Lecture Notes on Asynchronous Communication

15-816: Substructural Logics
Frank Pfenning

Lecture 13
October 11, 2016

In this lecture we will first update earlier observations regarding asynchronous communication and then develop an alternative logical integration of synchronous and asynchronous communication. The material is adapted from [DCPT12] and [PG15].

1 Asynchronous from Synchronous Communication

By *asynchronous communication* we mean this in the sense of the π-calculus: we send messages along channels and continue execution without waiting for the message to be received. We do expect, however, that the messages arrive in the order they were sent. When modeling asynchronous communication we need to ensure these two properties, which is usually achieved with a *message buffer* for each channel.

Of course, we want to accomplish all of this by logical means, within the scope of the interpretation of propositions as types, proofs as programs, and computation as cut reduction.

As an example, we consider communication of labels to the right, which is the operational interpretation of internal choice ($\oplus$). Operationally, the key observation was:

$$\mathsf{R}.l_k \; ; \; P \quad \simeq \quad P \mid (\mathsf{R}.l_k \; ; \leftrightarrow)$$

Instead of *synchronously* sending label l_k to the right and then continue with P when it is received, we immediately spawn P and also a (tiny) process that only carries l_k and terminates when that is received.

The fact that the second form is well-typed when the first one is means we can already express asynchronous communication. We just need to decide which of the two programs above to write.

From a logical perspective, the asynchronous form below on the right reduces to the synchronous form on the left by a permuting reduction followed by an identity reduction.

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

Does this generalize to the setting of ordered and linear logic? Yes, as we will illustrate using an ordered example, but it also works for the linear connectives. This time, we start with the logical rules and deduce what the right asynchronous form must be in analogy with the subsingleton case. We use $\bullet R^*$ as our illustrative example.

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

We want to "unshackle" P so it can execute independently from sending of w along x. This is accomplished by a cut. Filling in what we know, we have the following partial deduction:

$$\frac{\Omega \vdash P :: (x : B) \quad (w{:}A)\; (x{:}B) \vdash \ldots :: (x : A \bullet B)}{(w{:}A)\; \Omega \vdash \ldots :: (x : A \bullet B)}\,\mathsf{cut}$$

We see that there is a name clash, so we rename the x introduced by the cut to x'.

$$\frac{\Omega \vdash P :: (x' : B) \quad (w{:}A)\; (x'{:}B) \vdash \ldots :: (x : A \bullet B)}{(w{:}A)\; \Omega \vdash \ldots :: (x : A \bullet B)}\,\mathsf{cut}$$

Luckily, we can directly prove $A\; B \vdash A \bullet B$, even using our weakened rule

$\bullet R^*$. Filling that in, we obtain

$$\frac{\Omega \vdash [x'/x]P :: (x' : B) \quad \dfrac{\dfrac{}{x'{:}B \vdash x \leftarrow x' :: (x : B)}\,\mathsf{id}}{(w{:}A)\ (x'{:}B) \vdash \mathsf{send}\ x\ w\ ;\ x \leftarrow x' :: (x : A \bullet B)}\,\bullet R^*}{(w{:}A)\ \Omega \vdash (x' \leftarrow [x'/x]P\ ;\ \mathsf{send}\ x\ w\ ;\ x \leftarrow x') :: (x : A \bullet B)}\,\mathsf{cut}$$

Operationally, this corresponds to the transition

$$\frac{\mathsf{proc}(x, x' \leftarrow [x'/x]P\ ;\ \mathsf{send}\ x\ w\ ;\ x \leftarrow x')}{\mathsf{proc}(x', [x'/x]P) \quad \mathsf{proc}(\mathsf{send}\ x\ w\ ;\ x \leftarrow x')}\,\mathsf{cmp}^{x'}$$

where x' must be fresh, which represent the derived asynchronous behavior under the correspondence

$$\mathsf{send}\ x\ w\ ;\ P \quad \simeq \quad x' \leftarrow [x'/x]P\ ;\ \mathsf{send}\ x\ w\ ;\ x \leftarrow x'$$

Similar correspondences hold for all other sending constructs (see Exercise 2).

2 An Asynchronous Semantics

As the previous section illustrates, we can implement asynchronous communication given a language with a synchronous semantics. There are two drawbacks of this solution. One is purely a matter of experience: the asynchronous form is tedious to write, since it involves an additional cut and identity. The other is that in realistic settings, synchronous communication is too complex as primitive, while asynchronous communication provides a much more natural model. Synchronous communication is then usually achieved by a protocol of specific asynchronous exchanges.

If we want communication to be a priori asynchronous, all we have to do is to employ the insights from the previous section to devise asynchronous computation rules for the usual sending constructs. In other words, the construct on the left should behave like the construct to the construct to the right.

$$\mathsf{send}\ x\ w\ ;\ P \quad \simeq \quad x' \leftarrow [x'/x]P\ ;\ \mathsf{send}\ x\ w\ ;\ x \leftarrow x'$$

We accomplish this through a new predicate msg containing only certain specialized forms. The sending rule for $\bullet R^*$ becomes

$$\frac{\mathsf{proc}(x, \mathsf{send}\ x\ w\ ;\ P)}{\mathsf{proc}(x', [x'/x]P) \quad \mathsf{msg}(x, \mathsf{send}\ x\ w\ ;\ x \leftarrow x')}\,\bullet C\_\mathsf{send}^{x'}$$

When typing the configuration we type the message just as we would a process. The only reason to single it out with a separate predicate is to prevent immediate application of the same rule again, and again, etc. Messages are received in a way that is consistent with their interpretation as a (tiny) process, shown below for purpose of illustration.

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

We previously wrote the semantics of forwarding as a *global* substitution of x' for x, but here we know the client for x is the process Q_w offering along z. By linearity of channels in well-typed configurations, substitution Q_w is all that is required.[1] In fact, there is an alternative semantics for forwarding that takes this into account (see Exercise 3).

We summarize the above two steps from the synchronous semantics, using only processes, as the one-step transition in the asynchronous semantics using messages.

$$\dfrac{\mathsf{msg}(x, \mathsf{send}\ x\ w\ ;\ x \leftarrow x') \quad \mathsf{proc}(z, y \leftarrow \mathsf{recv}\ x\ ;\ Q_y)}{\mathsf{proc}(z, [x'/x]Q_w)}\ \bullet C\_\mathsf{recv}$$

3 Polarizing Propositions

In the previous section we have developed a semantics were all communication is asynchronous. Is this expressive enough to model synchronous communication if that is what we want in some places? Speaking purely operationally, the standard solution is to send the intended message and then wait for an acknowledgment of receipt before continuing. We *could* introduce this kind of operational solution here in an ad hoc way, but it would be much more satisfactory if it had a good logical justification.

And, yes, there is such a justification or I probably wouldn't have brought it up.

Surprisingly, we only need to generalize slightly the mechanism of modes and shifts in order to find a logical foundation for synchronous communication in an asynchronous language.

[1] Note, however, if we do not perform the two steps together then process z could send x do some other process and the second step would be wrong.

The first key idea is to classify propositions based on whether they send or receive when viewed from the provider perspective. This is the same as asking which right rules for the connectives are invertible in the sense that they can always be applied during the search for a proof of $\Omega \vdash A$, where A is constructed by that connective. This is also the same as saying that the proof if identity expansion has the right rule as its final inference. We call those with invertible right rules *negative* connectives, while those with non-invertible right rules are *positive* connectives.

$$\begin{array}{lrcl} \text{Negative} & A^-, B^- & ::= & p^- \mid A^+ \setminus B^- \mid B^- \,/\, A^+ \mid A^- \,\&\, B^- \\ \text{Positive} & A^+, B^+ & ::= & p^+ \mid A^+ \bullet B^+ \mid A^+ \circ B^+ \mid \mathbf{1} \mid A^+ \oplus B^+ \end{array}$$

Propositional variables p can be given either polarity. Note that in the case of implicational connectives the polarity of the antecedent is positive, since the role of polarities is reversed for antecedents. At this point, just as in the case of combining logics, we need a way to go back and forth between these classes of propositions. For this we reuse the idea behind shifts, except of going between ordered and linear we remain in the ordered layer, changing only the polarity. The directions of the shifts is determined by the fact that the $\uparrow R$ rule is invertible (and therefore $\uparrow^{\mathsf{o}}_{\mathsf{o}} A$ should be negative) and conversely that the $\downarrow R$ rules is not invertible (and therefore $\downarrow^{\mathsf{o}}_{\mathsf{o}} A$ should be positive).

$$\begin{array}{lrcl} \text{Negative} & A^-, B^- & ::= & p^- \mid A^+ \setminus B^- \mid B^- \,/\, A^+ \mid A^- \,\&\, B^- \mid \uparrow^{\mathsf{o}}_{\mathsf{o}} A^+ \\ \text{Positive} & A^+, B^+ & ::= & p^+ \mid A^+ \bullet B^+ \mid A^+ \circ B^+ \mid \mathbf{1} \mid A^+ \oplus B^+ \mid \downarrow^{\mathsf{o}}_{\mathsf{o}} A^- \end{array}$$

The fact that negative propositions are "above" the positive propositions here does not imply an independence principle: the *mode* of both layers is the same (O, in this case) and therefore a proof of a positive succedent can depend on negative antecedents and vice versa. The rules for these constructs are boring. They will become more interesting when we discuss *focusing* where the shifts will play a critical role. In the purely ordered setting (so all the propositions are ordered) we just have:

$$\frac{\Omega \vdash A^+}{\Omega \vdash \uparrow^{\mathsf{o}}_{\mathsf{o}} A^+}\uparrow R \qquad \frac{\Omega_1\ A^+\ \Omega_2 \vdash C}{\Omega_1\ (\uparrow^{\mathsf{o}}_{\mathsf{o}} A^+)\ \Omega_2 \vdash C}\uparrow L$$

$$\frac{\Omega \vdash A^-}{\Omega \vdash \downarrow^{\mathsf{o}}_{\mathsf{o}} A^-}\downarrow R \qquad \frac{\Omega_1\ A^-\ \Omega_2 \vdash C}{\Omega_1\ (\downarrow^{\mathsf{o}}_{\mathsf{o}} A^-)\ \Omega_2 \vdash C}\downarrow L$$

The operational semantics is the same as for the shifts from the previous lecture. $\uparrow^{\mathsf{o}}_{\mathsf{o}} A^+$ is negative and therefore receives a shift. Conversely $\downarrow^{\mathsf{o}}_{\mathsf{o}} A^-$ is positive and therefore sends a shift.

An interesting aspect of the operational semantics is the overall polarization: when a process offers along a channel $x : A^+$ it will send a message, and then continue to send messages until we reach $x : \mathbf{1}$ and we terminate, or we reach $x : \downarrow^{\mathsf{O}}_{\mathsf{O}} B^-$. In the second case, the process now sends a last message—a shift—and then starts to receive messages according to the type B^-. Again, the process will continuously receive since the type remains negative, until we reach a form $x : \uparrow^{\mathsf{O}}_{\mathsf{O}} C^+$ and the cycle begins anew.

4 Example: Polarizing Stacks

As an example, we reconsider stacks.

$$\begin{array}{l}\mathsf{stack}_A = \&\{\ \mathsf{ins} : A \setminus \mathsf{stack}_A, \\ \qquad\qquad \mathsf{del} : \oplus\{\ \mathsf{none} : \mathbf{1}, \\ \qquad\qquad\qquad\quad \mathsf{some} : A \bullet \mathsf{stack}_A\}\}\end{array}$$

The stack interface starts with an external choice, so it is negative which we indicate by stack^-_A. When we go through an insertion and recurse, the type remains negative except we see that A itself occurs *under* a stack ($A \setminus \mathsf{stack}_A$) and therefore should be positive. Filling in this partial information, we have so far:

$$\begin{array}{l}\mathsf{stack}^-_{A^+} = \&\{\ \mathsf{ins} : A^+ \setminus \mathsf{stack}^-_{A^+}, \\ \qquad\qquad \mathsf{del} : \oplus\{\ \mathsf{none} : \mathbf{1}, \\ \qquad\qquad\qquad\quad \mathsf{some} : A^+ \bullet \mathsf{stack}^-_{A^+}\}\}\end{array}$$

At this point we notice some mismatches. In particular, internal choice is a positive connective, but when we enter the del branch of the external choice we expect a negative proposition. So we need to insert a shift and switch to positive polarity. Conversely, when we reach the recursive appeal to the stack type, we expect a positive polarity while stack is negative so we need another shift. We omit the superscript and subscripts on the shifts, since they always stay at level O.

$$\begin{array}{l}\mathsf{stack}^-_{A^+} = \&\{\ \mathsf{ins} : A^+ \setminus \mathsf{stack}^-_{A^+}, \\ \qquad\qquad \mathsf{del} : \uparrow \oplus\{\ \mathsf{none} : \mathbf{1}, \\ \qquad\qquad\qquad\quad \mathsf{some} : A^+ \bullet \downarrow \mathsf{stack}^-_{A^+}\}\}\end{array}$$

We obtained this type by adding the minimal number of shifts to polarize its unpolarized form. Such a minimal translation always exists and is easy to describe.

Let's talk through some sequences of interactions. As long as we insert elements into the stack, we encounter no shift and the direction of the communication remains the same. We might send, for example, the following sequence of six messages:

$$\text{to provider} \quad \mathsf{ins}\ x_1\ \mathsf{ins}\ x_2\ \mathsf{ins}\ x_3$$

Actually, let's see what this looks like in the explicit notation of messages, assuming we start sending along channel y_1.

$$\begin{array}{l}
\mathsf{msg}(y_1, y_1.\mathsf{ins}\ ;\ y_1 \leftarrow y_2) \\
\mathsf{msg}(y_2, \mathsf{send}\ y_2\ x_1\ ;\ y_2 \leftarrow y_3) \\
\mathsf{msg}(y_3, y_3.\mathsf{ins}\ ;\ y_3 \leftarrow y_4) \\
\mathsf{msg}(y_4, \mathsf{send}\ y_4\ x_2\ ;\ y_4 \leftarrow y_5) \\
\mathsf{msg}(y_5, y_5.\mathsf{ins}\ ;\ y_5 \leftarrow y_6) \\
\mathsf{msg}(y_6, \mathsf{send}\ y_6\ x_3\ ;\ y_6 \leftarrow y_7)
\end{array}$$

We see that the messages form a *linked list segment* from y_1 to y_7 where the mediating channels y_2, y_3, y_4, y_5, y_6 provide the links. These linked list segments in fact act as queues implementing message buffers. In [PG15] we created buffer queues as a separate data structure in the operational semantics, but I no longer believe their syntactic overhead is warranted.

Since communication is buffered and we have no bound on the number of insertions, the buffer must also be unbounded. Continuing the example, if we now decide to delete an element, we have to send a del label and then a shift to change direction of the communication:

$$\begin{array}{ll}
\text{to provider} & \mathsf{ins}\ x_1\ \mathsf{ins}\ x_2\ \mathsf{ins}\ x_3\ \mathsf{del}\ \mathsf{shift} \\
\text{to client} &
\end{array}$$

At this point we wait for the response, which means that the stack provider has to drain the whole buffer, up to and including the final shift and then respond. The response in this case will be the label some, followed by the channel x_3, followed by a shift.

$$\begin{array}{ll}
\text{to provider} & \mathsf{ins}\ x_1\ \mathsf{ins}\ x_2\ \mathsf{ins}\ x_3\ \mathsf{del}\ \mathsf{shift} \\
\text{to client} & \mathsf{some}\ x_3\ \mathsf{shift} \\
\text{to provider} &
\end{array}$$

Continuing for a few more interaction cycles:

to provider	ins x_1 ins x_2 ins x_3 del shift
to client	some x_3 shift
to provider	del shift
to client	some x_2 shift
to provider	del shift
to client	some x_1 shift
to provider	del shift
to client	none close

5 Synchronous from Asynchronous Communication

We recall the polarized type of stacks.

$$\begin{aligned}\mathsf{stack}^-_{A^+} = \&\{\ & \mathsf{ins} : A^+ \setminus \mathsf{stack}^-_{A^+}, \\ & \mathsf{del} : \uparrow \oplus\{\ \mathsf{none} : \mathbf{1}, \\ & \qquad\quad \mathsf{some} : A^+ \bullet \downarrow \mathsf{stack}^-_{A^+}\}\}\end{aligned}$$

As noted, the implementation requires an unbounded buffer because there are no changes of direction (which would be indicated by shifts) if we continuously insert elements.

If we want to keep the buffer bounded we can force synchronization after each element has been inserted. Amazingly, we can express this just be inserting a double-shift like so:

$$\begin{aligned}\mathsf{stack}^-_{A^+} = \&\{\ & \mathsf{ins} : A^+ \setminus \uparrow \downarrow \mathsf{stack}^-_{A^+}, \\ & \mathsf{del} : \uparrow \oplus\{\ \mathsf{none} : \mathbf{1}, \\ & \qquad\quad \mathsf{some} : A^+ \bullet \downarrow \mathsf{stack}^-_{A^+}\}\}\end{aligned}$$

Operationally, the first shift ($\uparrow$) will receive a shift message *from* the client, while the second shift ($\downarrow$) will sent a shift *to* the client and then wait again for messages. This second shift acts as an acknowledgment that all messages before have been received. This is because communication can only change direction when the buffer is empty, that is, the shift at the end of a message sequence has been received. Here is an example interaction, with

the additional synchronization:

to provider	ins x_1 shift
to client	shift
to provider	ins x_2 shift
to client	shift
to provider	ins x_3 shift
to client	shift
to provider	del shift
to client	some x_3 shift
to provider	...

We can now calculate the needed buffer size for this particular data structure and see that it is 3: The longest sequence of messages that can be in the buffer at any time are 'ins x shift' (to the provider) and 'some x shift' (to the client). In general, the calculation finds the longest path of the same polarity through the graph consisting of all the mutually recursive type definitions. If that is infinite, the buffer must be unbounded.

If we want to force fully synchronous communication, that is, every sent message waits for an acknowledgment, we just have to insert a double shift at every point in a type. The first shift sends a shift message, while the one immediately following waits for the acknowledgment in the form of a shift that is returns. This is a bit overly agressive, because if there is already a shift in the type, it is not necesary to make it a triple-shift (see [PG15] for more detail).

One pleasing property of this approach is that synchronization points are not left to the implementation, but are manifest in the type. Contrast this with Section 1 where it is only a matter of the code we write whether we want to force asynchrony in an otherwise synchronous language. This, and the fact that it is a more realistic abstraction, leads me to believe that asynchrony is the better default. The good news is that the proof theory supports both points of view, so there is no real right or wrong.

Exercises

Exercise 1 Provide an implementation of asynchronous send for $\backslash L^*$ as for $\bullet R^*$ in Section 1, and verify that synchronous and asynchronous forms are related by a commuting reduction followed by an identity reduction. If not, explain.

Exercise 2 Recall the correspondence between synchronous and asynchronous sends given at the end of Section 1 which was derived from $\bullet R^*$.

$$\mathsf{send}\ x\ w\ ;\ P \quad \simeq \quad x' \leftarrow [x'/x]P\ ;\ \mathsf{send}\ x\ w\ ;\ x \leftarrow x'$$

Provide analogous correspondences for $\mathbf{1}R$, $\oplus R$, $\&L$, and $\backslash L^*$.

Exercise 3 Develop an alternative semantics for forwarding $x \leftarrow y$ which involves communication only with directly connected processes (either along x or y) rather than the heavy-handed global replacement we have used so far. How are your rules related to the identity reductions (which can be read off from the proof of admissibility of cut)?

Exercise 4 Polarize each of the following types and analyze the maximal buffer sizes. If unbounded, insert synchronization points and then analyze the finite buffer size required for the resulting types.

1. $\mathsf{list}_A = \oplus\{\mathsf{cons} : A \bullet \mathsf{list}_A, \mathsf{nil} : \mathbf{1}\}$
2. $\mathsf{seg}_A = \mathsf{list}_A \,/\, \mathsf{list}_A$
3. $\mathsf{mapper}_{AB} = \&\{\mathsf{next} : (B \bullet \mathsf{mapper}_{AB}) \,/\, A, \mathsf{done} : \mathbf{1}\}$
4. $\mathsf{t\_map}_{AB} = \mathsf{list}_A \setminus (\mathsf{mapper}_{AB} \setminus \mathsf{list}_B)$
5. $\mathsf{folder}_{AB} = \&\{\mathsf{next} : (B \setminus (B \bullet \mathsf{folder}_{AB})) \,/\, A, \mathsf{done} : \mathbf{1}\}$
6. $\mathsf{t\_fold}_{AB} = (B \bullet \mathsf{folder}_{AB} \bullet \mathsf{list}_A) \setminus B$

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.

[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.