Lecture Notes on Adjoint SAX

15-836: Substructural Logics
Frank Pfenning

Lecture 15
October 31, 2023

1 Introduction

The version of SAX we introduced in the last lecture is still purely linear, although with recursion available when considered as a programming language. Not every function we want to write is linear, however, so we pursue the adjoint approach to mix linear with nonlinear types. This can be generalized further to a preorder of modes as in Lecture 11. With nonlinear types we can then express *multicast* (one message is sent to multiple recipients) and *shared servers* (a provider has multiple clients). Making communication *asynchronous* is a critical to this generalization. For example, it is difficult to conceptualize what synchronous delivery of a message to multiple clients might mean without at least a notion of (logical) time.

In MPASS we had a natural notion of channel that remained stable throughout communication, with a changing type. In SAX all messages (except unit) contain a continuation channel. Is there still a stable underlying notion of channel? We explore this using *messages sequences* that avoid many instances of allocating fresh continuation channels. While not formalized in this lecture, message sequences allow us implement channels as queues and, in some cases, calculate a precise bound on maximal size of queue [Willsey et al., 2016]. Message sequences in the syntax also allow some programs to be written more compactly.

2 Adding Adjoint Modalities to SAX

We recall the syntax of of SAX; the typing rules and dynamics can be found at the end of Lecture 14. Even if not formally distinguished, we use x' to denote a continuation channel.

$$
\begin{array}{llcll}
\text{Messages} & M & ::= & k(x') & (\oplus, \&) \\
& & \mid & (y, x') & (\otimes, \multimap) \\
& & \mid & (\,) & (\mathbf{1}) \\
\text{Continuations} & K & ::= & (\ell(x') \Rightarrow P_\ell(x'))_{\ell \in L} & (\oplus, \&) \\
& & \mid & ((y, x') \Rightarrow P(y, x')) & (\otimes, \multimap) \\
& & \mid & ((\,) \Rightarrow P) & (\mathbf{1}) \\
\text{Processes} & P & ::= & x \leftarrow P(x) \,;\, Q(x) & \text{cut} \\
& & \mid & \mathbf{fwd}\ x\ y & \text{id} \\
& & \mid & \mathbf{send}\ x\ M & \\
& & \mid & \mathbf{recv}\ x\ K & \\
& & \mid & \mathbf{call}\ p\ x\ y_1 \ldots y_n &
\end{array}
$$

We see that messages and continuations do double-duty for a pair of dual types. But here is no dual to $\mathbf{1}$—why? Actually, there is one we just haven't used it. See Section 4 for what it would mean.

To generalize to mixed linear/nonlinear logic we introduce a second layer of types and shifts that go between them.

$$
\begin{array}{llcl}
\text{Structural Types} & A_{\mathsf{S}} & ::= & \ldots \mid {\uparrow} A_{\mathsf{L}} \\
\text{Linear Types} & A_{\mathsf{L}} & ::= & \ldots \mid {\downarrow} A_{\mathsf{S}}
\end{array}
$$

Based on the symmetries we have seen so far, we might conjecture that they are dual in a way so that we just need a single new form of message and continuation, respectively, for both of these constructs. And that's indeed the case. We write the logical rules in the form of SAX based on their polarity and then assign program terms. How does this work? Recall that in the move from the sequent calculus to its semi-axiomatic form, the invertible rules remain the same and the noninvertible ones are turned into axioms. The up shift is negative, so its right rule stays intact. By our presupposition, Δ consists only of structural propositions. The left rule is turned into an axiom. We use here the *implicit* form without explicit rules for weaakening and contraction, so we allow structural antecedents in the axioms, denoted by Δ_{S}.

$$
\frac{\Delta \vdash A_{\mathsf{L}}}{\Delta \vdash {\uparrow} A_{\mathsf{L}}}\ {\uparrow}R
\qquad\qquad
\frac{}{\Delta_{\mathsf{S}}, {\uparrow} A_{\mathsf{L}} \vdash A_{\mathsf{L}}}\ {\uparrow}L
$$

The rules suggest a transition from a channel to its continuation channel at a different mode. We write $\langle x' \rangle$ for this form message.

$$
\frac{\Delta \vdash P(x'_{\mathsf{L}}) :: (x'_{\mathsf{L}} : A_{\mathsf{L}})}{\Delta \vdash \mathbf{recv}\ x_{\mathsf{S}}\ (\langle x'_{\mathsf{L}} \rangle \Rightarrow P(x'_{\mathsf{L}})) :: (x_{\mathsf{S}} : {\uparrow} A_{\mathsf{L}})}\ {\uparrow}R
$$

$$
\frac{}{\Delta_{\mathsf{S}}, x_{\mathsf{S}} : {\uparrow} A_{\mathsf{L}} \vdash \mathbf{send}\ x_{\mathsf{S}}\ \langle x'_{\mathsf{L}} \rangle :: (x'_{\mathsf{L}} : A_{\mathsf{L}})}\ {\uparrow}L
$$

The downshift works out symmetrically. First, the logical rules.

$$\frac{}{\Delta_{\mathsf{S}}, A_{\mathsf{S}} \vdash {\downarrow} A_{\mathsf{S}}}\ {\downarrow}R \qquad \frac{\Delta, A_{\mathsf{S}} \vdash C_r}{\Delta, {\downarrow} A_{\mathsf{S}} \vdash C_r}\ {\downarrow}L$$

In the left rule, the succedent C_r could be linear or structural. Annotating it with processes:

$$\frac{}{\Delta_{\mathsf{S}}, x'_{\mathsf{S}} : A_{\mathsf{S}} \vdash \textbf{send}\ x_{\mathsf{L}}\ \langle x'_{\mathsf{S}} \rangle :: (x_{\mathsf{L}} :: {\downarrow} A_{\mathsf{S}})}\ {\downarrow}R$$

$$\frac{\Delta, x'_{\mathsf{S}} : A_{\mathsf{S}} \vdash Q(x'_{\mathsf{S}}) :: (z_r : C_r)}{\Delta, x_{\mathsf{L}} : {\downarrow} A_{\mathsf{S}} \vdash \textbf{recv}\ x_{\mathsf{L}}\ (\langle x'_{\mathsf{S}} \rangle \Rightarrow Q(x'_{\mathsf{S}})) :: (z_r : C_r)}\ {\downarrow}L$$

The cut rule may introduce either a linear or a structural channel and structural channels may be shared between the two branches. Since we have just two modes, L and S with S > L, there are three versions of cut and only two versions of the identity.

$$\frac{(\Delta \geq m \geq r) \quad \Delta_{\mathsf{S}}, \Delta \vdash A_m \quad \Delta_{\mathsf{S}}, \Delta', A_m \vdash C_r}{\Delta_{\mathsf{S}}, \Delta, \Delta' \vdash C_r}\ \mathsf{cut} \qquad \frac{}{\Delta_{\mathsf{S}}, A_m \vdash A_m}\ \mathsf{id}$$

We retain a nice symmetry and language for messages, continuations, and channels. However, the dynamics becomes more complicated because some messages along shared channels should be persistent, and shared services may also need to be persistent. You can find the rules in a recent paper [Pfenning and Pruiksma, 2023][1]. We will come back to the mixed linear/nonlinear programs in the next lecture when we talk about *futures*.

3 An Example: mapreduce

As an example of a mixed linear/nonlinear program we write mapreduce on *linear* trees with data at the leaves.

$$\mathsf{tree}_A = \oplus\{\mathsf{node} : \mathsf{tree}_A \otimes \mathsf{tree}_A, \mathsf{leaf} : A\}$$

$$\mathit{mapreduce}_{AB}\ (r : B)\ (f_{\mathsf{S}} : {\uparrow}(B \otimes B \multimap B))\ (h_{\mathsf{S}} : {\uparrow}(A \multimap B))\ (t : \mathsf{tree}_A) = \ldots$$

Here, f_{S} and h_{S} are variables of structural type because f is used at every node and h is used at every leaf. We omit the annotation of the linear variables. The type parameters A and B themselves are *linear* so we did not write $f_{\mathsf{S}} : B \times B \to B$ but There is a corresponding version where A and B are structural types (which would require changing the type for trees). We start by receiving from t.

[1]Available at https://www.cs.cmu.edu/~fp/papers/coordination23.pdf

```
$\mathsf{tree}_A = \oplus\{\mathsf{node} : \mathsf{tree}_A \otimes \mathsf{tree}_A, \mathsf{leaf} : A\}$

$\mathit{mapreduce}_{AB}\ (y : B)\ (f_{\mathsf{S}} : \uparrow(B \otimes B \multimap B))\ (h_{\mathsf{S}} : \uparrow(A \multimap B))\ (t : \mathsf{tree}_A) =$
  $\mathbf{recv}\ t\ (\ \mathsf{node}(l, r) \Rightarrow \ldots$
          $\mid \mathsf{leaf}(x) \Rightarrow \ldots\ )$
```

In the case of a node we need to make two (hopefully parallel!) recursive calls. In order to type them it is important that f_{S} and h_{S} can be shared. And, yes, these two calls proceed independently, each given a fresh destination y_i.

```
$\mathsf{tree}_A = \oplus\{\mathsf{node} : \mathsf{tree}_A \otimes \mathsf{tree}_A, \mathsf{leaf} : A\}$

$\mathit{mapreduce}_{AB}\ (y : B)\ (f_{\mathsf{S}} : \uparrow(B \otimes B \multimap B))\ (h_{\mathsf{S}} : \uparrow(A \multimap B))\ (t : \mathsf{tree}_A) =$
  $\mathbf{recv}\ t\ (\ \mathsf{node}(l, r) \Rightarrow y_1 \leftarrow \mathbf{call}\ \mathit{mapreduce}_{AB}\ y_1\ f_{\mathsf{S}}\ h_{\mathsf{S}}\ l\ ;$
                        $y_2 \leftarrow \mathbf{call}\ \mathit{mapreduce}_{AB}\ y_2\ f_{\mathsf{S}}\ h_{\mathsf{S}}\ r\ ;$
                        $\ldots$
          $\mid \mathsf{leaf}(x) \Rightarrow \ldots\ )$
```

Next, we'd like to call f on y_1 and y_2 but before we do we need to "unwrap" f_{S} to obtain the underlying linear function f_{L}. For this purpose we need a send action because $\uparrow(B \otimes B \multimap B)$ is a negative type. The process providing f_{S} is waiting first for a linear continuation channel f_{L} and then a pair of following that.

```
$\mathsf{tree}_A = \oplus\{\mathsf{node} : \mathsf{tree}_A \otimes \mathsf{tree}_A, \mathsf{leaf} : A\}$

$\mathit{mapreduce}_{AB}\ (y : B)\ (f_{\mathsf{S}} : \uparrow(B \otimes B \multimap B))\ (h_{\mathsf{S}} : \uparrow(A \multimap B))\ (t : \mathsf{tree}_A) =$
  $\mathbf{recv}\ t\ (\ \mathsf{node}(l, r) \Rightarrow y_1 \leftarrow \mathbf{call}\ \mathit{mapreduce}_{AB}\ y_1\ f_{\mathsf{S}}\ h_{\mathsf{S}}\ l\ ;$
                        $y_2 \leftarrow \mathbf{call}\ \mathit{mapreduce}_{AB}\ y_2\ f_{\mathsf{S}}\ h_{\mathsf{S}}\ r\ ;$
                        $p : B \otimes B \leftarrow \mathbf{send}\ p\ (y_1, y_2)\ ;$
                        $f_{\mathsf{L}} \leftarrow \mathbf{send}\ f_{\mathsf{S}}\ \langle f_{\mathsf{L}} \rangle\ ;$
                        $\ldots$
          $\mid \mathsf{leaf}(x) \Rightarrow \ldots\ )$
```

Now we can send the pair p to f_{L}, but we also need to pass it a destination. But that's just the overall output channel y.

```
$\mathsf{tree}_A = \oplus\{\mathsf{node} : \mathsf{tree}_A \otimes \mathsf{tree}_A, \mathsf{leaf} : A\}$

$\mathit{mapreduce}_{AB}\ (y : B)\ (f_{\mathsf{S}} : \uparrow(B \otimes B \multimap B))\ (h_{\mathsf{S}} : \uparrow(A \multimap B))\ (t : \mathsf{tree}_A) =$
  $\mathbf{recv}\ t\ (\ \mathsf{node}(l, r) \Rightarrow y_1 \leftarrow \mathbf{call}\ \mathit{mapreduce}_{AB}\ y_1\ f_{\mathsf{S}}\ h_{\mathsf{S}}\ l\ ;$
                        $y_2 \leftarrow \mathbf{call}\ \mathit{mapreduce}_{AB}\ y_2\ f_{\mathsf{S}}\ h_{\mathsf{S}}\ r\ ;$
                        $p : B \otimes B \leftarrow \mathbf{send}\ p\ (y_1, y_2)\ ;$
                        $f_{\mathsf{L}} \leftarrow \mathbf{send}\ f_{\mathsf{S}}\ \langle f_{\mathsf{L}} \rangle\ ;$
                        $\mathbf{send}\ f_{\mathsf{L}}\ (p, y)$
          $\mid \mathsf{leaf}(x) \Rightarrow \ldots\ )$
```

The case of a leaf is simpler: we just unwrap the function h_{S} and apply it to the data of type A.

```
$\mathsf{tree}_A = \oplus\{\mathsf{node} : \mathsf{tree}_A \otimes \mathsf{tree}_A, \mathsf{leaf} : A\}$

$\mathit{mapreduce}_{AB}\ (y : B)\ (f_{\mathsf{S}} : \uparrow(B \otimes B \multimap B))\ (h_{\mathsf{S}} : \uparrow(A \multimap B))\ (t : \mathsf{tree}_A) =$
  $\mathbf{recv}\ t\ (\ \mathsf{node}(l, r) \Rightarrow y_1 \leftarrow \mathbf{call}\ \mathit{mapreduce}_{AB}\ y_1\ f_{\mathsf{S}}\ h_{\mathsf{S}}\ l$ ;
                         $y_2 \leftarrow \mathbf{call}\ \mathit{mapreduce}_{AB}\ y_2\ f_{\mathsf{S}}\ h_{\mathsf{S}}\ r$ ;
                         $p : B \otimes B \leftarrow \mathbf{send}\ p\ (y_1, y_2)$ ;
                         $f_{\mathsf{L}} \leftarrow \mathbf{send}\ f_{\mathsf{S}}\ \langle f_{\mathsf{L}} \rangle$ ;
                         $\mathbf{send}\ f_{\mathsf{L}}\ (p, y)$
          $\mid \mathsf{leaf}(x) \Rightarrow$     $h_{\mathsf{L}} \leftarrow \mathbf{send}\ h_{\mathsf{S}}\ \langle h_{\mathsf{L}} \rangle$ ;
                         $\mathbf{send}\ h_{\mathsf{L}}\ (x, y)$ )
```

This process has significant parallelism beyond just the two recursive calls. The process f receives the results from the recursive calls and a destination and it can run in parallel with the recursive calls! This is a difference between fork/join parallelism and futures (to be explored in the next lecture). In fork/join we'd have to synchronize when the pair p is formed; here synchronization occurs when f needs to receive from its argument channels.

4 Bottom

What is dual to $\mathbf{1}$? Presumably, since $\mathbf{1}$ is positive, it would be negative. Let's recall the rules for the unit.

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

If we flip sides, we see that the *succedent* needs to be empty for the right rule.

$$\dfrac{\Delta \vdash \cdot}{\Delta \vdash \bot}\ \bot R \qquad \dfrac{}{\bot \vdash \cdot}\ \bot L$$

We know how to check that these are correct: cut reduction and identity expansion (locally), and cut and identity elimination globally. Locally, everything is fine.

$$\dfrac{\dfrac{\begin{array}{c}\mathcal{D}' \\ \Delta \vdash \cdot\end{array}}{\Delta \vdash \bot}\ \bot R \qquad \dfrac{}{\bot \vdash \cdot}\ \bot L}{\Delta \vdash \cdot}\ \mathsf{cut}_\bot \quad \longrightarrow_R \quad \begin{array}{c}\mathcal{D}' \\ \Delta \vdash \cdot\end{array}$$

$$\dfrac{}{\bot \vdash \bot}\ \mathsf{id}_\bot \quad \longrightarrow_E \quad \dfrac{\dfrac{}{\bot \vdash \cdot}\ \bot L}{\bot \vdash \bot}\ \bot R$$

As expected, the process assignment doesn't require anything new. We observe that the left rule is just renamed into an axiom, but doesn't change.

$$\frac{\Delta \vdash P :: \cdot}{\Delta \vdash \textbf{recv}\ x\ ((\,) \Rightarrow P) :: (x : \bot)}\ \bot R \qquad \frac{}{x : \bot \vdash \textbf{send}\ x\ (\,) :: \cdot}\ \bot X$$

The way we have biased the intuitionistic judgments, a process $\Delta \vdash P :: \cdot$ computes for its own sake, without a client. And it could not be closed $\cdot \vdash P :: \cdot$ is not provable (except perhaps by abusing recursion in some way) so it doesn't fit well into our applications.

5 Message Sequences

As mentioned in the introduction, when we moved from synchronous to asynchronous communication, we needed to introduce continuation channels. Creating fresh channels for every message is a good model for the theory, but not plausible for an implementation. Could we introduce *message sequences* that appear on the same channel as a way not only to make the communication model more realistic, but also write more compact programs?

Intuitively, a message sequence just replaces the continuation channel with another message. Conversely, when receiving along a channel we no longer match against a single message at a time, but a whole message sequence.

$$\begin{array}{llcll}
\text{Message Sequence} & \overline{M} & ::= & k(\overline{M}) & (\oplus, \&) \\
 & & \mid & (y, \overline{M}) & (\otimes, \multimap) \\
 & & \mid & (\,) & (\mathbf{1}) \\
 & & \mid & x' & \text{cont. channel} \\
\text{Continuations} & \overline{K} & ::= & (\overline{M} \Rightarrow P \mid \overline{K}) \mid \cdot & \\
\text{Processes} & P & ::= & x \leftarrow P(x) \ ; Q(x) & \text{cut} \\
 & & \mid & \textbf{fwd}\ x\ y & \text{id} \\
 & & \mid & \textbf{send}\ x\ \overline{M} & \\
 & & \mid & \textbf{recv}\ x\ \overline{K} & \\
 & & \mid & \textbf{call}\ p\ x\ y_1 \ldots y_n &
\end{array}$$

Before we formalize that statics and dynamics of this extended language, we consider two examples to see where the formal development should lead us. We begin with `append`, which has a relatively simple use of pattern matching.

```
type bin = +{'b0 : bin, 'b1 : bin, 'e : 1}
type list = +{'cons : bin * list, 'nil : 1}

proc append (R : list) (L : list) (K : list) =
  recv L ( 'cons(x,L') => R' <- call append R' L' K ;
                          send R 'cons(x,R')
         | 'nil() => fwd R K )
```

This might expand to

```
proc append (R : list) (L : list) (K : list) =
  recv L ( 'cons(p) => recv p ((x,L') =>
                R' <- call append R' L' K ;
                p' : bin * list <- send p' (x,R') ;
                send R 'cons(p'))
         | 'nil(u) => recv u (() => fwd R K) )
```

The second is process to compute $\lfloor \frac{x}{2} \rfloor$ for x in unary form.

```
type nat = +{'zero : 1, 'succ : nat}

proc half (r : nat) (x : nat) =
  recv x ( 'zero() => send r 'zero()
         | 'succ('zero()) => send r 'zero()
         | 'succ('succ(y)) => h <- call half h y ;
                              send r 'succ(h) )
```

This might expand to

```
proc half (r : nat) (x : nat) =
recv x ( 'zero(x') =>
           recv x' (() => u : 1 <- send u () ;
                          send r 'zero(u))
       | 'succ(x') =>
           recv x' ( 'zero(x'') =>
                       recv x'' (() => u : 1 <- send u () ;
                       send r 'zero(u))
                   | 'succ(y) => h <- call half' h y ;
                                 send r 'succ(h) ) )
```

We now have to update the statics and dynamics for this enriched language in a way that is consistent with SAX. As we often do, we start with the statics. Message sequences seem more manageable than the more general form of pattern matching, so we start with them. There are two classes of rules, one for positive types that send to a client and one for negative types that sends to a provider. In the premise, we have to check that the message sequence fits the type of the channel, but the original channel is no longer needed.

$$\frac{\Delta \vdash \overline{M} : \lfloor A \rfloor}{\Delta \vdash \mathbf{send}\ x\ \overline{M} :: (x : A)}\ \mathsf{send}^+$$

Now we have rules for each of the positive types with the corresponding messages.

$$\frac{\Delta \vdash \overline{M} : \lfloor A_k \rfloor}{\Delta \vdash k(\overline{M}) : \lfloor \oplus\{\ell : A_\ell\}_{\ell \in L} \rfloor} \oplus R$$

$$\frac{\Delta \vdash \overline{M} : \lfloor B \rfloor}{\Delta, y : A \vdash (y, \overline{M}) : \lfloor A \otimes B \rfloor} \otimes R \qquad \frac{}{\cdot \vdash (\,) : \lfloor \mathbf{1} \rfloor} \mathbf{1}R$$

When we encounter an actual continuation channel rather than a message, we use an instance of the identity.

$$\frac{}{x' : A \vdash x' : \lfloor A \rfloor} \mathsf{id}_R$$

Do these rules look familiar? They should! Think about it before you read on.

These are *almost* the rules for *right focus* except that we can apply the identity to finish the focusing phase for any proposition A, not just for atoms and negative propositions. Similar, the premise antecedent $y : A$ in $\otimes R$ arises from the identity on A, rather than focusing on $\lfloor A \rfloor$ on the right.

These differences reflects differences between *proof construction*, where we would like to chain together inferences as much as possible to minimize nondeterminism, and *proof reduction* where we would like the freedom to write sequences as long or as short as we would like to. We therefore call this *partial focusing* which is also reflected in the notation $\lfloor A \rfloor$. An interesting property of partial focusing is that the right rules that had become axioms have turned back into right rules!

To complete this thought, message sequences of negative type correspond to *partial left focus*! We use a new notation here, writing δ for a singleton succedent $z : C$.

$$\frac{\Delta, \lfloor A \rfloor \vdash \overline{M} :: \delta}{\Delta, x : A \vdash \textbf{send}\ x\ \overline{M} :: \delta}\ \mathsf{send}_L$$

$$\frac{\Delta, \lfloor A_k \rfloor \vdash \overline{M} :: \delta}{\Delta, \lfloor \&\{\ell : A_\ell\}_{\ell \in L} \rfloor \vdash k(\overline{M}) :: \delta}\ \&L \qquad \frac{\Delta, \lfloor B \rfloor \vdash \overline{M} :: \delta}{\Delta, y : A, \lfloor A \multimap B \rfloor \vdash (y, \overline{M}) :: \delta}\ \multimap L$$

$$\frac{}{\lfloor A \rfloor \vdash x' :: (x' : A)}\ \mathsf{id}_L$$

6 Pattern Matching

Along with sequences of messages, we also changed continuations so they can receive and discriminate whole message sequences. This looks more complicated than message sequences themselves since patterns can be nested and appear in different orders and to different depths. To allow this we define the operation of *projection* that filters out cases from a complex pattern match.

We might conjecture that since message sequences correspond to (partial) focusing that pattern matching will correspond to (partial) inversion. That's not far-fetched since the corresponding logical connectives are indeed invertible!

We start on the right.

$$\frac{\Delta\ ;\ A \vdash \overline{K} :: \delta}{\Delta, x : A \vdash \textbf{recv}\ x\ \overline{K} :: \delta}\ \mathsf{recv}_L$$

The judgment form $\Delta\ ;\ A \vdash \overline{K} :: \delta$ is inspired by the notation for inversion on the left, $\Delta\ ;\ \Omega \vdash C$. In the case of message sequences, the ordered inversion context Ω will always be a singleton.

We construct the rules such that $\overline{K}$ in the judgment $\Delta\ ;\ A \vdash \overline{K} :: \delta$ and later $\Delta \vdash \overline{K} : A$ cannot be empty. This rules out the case where the there is no branch for a (well-typed) message received.

We start with conjunction this time. When the antecedent is $A \otimes B$ then we need to receive a channel of type A and then B has to be matched against the remaining continuations.

$$\frac{\Delta, y : A\ ;\ B \vdash \overline{K} \mathbin{@} (y, \_) :: \delta}{\Delta\ ;\ A \otimes B \vdash \overline{K} :: \delta}\ \otimes L$$

The projection is only defined if all patterns are pairs, and rule can only be applied if the projection $\overline{K} \mathbin{@} (y, \_)$ is nonempty. Projection also instantiates the variable bound in the patterns with y so that all branches in $\overline{K} \mathbin{@} (y, \_)$ use the same variable. The fact that $y : A$ is added to the antecedents and not to the ordered context is a departure from the usual inversion, but important to enforce matching against message sequences (not trees).

$$\begin{array}{llcl}
(z, \overline{M}) \Rightarrow P(z) \mid \overline{K} & @ & (y, \_) & = \overline{M} \Rightarrow P(y) \mid \overline{K} \mathbin{@} (y, \_) \\
(z, \overline{M}) \Rightarrow P(z) & @ & (y, \_) & = \overline{M} \Rightarrow P(y) \\
\overline{K} & @ & (y, \_) & \text{undefined otherwise}
\end{array}$$

Here we have abbreviated $\overline{M} \Rightarrow P \mid \cdot$ as $\overline{M} \Rightarrow P$. Since we would like the language to remain deterministic, at the unit type there must only be a single branch and we revert back to the ordinary typing judgment for processes.

$$\frac{\Delta \vdash \overline{K} \mathbin{@} (\,) :: \delta}{\Delta\ ;\ \mathbf{1} \vdash \overline{K} :: \delta}\ \mathbf{1}L$$

$$\begin{array}{llcl}
((\,) \Rightarrow P) & @ & (\,) & = P \\
\overline{K} & @ & (\,) & \text{undefined otherwise}
\end{array}$$

Finally we come to external choice. The patterns must all start with a label, so we project onto each label of the external choice.

$$\frac{\Delta\ ;\ A_\ell \vdash \overline{K} \mathbin{@} \ell(\_) :: \delta \quad (\forall \ell \in L)}{\Delta\ ;\ \oplus\{\ell : A_\ell\}_{\ell \in L} \vdash \overline{K} :: \delta}\ \oplus L$$

If $\overline{K} \mathbin{@} \ell(\_)$ is empty then this means the label ℓ is not accounted for among the patterns even though it should be. In this case we won't be able to complete the typing derivation because the other rules $\oplus L$, $\mathbf{1}L$, and $\mathsf{cont/var}^+$ (see below) all require the continuation to have at least one branch. Furthermore, we enforce that all branches start with a label in L. This latter condition is not strictly necessary for progress and preservation but retains the connection to the logical inference rules.

$$\begin{array}{llcll}
(\ell(\overline{M}) \Rightarrow P \mid \overline{K}) & @ & \ell(\_) & = \overline{M} \Rightarrow P \mid (\overline{K} \mathbin{@} \ell(\_)) & \\
(k(\overline{M}) \Rightarrow P \mid \overline{K}) & @ & \ell(\_) & = \overline{K} \mathbin{@} \ell(\_) & \text{for } k \neq \ell \text{ and } k \in L \\
(\cdot) & @ & \ell(\_) & = \cdot & \\
\overline{K} & @ & \ell(\_) & \text{undefined otherwise} &
\end{array}$$

The last case arises when the pattern consists of a single branch with a single variable. We just revert to the usual typing judgment.

$$\frac{\Delta, x' : A \vdash P(x') :: \delta}{\Delta \,;\, A \vdash (x' \Rightarrow P(x')) :: \delta}\ \mathsf{cont/var}^{+}$$

This rule also marks a difference to full inversion: A does not need to be a negative type.

We show the remaining rules for right inversion on negative types without further discussion since we have seen all the necessary ideas already.

$$\frac{\Delta \vdash \overline{K} : A}{\Delta \vdash \mathbf{recv}\ x\ \overline{K} :: (x : A)}\ \mathbf{recv}_R$$

$$\frac{\Delta, y : A \vdash \overline{K}\ \texttt{@}\ (y, \_) : B}{\Delta \vdash \overline{K} : A \multimap B}\ {\multimap}R \qquad \frac{\Delta \vdash \overline{K}\ \texttt{@}\ \ell(\_) : A_\ell \quad (\forall \ell \in L)}{\Delta \vdash \overline{K} : \&\{\ell : A_\ell\}_{\ell \in L}}\ \&R$$

$$\frac{\Delta \vdash P(x') :: (x' : A)}{\Delta \vdash (x' \Rightarrow P(x')) : A}\ \mathsf{cont/var}^{-}$$

7 Dynamics for Message Sequences

We could give a dynamics for message sequences and general pattern matching directly on the extended syntax. We pursue here a different approach where the dynamics is defined by translation into the SAX core language. This translation has to create fresh channels for the middle of message sequences, and has to break up complex patterns into a nested matches of simple patterns.

The translations are type-directed, so we translate **send** $x\ \overline{M}$ with metalevel function $\mathbf{send}^*\ (x : A)\ \overline{M} = P$ where P uses only simple messages. Similarly, a **recv** $x\ \overline{K}$ is translated by $\mathbf{recv}^*\ (x : A)\ \overline{K} = P$. We keep in mind the following properties (becoming theorems) where the conclusion is typed in the original SAX system.

1. If $\Delta \vdash \mathbf{send}\ x\ \overline{M} :: (x : A)$ then $\Delta \vdash (\mathbf{send}^*\ (x : A)\ \overline{M}) :: (x : A)$
2. If $\Delta, x : A \vdash \mathbf{send}\ x\ \overline{M} :: \delta$ then $\Delta \vdash (\mathbf{send}^*\ (x : A)\ \overline{M}) :: \delta$
3. If $\Delta, x : A \vdash \mathbf{recv}\ x\ \overline{K} :: \delta$ then $\Delta, x : A \vdash (\mathbf{recv}^*\ x\ \overline{K}) :: \delta$
4. If $\Delta \vdash \mathbf{recv}\ x\ \overline{K} :: (x : A)$ then $\Delta \vdash (\mathbf{recv}^*\ x\ \overline{K}) :: (x : A)$

We have to generalize these properties to talk about partial focusing, which we leave as an exercise

$$\begin{array}{lcl}
\mathbf{send}^*\ (x : \oplus\{\ell : A_\ell\})\ k(\overline{M}) & = & x' \leftarrow \mathbf{send}^*\ (x' : A_k)\ \overline{M}\ ;\ \mathbf{send}\ x\ k(x') \\
\mathbf{send}^*\ (x : A \otimes B)\ (y, \overline{M}) & = & x' \leftarrow \mathbf{send}^*\ x'_B\ \overline{M}\ ;\ \mathbf{send}\ x\ (y, x') \\
\mathbf{send}^*\ (x : \mathbf{1})\ (\,) & = & \mathbf{send}\ x\ (\,) \\
\mathbf{send}^*\ (x : A)\ x' & = & \mathbf{fwd}\ x\ x' \\[1ex]
\mathbf{send}^*\ (x : \&\{\ell : A_\ell\})\ k(\overline{M}) & = & x' \leftarrow \mathbf{send}\ x\ k(x')\ ;\ \mathbf{send}^*\ (x' : A_k)\ \overline{M} \\
\mathbf{send}^*\ (x : A \multimap B)\ (y, \overline{M}) & = & x' \leftarrow \mathbf{send}\ x\ (y, x')\ ;\ \mathbf{send}^*\ (x' : B)\ \overline{M} \\
\mathbf{send}^*\ (x : A)\ x' & = & \mathbf{fwd}\ x'\ x
\end{array}$$

$$\begin{array}{lcl}
\mathbf{recv}^*\ (x : \oplus\{\ell : A_\ell\}_{\ell \in L})\ \overline{K} & = & \mathbf{recv}\ x\ (\ell(x') \Rightarrow \mathbf{recv}^*\ (x : A_\ell)\ (\overline{K}\ @\ \ell(\_)))_{\ell \in L} \\
\mathbf{recv}^*\ (x : A \otimes B)\ \overline{K} & = & \mathbf{recv}\ x\ ((y, x') \Rightarrow \mathbf{recv}^*\ (x : B)\ (\overline{K}\ @\ (y, \_))) \\
\mathbf{recv}^*\ (x : \mathbf{1})\ \overline{K} & = & \mathbf{recv}\ x\ ((\,) \Rightarrow \overline{K}\ @\ (\,)) \\
\mathbf{recv}^*\ (x : A)\ (x' \Rightarrow P(x')) & = & P(x) \\[1ex]
\mathbf{recv}^*\ (x : \&\{\ell : A_\ell\}_{\ell \in L})\ \overline{K} & = & \mathbf{recv}\ x\ (\ell(x') \Rightarrow \mathbf{recv}^*\ (x : A_\ell)\ (\overline{K}\ @\ \ell(\_)))_{\ell \in L} \\
\mathbf{recv}^*\ (x : A \multimap B)\ \overline{K} & = & \mathbf{recv}\ x\ ((y, x') \Rightarrow \mathbf{recv}^*\ (x : B)\ (\overline{K}\ @\ (y, \_))) \\
\mathbf{recv}^*\ (x : A)\ (x' \Rightarrow P(x')) & = & P(x)
\end{array}$$

References

Frank Pfenning and Klaas Pruiksma. Relating message passing and shared memory, proof-theoretically. In S. Jongmans and A. Lopes, editors, *25th International Conference on Coordination Models and Languages (COORDINATION 2023)*, pages 3–27, Lisbon, Portugal, June 2023. Springer LNCS 13908. Notes to an invited talk.

Max Willsey, Rokhini Prabhu, and Frank Pfenning. Design and implementation of Concurrent C0. In *Fourth International Workshop on Linearity*, pages 73–82. EPTCS 238, June 2016.