Lecture Notes on
From Subsingleton to Ordered Logic

15-816: Substructural Logics
Frank Pfenning

Lecture 7
September 20, 2016

In this lecture we first discuss[1] an *asynchronous* model of communication, where messages are always sent and buffered, even if the recipient is not yet ready to receive. Asynchronous communication increases parallelism in a concurrent program, because it let's the sender continue earlier. It is also more realistic with respect to actual implementation models. We will see the *logical origins* of asynchronous communication [DCPT12], and later that synchronous session-typed communication is actually a special case of asynchronous communication [PG15], which is decidedly not the case in the untyped setting of the π-calculus [Pal03].

Then we generalize our operational interpretation of proofs as processes from the subsingleton fragments to all of ordered logic (to the extent we have introduced it so far).

1 Asynchronous Communication

So far, communication has been *synchronous*: the matched processes that are sending and receiving a message continue with their remaining programs together. *Asynchronous communication* in our case will mean that the sending process will not have to wait for the receiving process, but can continue with the remainder of its computation right away. In contrast, attempting to receive a message will *block* until a message arrives. Asynchronous communication requires a *message buffer*, which turns out to be naturally

[1]Actually, we first discussed some homework solutions, which I will omit here to the delight of future generations of students.

representable in our operational framework. Perhaps more suprisingly, it also has a clear logical interpretation [DCPT12].

As an example, let's take a look at internal choice. Its operational semantics is specified by the following ordered rule of inference.

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

We now decompose this into two rules, one ($\oplus C_s$) to send a message and one ($\oplus C_r$) to receive a message. In order to express this, we use a new proposition $\mathsf{msg}(m)$.

$$\frac{\mathsf{proc}(\mathsf{R}.l_k\ ;\ P)}{\mathsf{proc}(P) \quad \mathsf{msg}(\mathsf{R}.l_k)}\ \oplus C_s \qquad\qquad \frac{\mathsf{msg}(\mathsf{R}.l_k) \quad \mathsf{proc}(\mathsf{caseL}(l_i \Rightarrow Q_i)_{i\in I})}{\mathsf{proc}(Q_k)}\ \oplus C_r$$

Note that if a process sends multiple messages, the fact that we have an ordered context will neatly preserve their relative order. In this manner, the messages taken together form a buffer between the two processes. For example:

$$\dfrac{\dfrac{\mathsf{proc}(\mathsf{R}.l_k\ ;\ \mathsf{R}.l_j\ ;\ P) \quad \mathsf{proc}(Q)}{\mathsf{proc}(\mathsf{R}.l_j\ ;\ P) \quad \mathsf{msg}(\mathsf{R}.l_k) \quad \mathsf{proc}(Q)}\ \oplus C_s}{\mathsf{proc}(P) \quad \mathsf{msg}(\mathsf{R}.l_j) \quad \mathsf{msg}(\mathsf{R}.l_k) \quad \mathsf{proc}(Q)}\ \oplus C_s$$

2 Typing Messages

What should the types of messages be? We have collected enough invariants on the computation state in the last lecture in order to derive what needs to happen here. Let's look at the sending rule first, where we have written the interface types on the right below.

$$\frac{\mathsf{proc}(\mathsf{R}.l_k\ ;\ P)}{\mathsf{proc}(P) \quad \mathsf{msg}(\mathsf{R}.l_k)}\ \oplus C_s \qquad\qquad \frac{\omega \vdash (\mathsf{R}.l_k\ ;\ P) : \oplus\{l_i : A_i\}_{i\in I}}{\omega \vdash P : A_k \qquad ??}$$

It is clear from the interface type that we must have

$$A_k \vdash \mathsf{msg}(\mathsf{R}.l_k) : \oplus\{l_i : A_i\}_{i\in I}$$

because it matches the type of P to its left and the type of the interface to the right.

Now comes the central insight of asynchronous communication:

$$\mathsf{msg}(\mathsf{R}.l_k) \simeq \mathsf{proc}(\mathsf{R}.l_k \ ; \leftrightarrow)$$

From the typing perspective, this is easily verified:

$$\dfrac{\dfrac{}{A_k \vdash \leftrightarrow : A_k}\ \mathsf{id}}{A_k \vdash (\mathsf{R}.l_k \ ; \leftrightarrow) : \oplus\{l_i : A_i\}_{i\in I}}\ \oplus R_k$$

We can also see that message receipt

$$\dfrac{\mathsf{msg}(\mathsf{R}.l_k) \quad \mathsf{proc}(\mathsf{caseL}(l_i \Rightarrow Q_i)_{i\in I})}{\mathsf{proc}(Q_k)}\ \oplus C_r$$

works like synchronous communication if we replace the message by a process:

$$\dfrac{\dfrac{\mathsf{proc}(\mathsf{R}.l_k \ ; \leftrightarrow) \quad \mathsf{proc}(\mathsf{caseL}(l_i \Rightarrow Q_i)_{i\in I})}{\mathsf{proc}(\leftrightarrow) \quad \mathsf{proc}(Q_k)}\ \oplus C}{\mathsf{proc}(Q_k)}\ \mathsf{fwd}$$

On the other hand, we really need to treat a message differently from a process, because the *process* $\mathsf{proc}(\mathsf{R}.l_k \ ; \leftrightarrow)$ would immediately spawn another message qua process, which would spawn another message, and so on. So in the operational semantics we use $\mathsf{proc}(P)$ and $\mathsf{msg}(m)$, where our underlying intuition for the meaning of a message is

$$\mathsf{msg}(m) \simeq \mathsf{proc}(m \ ; \leftrightarrow)$$

Rewriting the remaining rules via messages is now straightforward, giving us a complete asynchronous operational semantics. The rules for forwarding, composition, and definitions do not change since they do not involve communication.

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

$$\frac{\mathsf{proc}(\mathsf{R}.l_k\ ;\ P)}{\mathsf{proc}(P) \quad \mathsf{msg}(\mathsf{R}.l_k)}\ \oplus C_s \qquad \frac{\mathsf{msg}(\mathsf{R}.l_k) \quad \mathsf{proc}(\mathsf{caseL}(l_i \Rightarrow Q_i)_{i\in I})}{\mathsf{proc}(Q_k)}\ \oplus C_r$$

$$\frac{\mathsf{proc}(\mathsf{L}.l_k\ ;\ Q)}{\mathsf{msg}(\mathsf{L}.l_k) \quad \mathsf{proc}(Q)}\ \& C_s \qquad \frac{\mathsf{proc}(\mathsf{caseR}(l_i \Rightarrow P_i)_{i\in I}) \quad \mathsf{msg}(\mathsf{L}.l_k)}{\mathsf{proc}(P_k)}\ \& C_r$$

$$\frac{\mathsf{proc}(\mathsf{closeR})}{\mathsf{msg}(\mathsf{closeR})}\ \mathbf{1}C_s \qquad \frac{\mathsf{msg}(\mathsf{closeR}) \quad \mathsf{proc}(\mathsf{waitL}\ ;\ Q)}{\mathsf{proc}(Q)}\ \mathbf{1}C_r$$

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

We see that closeR is somewhat of a special case since it requires the antecedents to be empty, so we cannot forward and $\mathsf{msg}(\mathsf{closeR}) \simeq \mathsf{proc}(\mathsf{closeR})$.

3 Asynchronous Communication as Commuting Cut Reduction

The intuition

$$\mathsf{msg}(m) \simeq \mathsf{proc}(m\ ;\ \leftrightarrow)$$

is the key providing a proof-theoretic understanding of asynchronous sending of messages. Consider once again

$$\frac{\mathsf{proc}(\mathsf{R}.l_k\ ;\ P)}{\mathsf{proc}(P) \quad \mathsf{msg}(\mathsf{R}.l_k)}\ \oplus C_s \qquad \frac{\mathsf{proc}(\mathsf{R}.l_k\ ;\ P)}{\mathsf{proc}(P) \quad \mathsf{proc}(\mathsf{R}.l_k\ ;\ \leftrightarrow)}$$

where the second form uses our definition. But the second form produces two processes, which is the exact behavior of a cut:

$$\frac{\mathsf{proc}(\mathsf{R}.l_k\ ;\ P)}{\mathsf{proc}(P) \quad \mathsf{proc}(\mathsf{R}.l_k\ ;\ \leftrightarrow)} \qquad \frac{\mathsf{proc}(P \mid (\mathsf{R}.l_k\ ;\ \leftrightarrow))}{\mathsf{proc}(P) \quad \mathsf{proc}(\mathsf{R}.l_k\ ;\ \leftrightarrow)}\ \mathsf{cmp}$$

But what is the relationship between

$$(\mathsf{R}.l_k\ ;\ P) \quad \text{and} \quad (P \mid (\mathsf{R}.l_k\ ;\ \leftrightarrow)) \quad ?$$

Writing out the proofs:

$$\dfrac{\omega \vdash P : A_k}{\omega \vdash \mathsf{R}.l_k \ ; \ P : \oplus\{l_i : A_i\}_{i \in I}} \oplus R_k \qquad \dfrac{\omega \vdash P : A_k \quad \dfrac{\dfrac{}{A_k \vdash \leftrightarrow : A_k}\ \mathsf{id}}{A_k \vdash \mathsf{R}.l_k \ ; \leftrightarrow : \oplus\{l_i : A_i\}_{i \in I}} \oplus R_k}{\omega \vdash P \mid (\mathsf{R}.l_k \ ; \leftrightarrow) : \oplus\{l_i : A_i\}_{i \in I}}\ \mathsf{cut}$$

we see that eliminating the cut in the second proof above will lead to the first one: we push the cut up but the $\oplus R_k$ rule with a commuting cut reduction and then eliminate the resulting cut with the identity.

This means from left to right we *introduce* a cut that can be eliminated by a commuting conversion. Introducing a cut will lead to more processes, but at the same time it will also lead to more parallelism because processes can execute independently.

This also tells us that if we stick with synchronous communication we can easily rewrite our programs to behave asynchronously (at least in this case) simply by rewriting $(\mathsf{R}.l_k \ ; \ P)$ as $(P \mid (\mathsf{R}.l_k \ ; \leftrightarrow))$. In other words, in the synchronous calculus we already had the expressive power of asynchronous communication simply by introducing a pair of cut (= spawn) and identity (= forwarding). For the special case of closeR this analysis is not needed, since closeR has no continuation and we cannot logically distinguish between proc(closeR) and msg(closeR).

From a programming point of view, however, it is much more convenient to stick with the standard construction $\mathsf{R}.l_k \ ; \ P$ and interpret all such sends as asynchronous. Interestingly, we show later (or you can read in [PG15]) that we can also go the other way: if we assume all communication is asynchronous we can also recover synchronous communication based on logical principles.

4 From Subsingleton to Ordered Logic

A big next step in this course is investigate how our ideas so far generalize from subsingleton 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 $(A \ / \ B)$, 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

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

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 (see Exercise 1). Instead, we uniquely label each antecedent as well as the succedent[2] with a *channel name*.

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

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.

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.

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

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 (see Exercise 2). 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 (see Lecture 8 for the rules).

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

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

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

Over. As the final connective in this lecture we consider $A \,/\, B$. First, we review the purely logical rules.

$$\frac{\Omega\ B \vdash A}{\Omega \vdash A \,/\, B}\ /R \qquad \frac{\Omega' \vdash B \quad \Omega_L\ A\ \Omega_R \vdash C}{\Omega_L\ (A \,/\, B)\ \Omega'\ \Omega_R \vdash C}\ /L$$

It turns out that the fact the $/L$ rule has two premises complicates the operational reading, so we use the simplified version $/L^*$. With this rule, cut elimination no longer holds (see Exercise L4.6), but cut reduction and identity expansion still do. Since in the setting of proofs as processes we are less concerned about full cut elimination, this is acceptable here. Eventually, we can arrange that all rules except cut have at most one premise, which means that only cut spawns new processes. Recall also that with $/L^*$ and cut, $/L$ is derivable. We return to this issue in Lecture 9.

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

Which of these sends and which receives? In general, there is information in the rule which cannot always be applied (and therefore is not *invertible*), which in this case is $/L^*$. Therefore the process assigned to this rule sends, while the $/R$ receives. We can mechanically fill in channel names and notice that the channel x in $/R$ transitions from type $A \,/\, B$ to type A, so the same transition has to take place in $/L^*$.

$$\frac{\Omega\ (y{:}B) \vdash P_y :: (x{:}A)}{\Omega \vdash ? :: (x{:}A \,/\, B)}\ /R \qquad \frac{\Omega_L\ (x{:}A)\ \Omega_R \vdash Q :: (z{:}C)}{\Omega_L\ (x{:}A \,/\, B)\ (w{:}B)\ \Omega_R \vdash ? :: (z{:}C)}\ /L^*$$

Staring at this rule for a while, we can see *which* information must be transmitted. We reveal the answer on the next page, but you should try to find the answer first.

It is the channel w! Essentially, it is first owned (that is, used) by the process executing the left rule and afterwards it is owned by the process executing the right rule.

$$\frac{\Omega\ (y{:}B) \vdash P_y :: (x{:}A)}{\Omega \vdash (y \leftarrow \mathsf{recv}\ x\ ;\ P_y) :: (x{:}A\ /\ B)}\ /R \qquad \frac{\Omega_L\ (x{:}A)\ \Omega_R \vdash Q :: (z{:}C)}{\Omega_L\ (x{:}A\ /\ B)\ (w{:}B)\ \Omega_R \vdash (\mathsf{send}\ x\ w\ ;\ Q) :: (z{:}C)}\ /L^*$$

The following computation rule implements the cut reduction of $/R$ and $/L^*$.

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

We see that we substitute the received channel w for the bound channel name y in P_y. We will usually write $[w/y]P_y$ as P_w. Since w cannot appear in Q but will appear in P_w, this amounts to an *ownership transfer* for the channel w from one process to another.

In the next lecture we will complete the logical connectives of ordered logic and their operational reading and then summarize them.

Exercises

Exercise 1 Explore whether or not we obtain a logic (and whether this logic has a reasonable operational interpretation) if we restrict the ordered judgment $\Omega\ A \vdash P : B$ so that P can only communicate with the process providing A on the left and B on the right.

Exercise 2 We gave up on the ordered operational semantics because a process needs to communicate with another process that is not an immediate neighbor. Specify an *asynchronous* operational semantics that proceeds via ordered inference and let's messages flow through the configuration. Assess positives and negatives of this semantics.

References

[DCPT12] Henry DeYoung, Luís Caires, Frank Pfenning, and Bernardo Toninho. Cut reduction in linear logic as asynchronous session-typed communication. In P. Cégielski and A. Durand, editors, *Proceedings of the 21st Conference on Computer Science Logic*, CSL 2012, pages 228–242, Fontainebleau, France, September 2012. Leibniz International Proceedings in Informatics.

[Pal03] Catuscia Palamidessi. Comparing the expressive power of the synchronous and the asynchronous π-calculus. *Mathematical Structures in Computer Science*, 13(5):685–719, 2003.

[PG15] Frank Pfenning and Dennis Griffith. Polarized substructural session types. In A. Pitts, editor, *Proceedings of the 18th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS 2015)*, pages 3–22, London, England, April 2015. Springer LNCS 9034. Invited talk.