Lecture Notes on Linear Message Passing II

15-836: Substructural Logics
Frank Pfenning

Lecture 6
September 14, 2023

1 Introduction

We continue the development of a concurrent linear message-passing language. In the last lecture we introduced only the constructs below. Spawn and forward work for arbitrary types, other others apply to a specific type. The actions are viewed from the provider's point of view.

Process	**Action**	**Rule**	**Type**
$x_A \leftarrow P(x) \; ; \; Q(x)$	spawn	cut	A
$\mathbf{fwd}\; x\; y$	forward	id	A
$\mathbf{send}\; x\; k \; ; \; P$	send label k	$\oplus R$	$\oplus\{\ell : A_\ell\}_{\ell \in L}$
$\mathbf{recv}\; x\; (\ell \Rightarrow Q_\ell)_{\ell \in L}$	receive label k	$\oplus L$	$\oplus\{\ell : A_\ell\}_{\ell \in L}$
$\mathbf{send}\; x\; (\,)$	send unit	$\mathbf{1}R$	$\mathbf{1}$
$\mathbf{recv}\; x\; ((\,) \Rightarrow Q)$	receive unit	$\mathbf{1}L$	$\mathbf{1}$
$\mathbf{call}\; p\; x\; y_1 \ldots y_n$	call process p		

The last one is not associated with any particular type but invokes a defined process with name p, passing it x and $y_1, \ldots, y_n$. This is how recursion enters (and exceeds the sequent calculus of linear logic) because these processes may be recursively defined.

We begin by reformulating the dynamics slightly from the previous lecture, recognizing it as a form of *linear inference*!

2 Dynamics as Linear Inference

We mentioned in an earlier lecture that linear inference provides a form of *true concurrency* because with the CBA-graphs we cannot actually tell the order of independent inferences. The rules from the last lecture can in fact be written in the form of linear inference and take advantage of the earlier observation to get a concurrent semantics "for free".

We need a predicate, say proc, so that $\mathsf{proc}(P)$ is a semantic object representing a running process in state P. Today, we refer to these as *objects* rather than propositions, because we have already coopted those to represent types! We write channels now as a, b and c, because such channels replace variables in the program at runtime. They are typed just as variables are. The whole collection of semantic objects form a *configuration* $\mathcal{C}$, where the order is irrelevant.

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

Recall that such a rule picks out two matching objects from the linear state and replaces them by the conclusion. Similarly:

$$\frac{\mathsf{proc}(\mathbf{send}\ a\ k\ ;\ P)\quad \mathsf{proc}(\mathbf{recv}\ a\ (\ell \Rightarrow Q_\ell)_{\ell\in L})\quad (k \in L)}{\mathsf{proc}(P)\qquad \mathsf{proc}(Q_k)}$$

$$\frac{\mathsf{proc}(x_A \leftarrow P(x)\ ;\ Q(x))\quad (a\ \text{fresh})}{\mathsf{proc}(P(a))\qquad \mathsf{proc}(Q(a))}$$

The condition on the freshness of a is somewhat problematic since the premise matches only part of the linear state, but a must be *globally fresh* in the whole state. At the more technical level, this corresponds to *existential quantification*, but since we haven't discussed quantifiers just rely on this global freshness condition.

Now that we know that the dynamics is "just" linear inference, we'll revert to the left-to-right arrow notation for the rules on semantic objects. We actually introduced this $\Delta \longrightarrow \Delta'$ for linear inference before as a more convenient notation (not to be confused with the Δ that types channels in the antecedent of a sequent). Here, it will be $\mathcal{C} \longrightarrow \mathcal{C}'$.

3 Typing Finite Internal Choice

We actually defined only the typing rules for the binary internal choice. Here are the ones for finite sums (or: internal choice between a finite number of alternatives).

$$\frac{\Delta \vdash A_k \quad (k \in L)}{\Delta \vdash \oplus\{\ell : A_\ell\}_{\ell\in L}}\ \oplus R \qquad\qquad \frac{\Delta, A_\ell \vdash C \quad (\forall \ell \in L)}{\Delta, \oplus\{\ell : A_\ell\}_{\ell\in L} \vdash C}\ \oplus L$$

The $\oplus L$ rule has one premise for each $\ell \in L$ and is therefore finitary. Adding process terms yields typing rules:

$$\frac{\Delta \vdash P :: (x : A_k) \quad (k \in L)}{\Delta \vdash \textbf{send}\ x\ k\ ;\ P :: (x : \oplus\{\ell : A_\ell\}_{\ell \in L})}\ \oplus R$$

$$\frac{\Delta, x : A_\ell \vdash Q_\ell :: (z : C) \quad (\forall \ell \in L)}{\Delta, x : \oplus\{\ell : A_\ell\}_{\ell \in L} \vdash \textbf{recv}\ x\ (\ell \Rightarrow Q_\ell)_{\ell \in L} :: (z : C)}\ \oplus L$$

It is convenient to assume that the set L is nonempty, but it certainly wouldn't be problematic logically since $\mathbf{0} = \oplus\{\,\}$.

4 Sending Channels Along Channels

A characteristic of Milner's π-calculus [Milner, 1999] is that one can send channels along channels, changing how the processes are interconnected. In the linear setting, such a protocol is the computational interpretation of $A \otimes B$ and $A \multimap B$. Since we have worked with positive types so far, we start with $A \otimes B$. First, let's review and analyze the logic rules.

$$\frac{\Delta_1 \vdash A \quad \Delta_2 \vdash B}{\Delta_1, \Delta_2 \vdash A \otimes B}\ \otimes R \qquad \frac{\Delta', A, B \vdash C}{\Delta', A \otimes B \vdash C}\ \otimes L$$

The cut reduction between these two rules is straightforward

$$\frac{\dfrac{\Delta_1 \vdash A \quad \Delta_2 \vdash B}{\Delta_1, \Delta_2 \vdash A \otimes B}\ \otimes R \qquad \dfrac{\Delta', A, B \vdash C}{\Delta', A \otimes B \vdash C}\ \otimes L}{\Delta', \Delta_1, \Delta_2 \vdash C}\ \mathsf{cut}_{A \otimes B}$$

$$\longrightarrow_R \quad \frac{\Delta_2 \vdash B \qquad \dfrac{\Delta_1 \vdash A \quad \Delta', A, B \vdash C}{\Delta', \Delta_1, B \vdash C}\ \mathsf{cut}_A}{\Delta', \Delta_1, \Delta_2 \vdash C}\ \mathsf{cut}_B$$

From the point of view of processes, the provider of the channel $x : A \otimes B$ spawns two processes, one providing A and one providing B. Both will be connected to the original client of x. There are several pragmatic reasons to chose the following alternative right rule, shown also with the channel-annotated versions.

$$\frac{\Delta \vdash B}{\Delta, A \vdash A \otimes B}\ \otimes R^* \qquad \frac{\Delta \vdash x : B}{\Delta, w : A \vdash x : A \otimes B}\ \otimes R^*$$

$$\frac{\Delta', A, B \vdash C}{\Delta', A \otimes B \vdash C}\ \otimes L \qquad \frac{\Delta', y : A, x : B \vdash C}{\Delta', x : A \otimes B \vdash C}\ \otimes L$$

A process ending with the $\otimes R^*$ rule will send the channel $w : A$ along $x : A \otimes B$ and continue to provide $x : B$. Reusing the send/receive style syntax for these processes, we get:

$$\frac{\Delta \vdash P :: (x : B)}{\Delta, w : A \vdash \textbf{send}\ x\ w\ ;\ P :: (x : A \otimes B)}\ \otimes R^*$$

$$\frac{\Delta', y : A, x : B \vdash Q(y) :: (z : C)}{\Delta', x : A \otimes B \vdash \textbf{recv}\ x\ (y \Rightarrow Q(y)) :: (z : C)}\ \otimes L$$

The computation rule just passes the given channel.

$$\mathsf{proc}(\textbf{send}\ a\ b\ ;\ P), \mathsf{proc}(\textbf{recv}\ a\ (y \Rightarrow Q(y))) \quad \longrightarrow \quad \mathsf{proc}(P), \mathsf{proc}(Q(b))$$

A nice property of this formulation with $\otimes R^*$ as compared to the standard two-premise version $\otimes R$ is that the only rule that spawns a new process is now the cut rule. Also, every construct (send or receive) has a single continuation, except for sending the unit which terminates a process.

But is this rule substitution actually okay? We should check that (a) $\otimes R^*$ is sound in the sense that we can derive it using $\otimes R$, and (b) that it is complete in the sense that we can use it to derive $\otimes R$. This is an instructive exercise, so you might want to give it a try before reading on.

In one direction we require an identity, in the other direction a cut. If we make up a syntax for the standard two-premise rule, we can translate the derivations below into small programs.

$$\dfrac{\dfrac{}{A \vdash A}\,\mathsf{id} \quad \Delta \vdash B}{\Delta, A \vdash A \otimes B}\,\otimes R \qquad\qquad \dfrac{\Delta_1 \vdash A \quad \dfrac{\Delta_2 \vdash B}{A, \Delta_2 \vdash A \otimes B}\,\otimes R^*}{\Delta_1, \Delta_2 \vdash A \otimes B}\,\mathsf{cut}$$

The downside of a wholesale *replacement* of $\otimes R$ with $\otimes R^*$ is that cut elimination no longer holds in its usual formulation. So the new calculus is most easily justified by translation as we did here. Alternatively, one can formulate an alternative "cut elimination" theorem that allows some cuts that satisfy the subformula property (so-called *analytic cuts*) to remain in proofs. We will not pursue this further here since it does not impact the computational interpretation of the MPASS language.

5 Example: Sequences

In a functional language it is often convenient to work with lists. Here, they are sequences of channels of some arbitrary type A, but we still call them lists.

$$\mathsf{list}_A = \oplus\{\mathsf{nil} : \mathbf{1}, \mathsf{cons} : A \otimes \mathsf{list}_A\}$$

This type is entirely positive, so messages still only flow from provider to client. We can define the standard constructors corresponding to the alternatives in the sum. Since MPASS at present does not support polymorphism, we restrict ourselves to lists of binary numbers. It should be clear that nothing depends on this choice. We can define the boilerplate constructors corresponding to nil and cons.

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

proc nil (l : list) = send l 'nil ; send l ()
proc cons (l : list) (x : bin) (k : list) =
  send l 'cons ; send l x ; fwd l k
```

A standard *functional* program appends two lists. We can write the same program in different ways. In Listing 1 we show a version where the recursive call to append is a *tail call* (which usually isn't possible in a functional language). We annotate each line with the typing of the continuation of the process following this line. In the file lecture6.mps there is also a test case for the append process.

6 External Choice

We now come to negative connectives, starting with external choice. Analogously to internal choice, we also generalize external choice to have a finite number of

```
proc append (l : list) (k1 : list) (k2 : list) =
  recv k1 ( 'nil =>           % k1 : 1, k2 : list |- l : list
              recv k1 (() =>  % k2 : list |- l : list
              fwd l k2)
          | 'cons =>          % k1 : bin * list, k2 : list |- l : list
              send l 'cons ;  % k1 : bin * list, k2 : list |- l : bin * list
              recv k1 (x =>   % x : bin, k1 : list, k2 : list |- l : bin * list
              send l x ;      % k1 : list, k2 : list |- l : list
              call append l k1 k2) )
```

Listing 1: Append process in MPASS

alternatives. First, the purely logical rules.

$$\frac{\Delta \vdash A_\ell \quad (\forall \ell \in L)}{\Delta \vdash \&\{\ell : A_\ell\}_{\ell \in L}}\ \&R \qquad \frac{\Delta, A_k \vdash C \quad (k \in L)}{\Delta, \&\{\ell : A_\ell\}_{\ell \in L} \vdash C}\ \&L$$

Conversely to the rules for internal choice, the provider has to be prepared for the client to choose a suitable $k \in L$. We also see that if these two proofs are combined with a cut, then in this case the *clients sends a label to the provider*. This is a characteristic of negative types, where the right rules are invertible and therefore contain no information.

Because of the symmetry between external and internal choice, we can reuse the process notation, just swapping the roles of provider and client.

$$\frac{\Delta \vdash P_\ell :: (x : A_\ell) \quad (\forall \ell \in L)}{\Delta \vdash \mathbf{recv}\ x\ (\ell \Rightarrow P_\ell)_{\ell \in L} :: (x : \&\{\ell : A_\ell\}_{\ell \in L})}\ \&R$$

$$\frac{\Delta, x : A_k \vdash Q :: (z : C) \quad (k \in L)}{\Delta, x : \&\{\ell : A_\ell\}_{\ell \in L} \vdash \mathbf{send}\ x\ k\ ;\ Q :: (z : C)}\ \&L$$

Furthermore, the same rule in the dynamics still applies and doesn't need to be changed! Writing it again in our linear notation, just for reference:

$$\mathsf{proc}(\mathbf{recv}\ a\ (\ell \Rightarrow P_\ell)_{\ell \in L}), \mathsf{proc}(\mathbf{send}\ a\ k\ ;\ Q) \quad \longrightarrow \quad \mathsf{proc}(P_k), \mathsf{proc}(Q) \quad (k \in L)$$

7 Linear Implication

Linear implication $A \multimap B$ is symmetric to $A \otimes B$: the provider *receives* a channel of type A instead of sending one. We also change the sequent calculus rules in an analogous way to avoid multiple premises for the $\multimap L$ rule.

$$\frac{\Delta, A \vdash B}{\Delta \vdash A \multimap B}\ \multimap R \qquad \frac{\Delta', B \vdash C}{\Delta', A, A \multimap B \vdash C}\ \multimap L^*$$

We move directly to the process assignment, leaving the logical investigation of these rules to Assignment 2.

$$\frac{\Delta, y : A \vdash P :: (x : B)}{\Delta \vdash \textbf{recv}\ x\ (y \Rightarrow P(y)) :: (x : A \multimap B)} \multimap R$$

$$\frac{\Delta', x : B \vdash Q :: (z : C)}{\Delta', w : A, x : A \multimap B \vdash \textbf{send}\ x\ w\ ;\ Q :: (z : C)} \multimap L^*$$

Once again, the computational rule for this construct already exists: the rule for tensor applies. We restate it for completeness.

$$\mathsf{proc}(\textbf{recv}\ a\ (y \Rightarrow P(y))), \mathsf{proc}(\textbf{send}\ a\ b\ ;\ Q) \quad \longrightarrow \mathsf{proc}(P(b)), \mathsf{proc}(Q)$$

8 Example: A Storage Server

We now use an interface to a storage server as an example incorporating negative types (external choice and linear implication). The client has the option between inserting or deleting an element from the store. When deleting, the provider replies with none if there is no element in the store, or some followed by the element if there is one.

$$\begin{aligned} \mathsf{store}_A = \&\{\ & \mathsf{ins} : A \multimap \mathsf{store}_A, \\ & \mathsf{del} : \oplus\{\ \mathsf{none} : \mathbf{1}, \mathsf{some} : A \otimes \mathsf{store}_A\ \}\ \} \end{aligned}$$

As for sequences, in the implementation we commit to the type A to be the binary numbers since MPASS doesn't currently support polymorphism.

Our implementation behaves like a stack; for a queue, see Assignment 3. The state of the store is represented by a sequence of processes, each holding one element, with the last one being empty. We define this empty process first. In the case of insert we have to start a `node` process with the element we received. The tail of the this new store must again be empty, so we have to spawn a new empty process.

```
proc empty (s : store) =
     recv s ( 'ins => recv s (x => e <- call empty e ;
                                   call node s x e )
            | 'del => send s 'none ; send s () )
```

When receiving a `'del` message we respond with `'none` and terminate. In order to keep the store service running, we could also change the type and recurse. With that change, though, a process using a store could never terminate, because the store it uses can never terminate. So we would have to add another option to the interface to deallocate the store. However, due to linearity, this could only succeed if the store is in fact empty, so it seems better to integrate this into the interaction after a deletion.

For a node process, we create a new node s' with the prior value x and then become (through a tail call) a node holding the new value y. The external interface remains the channel s. We'll walk through this code practicing *type-directed programming*: letting the types inform us about our choices. Linearity is an even bigger help here than types are in functional programming already because they further constrain our choices.

We begin with just the header, showing the typing judgment in effect for the body of the node.

```
proc node (s : store) (x : bin) (t : store) =
     % (x : bin) (t : store) |- (s : store)
```

At this point we can't do anything useful with x or t. On the other hand, the type of s is an external choice between labels ins and del, so the client will send one of these two messages. We write down the two branches and note the typing in the first branch. Note that the type of s has advanced from store to bin $\multimap$ store due to the receipt of the ins label.

```
proc node (s : store) (x : bin) (t : store) =
   recv s ( 'ins => ... % (x : bin) (t : store) |- (s : bin -o store)
          | 'del => ... )
```

From the type of s we can see it is still the client's turn, this time to send us a channel (say y) of type bin. After receiving y we own channel y (in addition to x and t), and the type of s has cycled all the way around back to store.

```
proc node (s : store) (x : bin) (t : store) =
   recv s ( 'ins => recv s (y =>  ...
                    % (y : bin) (x : bin) (t : store) |- (s : store)
          | 'del => ... )
```

At this point it seems we have more than one option. We could insert x or y into t. Or we could create a new node holding x with tail t and then recur with y. Let's do the latter, because it looks like it might avoid a cascading sequent of inserts rippling to the end of the chain of nodes. The idiom allocates a new channel s' and spawns a new process mode providing this channel. Since we pass it x and t they disappear from the antecedents and are supplanted by s'.

```
proc node (s : store) (x : bin) (t : store) =
   recv s ( 'ins => recv s (y => s' <- call node s' x t ;
                    % (y : bin) (s' : store) |- (s : store)
          | 'del => ... )
```

At this point we can make a recursive call to *node*, providing s and holding the value y with tail s'.

```
proc node (s : store) (x : bin) (t : store) =
   recv s ( 'ins => recv s (y => s' <- call node s' x t ;
                                 call node s y s')
          | 'del => ... )
```

This takes care of the insert branch. In the delete branch we have the following type:

```
  proc node (s : store) (x : bin) (t : store) =
     recv s ( 'ins => recv s (y => s' <- call node s' x t ;
                                   call node s y s')
            | 'del =>
       % (x : bin) (t : store) |- (s : +{'none : 1, 'some : bin * store})
                ... )
```

Since we hold an element x we have to respond with the some label.

```
proc node (s : store) (x : bin) (t : store) =
     recv s ( 'ins => recv s (y => s' : store <- call node s' x t ;
                                   call node s y s')
            | 'del => send s 'some ;
                %  (x : bin) (t : store) |- (s : bin * store)
            )
```

Again, the positive type bin ⊗ store means we should send a channel of type bin along s—in this case the element we hold (x).

```
proc node (s : store) (x : bin) (t : store) =
     recv s ( 'ins => recv s (y => s' : store <- call node s' x t ;
                                   call node s y s')
            | 'del => send s 'some ;
                      send s x ;
                      % (t : store) |- (s : store)
            )
```

At this point we forward from t to s, since t represents the remainder of the stack.

```
proc node (s : store) (x : bin) (t : store) =
     recv s ( 'ins => recv s (y => s' : store <- call node s' x t ;
                                   call node s y s')
            | 'del => send s 'some ;
                      send s x ;
                      fwd s t )
```

We can convert a store back to a list simply by recursively deleting the elements until the store is empty. This time we just show the code at the end, but we recommend you walk through it similar to the way we did for the *node* process.

```
  proc store2list (l : list) (s : store) =
     send s 'del ;
     recv s ( 'none => recv s (() => send l 'nil ; send l ())
            | 'some => recv s (x => send l 'cons ; send l x ;
                       call store2list l s) )
```

You can find a few more examples of store/list programs in the file lecture6.mps.

9 A Brief Note on Parametricity

As a side remark, the *structural* version of the polymorphic type

$$\begin{array}{lll} \mathsf{store}_A & = \&\{ & \mathsf{ins} : A \to \mathsf{store}_A, \\ & & \mathsf{del} : +\{\,\mathsf{none} : \mathbf{1}, \mathsf{some} : A \times \mathsf{store}_A\,\}\,\} \end{array}$$

guarantees that the elements returned by the store are among the ones inserted into an empty store. This is an application of *parametricity* [Reynolds, 1983]. However, the store may drop or duplicate elements. The linear type

$$\begin{array}{lll} \mathsf{store}_A & = \&\{ & \mathsf{ins} : A \multimap \mathsf{store}_A, \\ & & \mathsf{del} : \oplus\{\,\mathsf{none} : \mathbf{1}, \mathsf{some} : A \otimes \mathsf{store}_A\,\}\,\} \end{array}$$

sharpens this: the elements returned must be a *permutation* of the elements inserted.

We also conjecture (but haven't proved) that in the case of ordered connectives we can guarantee the behavior of a queue or a stack.

10 Summary

We summarize the statics (typing rules) and dynamics (computation rules) of the MPASS language in Figure 1 and Figure 2 respectively.

References

Robin Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.

John C. Reynolds. Types, abstraction, and parametric polymorphism. In R.E.A. Mason, editor, *Information Processing 83*, pages 513–523. Elsevier, September 1983.

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

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

$$\frac{\Delta \vdash P :: (x : A_k) \quad (k \in L)}{\Delta \vdash \textbf{send}\ x\ k \;;\; P :: (x : \oplus\{\ell : A_\ell\}_{\ell \in L})}\ \oplus R$$

$$\frac{\Delta, x : A_\ell \vdash Q_\ell :: (z : C) \quad (\forall \ell \in L)}{\Delta, x : \oplus\{\ell : A_\ell\}_{\ell \in L} \vdash \textbf{recv}\ x\ (\ell \Rightarrow Q_\ell)_{\ell \in L} :: (z : C)}\ \oplus L$$

$$\frac{\Delta \vdash P :: (x : B)}{\Delta, w : A \vdash \textbf{send}\ x\ w \;;\; P :: (x : A \otimes B)}\ \otimes R^*$$

$$\frac{\Delta', y : A, x : B \vdash Q(y) :: (z : C)}{\Delta', x : A \otimes B \vdash \textbf{recv}\ x\ (y \Rightarrow Q(y)) :: (z : C)}\ \otimes L$$

$$\frac{\Delta \vdash P_\ell :: (x : A_\ell) \quad (\forall \ell \in L)}{\Delta \vdash \textbf{recv}\ x\ (\ell \Rightarrow P_\ell)_{\ell \in L} :: (x : \&\{\ell : A_\ell\}_{\ell \in L})}\ \& R$$

$$\frac{\Delta, x : A_k \vdash Q :: (z : C) \quad (k \in L)}{\Delta, x : \&\{\ell : A_\ell\}_{\ell \in L} \vdash \textbf{send}\ x\ k \;;\; Q :: (z : C)}\ \& L$$

$$\frac{\Delta, y : A \vdash P :: (x : B)}{\Delta \vdash \textbf{recv}\ x\ (y \Rightarrow P(y)) :: (x : A \multimap B)}\ \multimap R$$

$$\frac{\Delta', x : B \vdash Q :: (z : C)}{\Delta', w : A, x : A \multimap B \vdash \textbf{send}\ x\ w \;;\; Q :: (z : C)}\ \multimap L^*$$

Figure 1: Statics for MPASS

$$
\begin{array}{lcll}
\mathsf{proc}(x_A \leftarrow P(x) \;;\; Q(x)) & \longrightarrow & \mathsf{proc}(P(a)), \mathsf{proc}(Q(a)) & (a \text{ fresh}) \\
\mathsf{proc}(P(b)), \mathsf{proc}(\mathbf{fwd}\; a\; b) & \longrightarrow & \mathsf{proc}(P(a)) & \\
\mathsf{proc}(\mathbf{fwd}\; a\; b), \mathsf{proc}(Q(a)) & \longrightarrow & \mathsf{proc}(Q(b)) & \\
\mathsf{proc}(\mathbf{send}\; x\; k \;;\; P), \mathsf{proc}(\mathbf{recv}\; x\; (\ell \Rightarrow Q_\ell)_{\ell \in L}) & \longrightarrow & \mathsf{proc}(P), \mathsf{proc}(Q_k) & (k \in L) \\
\mathsf{proc}(\mathbf{send}\; x\; (\,)), \mathsf{proc}(\mathbf{recv}\; x\; ((\,) \Rightarrow Q)) & \longrightarrow & \mathsf{proc}(Q) & \\
\mathsf{proc}(\mathbf{send}\; a\; b \;;\; P), \mathsf{proc}(\mathbf{recv}\; a\; (y \Rightarrow Q(y))) & \longrightarrow & \mathsf{proc}(P), \mathsf{proc}(Q(b)) &
\end{array}
$$

Figure 2: Dynamics for MPASS