal choice* $A \oplus B$ as a guiding example. There are two—we show the first one, since the second one is entirely symmetric.

$$\frac{\left.\begin{array}{cc} \dfrac{\begin{array}{c} P_1 \\ \Delta \vdash x : A \end{array}}{\Delta \vdash x : A \oplus B}\ \oplus R_1 & \dfrac{\begin{array}{cc} Q_1 & Q_2 \\ \Delta', x : A \vdash z : C & \Delta', x : B \vdash z : C \end{array}}{\Delta', x : A \oplus B \vdash z : C}\ \oplus L \end{array}\right\} = Q}{\Delta, \Delta' \vdash z : C}\ \mathsf{cut}_{A \oplus B}$$

$$\longrightarrow_R \quad \frac{\begin{array}{cc} P_1 & Q_1 \\ \Delta \vdash x : A & \Delta', x : A \vdash z : C \end{array}}{\Delta, \Delta' \vdash z : C}\ \mathsf{cut}_A$$

There are some syntactic details to consider, but the first and most important question is "*What is the flow of information here between the first premise (process P) and the second premise (process Q)?*" We see that Q, with the $\oplus L$ rule is prepared for both eventualities: either A might be true or B might be true. This choice is made by P which ends in either $\oplus R_1$ (A is true) or $\oplus R_2$ (B is true). Therefore, P has to communicate this information to Q.

We say that P either sends π_1 or π_2, and Q is set to receive and branch on either of those two tokens. So we write

$$\begin{array}{lcl} P & = & \mathbf{send}\ x\ \pi_1\ ;\ P_1 \\ Q & = & \mathbf{recv}\ x\ (\pi_1 \Rightarrow Q_1 \mid \pi_2 \Rightarrow Q_2) \end{array}$$

where P could also send π_2. If we write the process P into the judgment in the form

$$\Delta \vdash P :: (x : A)$$

then we get the following three rules

$$\frac{\Delta \vdash P_1 :: (x : A)}{\Delta \vdash \mathbf{send}\ x\ \pi_1\ ;\ P_1 :: (x : A \oplus B)}\ \oplus R_1 \qquad \frac{\Delta \vdash P_2 :: (x : B)}{\Delta \vdash \mathbf{send}\ x\ \pi_2\ ;\ P_2 :: (x : A \oplus B)}\ \oplus R_2$$

$$\frac{\Delta, x : A \vdash Q_1 :: (z : C) \quad \Delta, x : B \vdash Q_2 :: (z : C)}{\Delta, x : A \oplus B \vdash \mathbf{recv}\ x\ (\pi_1 \Rightarrow Q_1 \mid \pi_2 \Rightarrow Q_2) :: (z : C)}\ \oplus L$$

These rules are actually closely related to the rules for proof terms from the last lecture, except that our purpose and therefore notation are quite different. For one, we have labeled the succedent with a variable that represent a communication channel. For another, we have used the terms *send* and *receive* to capture the communication action.

We now prefer to read these rules as *typing rules* for the processes P and Q, but we should keep in mind that erasing all the process information turns them back into the familiar logical rules.

We can now go back to the cut reduction and annotate each sequent with its process term. Writing the cut with x as a private channel as $P \parallel_x Q$, we can read off the reduction on processes from the reduction on proofs.

$$\begin{array}{lcl} (\mathbf{send}\ x\ \pi_1\ ;\ P_1) \parallel_x (\mathbf{recv}\ x\ (\pi_1 \Rightarrow Q_1 \mid \pi_2 \Rightarrow Q_2)) & \longrightarrow_R & P_1 \parallel_x Q_1 \\ (\mathbf{send}\ x\ \pi_2\ ;\ P_2) \parallel_x (\mathbf{recv}\ x\ (\pi_1 \Rightarrow Q_1 \mid \pi_2 \Rightarrow Q_2)) & \longrightarrow_R & P_2 \parallel_x Q_2 \end{array}$$

An interesting observation here is that the type of the channel x *evolves* from $A \oplus B$ to either A or B, depending on whether the message was π_1 or π_2. In most programming languages the type of a variable never changes, but here this seems essential. We also see that the "outside" channels (the ones in the conclusion of the cut) which we wrote as Δ, Δ' and $z : C$ do not change their type. This will be important in Lecture 7 when we investigate the properties of the programming language as distinct from the properties of the proof system.

4 Communication and Polarity

With the example of internal choice $A \oplus B$, we have seen that the type prescribes a *communication protocol*: the provider sends either π_1 or π_2 and the client must be prepared to receive either one. Before we look at other connectives, can we predict whether the provider or the client will have information to send? A key idea is that the rule for an invertible connective does not have any information. After all, the premises can be derived if and only if the conclusion can. On the other hand, noninvertible connectives are noninvertible precisely because applying them requires a choice. The information contained in this choice (like π_1 and π_2) is then communicated to the connected process.

Recall that all *positive* connectives are noninvertible on the right. Therefore, taking the provider's perspective, the right rules for positive connectives will *send* a message, while their left rules will *receive* a message. In linear logic, these are $A \oplus B$, $\mathbf{1}$, and $A \otimes B$. Looking at the unit, we have

$$\frac{}{\cdot \vdash \mathbf{1}}\ \mathbf{1}R \qquad \frac{\Delta \vdash C}{\Delta, \mathbf{1} \vdash C}\ \mathbf{1}L$$

$\mathbf{1}$ is not a right invertible connective, because the rule cannot be applied to a section $\Delta \vdash 1$ unless Δ is empty.

$$\frac{\dfrac{}{\cdot \vdash (x : \mathbf{1})}\ \mathbf{1}R \quad \dfrac{\overset{Q}{\Delta' \vdash C}}{\Delta', x : \mathbf{1} \vdash C}\ \mathbf{1}L}{\Delta' \vdash (z : C)}\ \mathsf{cut} \qquad \longrightarrow_R \qquad \overset{Q}{\Delta' \vdash C}$$

The information conveyed, therefore, is only that the associated process terminates, which is done by sending the unit message $()$. First, the typing rules and then the reduction rule.

$$\frac{}{\cdot \vdash \mathbf{send}\ x\ ()\ ::\ (x : \mathbf{1})}\ \mathbf{1}R \qquad \frac{\Delta \vdash Q :: (z : C)}{\Delta, x : \mathbf{1} \vdash \mathbf{recv}\ x\ (() \Rightarrow Q)\ ::\ (z : C)}\ \mathbf{1}L$$

$$\mathbf{send}\ x\ ()\ \|_x\ \mathbf{recv}\ x\ (() \Rightarrow Q) \quad \longrightarrow_R \quad Q$$

Before we move on to the meaning of $A \otimes B$ and the other linear connectives, we program some small examples that are already expressible with just $A \oplus B$ and 1.

5 An Example: Booleans

A very simple type is that of the Booleans.

$$\mathsf{bool} \quad = \quad \mathbf{1} \oplus \mathbf{1}$$

Perhaps not coincidentally, $1 + 1 = 2$ is also the number of different message sequences that can be communicated on a channel $x : \mathsf{bool}$. Namely:

$$\cdot \vdash \mathit{false} :: (c : \mathsf{bool})$$
$$\cdot \vdash \mathit{true} :: (c : \mathsf{bool})$$

$$\mathit{false} = \mathbf{send}\ c\ \pi_1\ ;\ \mathbf{send}\ c\ (\,)$$
$$\mathit{true} = \mathbf{send}\ c\ \pi_2\ ;\ \mathbf{send}\ c\ (\,)$$

It is possible for a process P with the typing $\cdot \vdash P :: (c : \mathsf{bool})$ to spawn other processes and perform a lot of computation, but ultimately it can only send $\pi_1, (\,)$ or $\pi_2, (\,)$ along c, because that is what the type of the channel enforces. It might also fail to terminate if the language permits recursion, which we come to in the next section.

Before that, let's consider a process that receives a Boolean and passes on the negation.

$$a : \mathsf{bool} \vdash \mathit{neg} :: (c : \mathsf{bool})$$

$$\begin{aligned}\mathit{neg} = \mathbf{recv}\ a\ (\,&\pi_1 \Rightarrow \mathbf{recv}\ a\ (() \Rightarrow \mathbf{send}\ c\ \pi_2\ ;\ \mathbf{send}\ c\ (\,))\\ &\mid \pi_2 \Rightarrow \mathbf{recv}\ a\ (() \Rightarrow \mathbf{send}\ c\ p_1\ ;\ \mathbf{send}\ c\ (\,))\,)\end{aligned}$$

From the purely logical perspective, this is uninteresting because this program represents a proof of

$$\mathbf{1} \oplus \mathbf{1} \vdash \mathbf{1} \oplus \mathbf{1}$$

There should be four cut-free and identity-free proofs of this proposition that represents the four unary Boolean functions.

Even though for the moment we have a perfect correspondence between proofs and programs, there is a shift in perspective. Proofs are primarily thought of as evidence for truth, while programs are primarily thought of as objects that compute. Through the correspondence each view influences the others, and we can see relationships and interpretations that may otherwise be missed or found insignificant.

6 Another Example: Natural Numbers

Natural numbers are a mainstay both in logic and programming languages. In logic, they are studied in Peano Arithmetic, in programming languages they are either primitive (perhaps with a bounded range) or thought of as an inductive type.

We will put off any investigation of induction and inductive types and instead go directly to *recursion*, both at the level of types and at the level of programs. Consider, for example, the type $\mathbf{1} \oplus \mathbf{1} \oplus \mathbf{1}$. This has three possible message sequences, $\mathbf{1} \oplus \mathbf{1} \oplus \mathbf{1} \oplus \mathbf{1}$ has four, and so on. There are infinitely many natural numbers, so the definition would be infinite:

$$\mathsf{nat} = \mathbf{1} \oplus (\mathbf{1} \oplus \ldots)$$

Using recursion, we can express this directly as

$$\mathsf{nat} = \mathbf{1} \oplus \mathsf{nat}$$

The corresponding message sequences can also be recursively defined:

$$\overline{n} = \pi_1\,(\,) \mid \pi_2\,\overline{n}$$

We see that the use of π_1 and π_2 is a bit awkward from the programming perspective, so we generalize the binary sum $A \oplus B$ to $\oplus\{\ell : A_\ell\}_{\ell \in L}$ where ℓ are *labels* (also called *tags*) and L is a *finite* index set. Then the binary sum can be defined with the index set $\{\pi_1, \pi_2\}$, maybe written as $A \oplus B \triangleq \oplus\{\pi_1 : A, \pi_2 : B\}$.

Then we define:

$$\mathsf{nat} = \oplus\{\mathsf{zero} : \mathbf{1}, \mathsf{succ} : \mathsf{nat}\}$$

$$\cdot \vdash \mathit{zero} :: (n : \mathsf{nat})$$
$$\mathit{zero} = \mathbf{send}\ n\ \mathsf{zero}\ ;\ \mathbf{send}\ n\ (\,)$$

In order to define the successor process, it is convenient to consider the computational interpretation of the identity. First, the cut reductions, which show that cut and identity "cancel" each other. The structure of A is irrelevant, because the cut is directly eliminated.

$$\dfrac{\dfrac{\overset{P(x)}{\Delta \vdash x : A} \quad \dfrac{}{x : A \vdash y : A}\,\mathsf{id}}{\Delta \vdash y : A}\,\mathsf{cut} \qquad \longrightarrow_R \qquad \overset{P(y)}{\Delta \vdash y : A}$$

$$\dfrac{\dfrac{}{y : A \vdash x : A}\,\mathsf{id} \quad \overset{Q(x)}{\Delta', x : A \vdash z : C}}{\Delta', y : A \vdash z : C}\,\mathsf{cut} \qquad \longrightarrow_R \qquad \overset{Q(y)}{\Delta', y : A \vdash z : C}$$

We see that cut reduction in these two cases performs a variable substitution or renaming ($P(x)$ becomes $P(y)$ and $Q(x)$ becomes $Q(y)$). This renaming is necessary so that the conclusion before and after the reduction remains the same. Computationally, this is necessary because the channels in the conclusion of the cut are connected to other processes (either clients or providers), and these other processes should be able continue to communicate along the same channels as before.

We refer to this operation as *forwarding* because the identity intuitively forwards any messages on the x and y channels to the other.

$$\dfrac{}{x : A \vdash \mathbf{fwd}\ y\ x :: (y : A)}\,\mathsf{id}$$

The provided channel y here comes first, which may seem unintuitive but is part of a number of coordinated decisions is the design of the MPASS programming

language. The reductions:

$$\begin{array}{lcl} P(x) \parallel_x \mathbf{fwd}\ y\ x & \longrightarrow_R & P(y) \\ \mathbf{fwd}\ x\ y \parallel_x Q(x) & \longrightarrow_R & Q(y) \end{array}$$

To make sure the forwarder is connected to the correct provider $P(x)$ or client $Q(x)$, the channel x must actually occur in these processes. Since communication channels are private and linear, this condition is sufficient to guarantee a correct reduction.

We can now complete the brief example of natural numbers by writing the successor process.

$$\mathsf{nat} = \oplus\{\mathsf{zero} : \mathbf{1}, \mathsf{succ} : \mathsf{nat}\}$$

$$\begin{array}{l} \cdot \vdash \mathit{zero} :: (n : \mathsf{nat}) \\ \mathit{zero} = \mathbf{send}\ n\ \mathsf{zero}\ ;\ \mathbf{send}\ n\ (\,) \end{array}$$

$$\begin{array}{l} m : \mathsf{nat} \vdash \mathit{succ} :: (n : \mathsf{nat}) \\ \mathit{succ} = \mathsf{send}\ n\ \mathsf{succ}\ ;\ \mathbf{fwd}\ n\ m \end{array}$$

In programming language parlance, types like nat are *equirecursive*, which means here that there is no message associated with the unfolding of the recursion. In the context of a language such as ML we would think of the type nat as *inductive* because we would like values of this type to be isomorphic to the usual natural numbers. In a non-strict language such as Haskell the type would instead be interpreted coinductively because we can write a simple program that produces an infinite stream of succ constructors. Similarly, in the context of our message-passing programming language, a recursive program could easily send an infinite stream of succ labels. So types in MPASS are interpreted coinductively. This means that we *disallow* type definition such as $\omega = \omega$: the right-hand side of a type definition must always start with a constructor so it uniquely represents a potentially infinite type (that is, a potentially infinite communication protocol) in a finitary way.

For example, the type

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

represents an infinite stream of bits 0 and 1. It is easy to write a transducer process that negates each bit as it comes in.

7 MPASS Syntax

We introduce the syntax of the MPASS language for the remaining examples of this lecture and the following two lectures. You can download a version of MPASS from the course resources page. This contains a `readme.txt` file with a full grammar

and other useful information. The examples from this lecture can be found in the file lecture5.mps.

We can define types recursively at the top level using the **type** keyword, including mutual recursion. Labels must be preceded by a single quote so they are syntactically distinguished from the names of types, channels, and processes.

```
% unary natural numbers
type nat = +{'zero : 1, 'succ : nat}
```

Processes are defined with the **proc** keyword, followed by the name of the process, then followed by the channel provided by the process and its type. Continuing the example:

```
proc zero (n : nat) = send n 'zero ; send n ()
```

This avoids the need for a separate type declaration. If the process additionally *uses* channels, they follow after the provided one.

```
proc succ (n : nat) (m : nat) = send n 'succ ; fwd n m
```

8 Example: Natural Numbers in Binary Form

We have already seen natural numbers in binary form as an example for ordered inference. Now we think of them in terms of message-passing like the natural numbers, where the least significant bit is sent first.

```
type bin = +{'b0 : bin, 'b1 : bin, 'e : 1}

proc zero (x : bin) = send x 'e ; send x ()
proc succ (x : bin) (y : bin) =
  recv y ( 'b0 => send x 'b1 ; fwd y x
         | 'b1 => send x 'b0 ; call succ x y   % carry
         | 'e => recv y (() => send x 'b1 ; send x 'e ; send x () ) )
```

We also see an example for the **call** keyword. Its first argument is a process (here a recursive call), the second is the provided channel, and the remaining ones are the used channels passed to the process. These arguments must match the type declarations in the process header.

To write and understand such a program it is often extremely valuable to calculate the type of every variable at various program points. That's because they change, and yet determine what may be possible as a next interaction.

The syntax for cut is, abstractly $x_A \leftarrow P(x)\ ;\ Q(x)$, where $P(x)$ *provides* x and $Q(x)$ *uses* x. The typing rule:

$$\frac{\Delta \vdash P(x) :: (x : A) \quad \Delta', x : A \vdash Q(x) :: (z : C)}{\Delta, \Delta' \vdash x_A \leftarrow P(x)\ ;\ Q(x) :: (z : C)}\ \mathsf{cut}$$

The type A indicates the type of the new private channel, which may not be readily inferable. We do not indicate how the antecedents of the conclusion are to be split between Δ and Δ'; instead this is determined by a type-checking algorithm.

Dynamically, cut creates a (globally fresh) channel a, spawns the process $P(a)$ and continues as $Q(a)$. So there is a small asymmetry here inherent in the nature of the (intuitionstic) sequent with a single conclusion.

$$x_A \leftarrow P(x)\ ;\ Q(x) \quad \longrightarrow \quad P(a) \parallel_a Q(a) \quad (a \text{ fresh})$$

Since cut just allocates a fresh channel and spawns a new process, there is no interprocess communication involved in its operational interpretation.

We can use this to test our small successor programs: we create a channel initialized to zero and then increment it several times. We show the current state of the typing judgment after a line of code. For example, the first call to `succ` will pass x_0 to it, so it is no longer in the current context.

```
proc test (x : bin) =
  x0 <- call zero x0 ;      % x0 : bin |- x : bin
  x1 <- call succ x1 x0 ;   % x1 : bin |- x : bin
  x2 <- call succ x2 x1 ;   % x2 : bin |- x : bin
  x3 <- call succ x3 x2 ;   % x3 : bin |- x : bin
  fwd x x3
```

We recognize an idiom here, where we allocate a fresh channel like x_1 and spawn a new named process providing it at the same time. When we use cut this way we can omit the type annotation for the new channel.

Because our language is concurrent, all these successor processes may be lined up in a pipeline, passing bits through. Because the process `test` provides a channel x but does not use any channels, we can execute this program with the **`exec`** keyword.

```
exec test
```

This will print back the channels that have been created and are externally observable, starting with the initial channel (0), and the message observed on each of the channels. Here, there is only one because the other ones are closed when the successor processes terminate.

```
% executing test
(0) -> b1.b1.e.()
```

So the sequence of messages on channel (0) is `b1`, followed by `b1`, followed by `e` and $(\,)$ which closes this channel.

9 Summary

We have introduced a message-passing interpretation of sequent calculus proofs in linear logic and given it a syntax. We will summarize the statics (typing rules) and

dynamics (computation rules) after the next lecture. So far, we have only consider internal choice and unit, but due to the presence of recursion we were already able to write some small but nontrivial programs.

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.

Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

H. B. Curry. Functionality in combinatory logic. *Proceedings of the National Academy of Sciences, U.S.A.*, 20:584–590, 1934.

Gerhard Gentzen. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1935. English translation in M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131, North-Holland, 1969.

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

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.

Kohei Honda. Types for dyadic interaction. In E. Best, editor, *4th International Conference on Concurrency Theory (CONCUR 1993)*, pages 509–523. Springer LNCS 715, 1993.

Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In P. America, editor, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'91)*, pages 133–147, Geneva, Switzerland, July 1991. Springer-Verlag LNCS 512.

Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In C. Hankin, editor, *7th European Symposium on Programming Languages and Systems (ESOP 1998)*, pages 122–138. Springer LNCS 1381, 1998.

W. A. Howard. The formulae-as-types notion of construction. Unpublished note. An annotated version appeared in: *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, 479–490, Academic Press (1980), 1969.

Dag Prawitz. *Natural Deduction*. Almquist & Wiksell, Stockholm, 1965.

Philip Wadler. Propositions as sessions. In *Proceedings of the 17th International Conference on Functional Programming (ICFP 2012)*, pages 273–286, Copenhagen, Denmark, September 2012. ACM Press.