Lecture Notes on Preservation and Progress

15-836: Substructural Logics
Frank Pfenning

Lecture 7
September 19, 2023

1 Introduction

Our investigation has shown the close correspondences between linear propositions and session types, between sequent proofs and synchronous message-passing programs, and between cut reduction and communication. Despite these close connections, there are also differences. For example, permuting cut reductions and identity expansion are related to process equalities, but not directly to computation. Here are some other key differences:

Recursion. Recursion is a central concept in programming languages but much less prevalent in the study of logic. Nevertheless, there is a whole branch of logic dedicated to arithmetic, including induction and primitive recursion. See, for example, once again Gentzen's pioneering work [Gentzen, 1936]. Also, *infinitary proofs* have been studied—we'll see an example in Lecture 8 on *Subtyping*.

Observability. The primary purpose of proofs is to convince you that a proposition is true and explain why. As such, the whole proof must be subject to inspection so we can check it and also understand it. The primary purpose of programs is computation. As such, we are mostly interested in observing its outcome (assuming that maybe we have separately verified or trust in its correctness). But we do not observe functions directly, only their results on particular inputs. This gives the provider of a library the freedom to change function definitions (e.g., improve their efficiency) without changing the observable input/output behavior.

After motivating our MPASS language through logic and proof theory, we will see the differences that arise when we put our programmers' hats on. In particular, the theorems we prove about the logic and the theorems we prove about our linear message-passing programming language will by necessity be different. Nevertheless, we can see how the effort we invested in proof of cut elimination does not go to waste; it is just that the key insights appear in different contexts.

From the foregoing discussion it might appear that logic and programming and in fact two different subjects, albeit with close connections. In my view they are in fact synthesized and generalized in *constructive type theory*. On one side, type theory generalizes logic by providing the intrinsic ability to talk about its own proofs. On the other side, type theory generalizes programming languages by providing the intrinsic ability to reason about their correctness. I don't know how much opportunity we will have to explore *substructural type theory* (in this sense) in this course. Many questions here are not yet well understood.

2 Integrating Recursion

In the last two lectures and the MPass examples we have seen that recursion is introduced in two ways, both coming down to *definitions*:

(1) *Types* may be defined recursively. For example,

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

Such types are *equirecursive* in the sense that recursion at the type level is not associated with any messages. During type-checking we are permitted to silently replace a type name such as nat with its definition. In Lecture 8 we explore the algorithmic consequences of this decision.

(2) *Processes* may be defined recursively. For example, the successor process on binary numbers required recursion in order to represent the carry bit.

These two go hand-in-hand: often the recursive structure of processes is dictated by the recursive structure of the types they operate on.

Based on these observation we integrate recursion into our programming language via a *signature* Σ containing definitions at the type and process level. We write t and s for type names, p for process names, and $\overline{(y : B)}$ for a sequence of parameters $y_i : B_i$.

$$\text{Signature} \quad \Sigma \quad ::= \quad \cdot \mid \Sigma, t = A \mid \Sigma, p\ (x : A)\ \overline{(y : B)} = P$$

Because we want to make mutual recursion as natural as possible, the individual declarations in a signature Σ are checked for correctness against the whole signature rather than the usual left-to-right manner. We write $\vdash \Sigma\ \mathit{sig}$ to mean that Σ is a valid signature, and $\vdash_\Sigma \Sigma'\ \mathit{sig}$ to mean that all declarations in Σ' are valid in the signature Σ. We have the following rules:

$$\frac{}{\vdash_\Sigma (\cdot)\ \mathit{sig}} \qquad \frac{\vdash_\Sigma \Sigma'\ \mathit{sig} \quad \vdash_\Sigma A\ \mathit{type}}{\vdash_\Sigma (\Sigma', t = A)\ \mathit{sig}} \qquad \frac{\vdash_\Sigma \Sigma'\ \mathit{sig} \quad \overline{(y : B)} \vdash_\Sigma P(x, \overline{y}) :: (x : A)}{\vdash_\Sigma (\Sigma', p\ (x : A)\ \overline{(y : B)} = P(x, \overline{y})\ \mathit{sig}}$$

In these rules the index Σ in $\vdash_\Sigma$ never changes: the signature is in a sense global. Therefore we omit it from all the judgments and imagine that in any given situation it will be fixed and valid. Furthermore, all type names and process names in a signature must be distinct.

The judgment $\vdash_\Sigma A\ \mathit{type}$ means that A is (or can be implicitly expanded to) one of the types in our language, and that all the type names in A are defined in Σ.

Computationally, a **call** simply expands to its definition.

$$\mathsf{proc}(\mathbf{call}\ p\ a\ \overline{b}) \quad\longrightarrow\quad \mathsf{proc}(P(a,\overline{b})) \quad \text{where } p\ (x : A)\ \overline{(y : B)} = P(x,\overline{y}) \in \Sigma$$

3 Typing Configurations of Processes

When running a program, we imagine starting with a single process P. During the computation, many new processes may be spawned and interact with each other. We think of these processes of defining a *multiset* in the sense of linear inference. We can define it more syntactically with the following.

$$\text{Configuration} \quad \mathcal{C} \quad ::= \quad \mathsf{proc}(P) \mid \mathcal{C}_1, \mathcal{C}_2 \mid \cdot$$

Here, the comma operator is associative and commutative with the empty configuration $(\cdot)$ as its unit.

There are many meaningless configurations, such as one where one process sends a label that the recipient does not expect, or one where a process send unit while the recipient expects a channel. Undoubtedly, while programming in MPASS you have encountered such incorrect processes, which would have become incorrect configurations when running.

How do we ensure configurations are meaningful, which is to say, they are well-typed? For a single process, our judgment is $\Delta \vdash P :: (x : A)$ which means that P provides x at type A and uses the channels in Δ at their given types. A configuration may consist of multiple processes, so this generalizes to

$$\Delta \vdash \mathcal{C} :: \Delta'$$

which means the configuration $\mathcal{C}$ *uses* (is a client to) all the channel in Δ and provides all the channels in Δ'. We might at first hypothesize the following rule:

$$\frac{\Delta \vdash P :: (a : A)}{\Delta \vdash \mathsf{proc}(P) :: (a : A)}\ \mathsf{proc?}$$

Here, we take advantage of the fact that channels a, b, c behave exactly the same under typing as variables x, y, z so that the typing judgment in the premise is well-defined. But this is not quite sufficient: There may be *other* channels among the antecedents to $\mathsf{proc}(P)$ that are not used by P. These will still be available to clients of this configuration. So we get

$$\frac{\Delta \vdash P :: (a : A)}{\Delta', \Delta \vdash \mathsf{proc}(P) :: (\Delta', a : A)}\ \mathsf{proc}$$

For process terms, cut requires that the provider and the client agree on the type of the private channel that enables communication between them. For configurations, this has to hold for *all* channels since they are all derived from cut. The empty context just passes through all channels provided to it, since it neither uses nor provides any channels of its own.

$$\frac{\Delta_0 \vdash \mathcal{C}_1 :: \Delta_1 \quad \Delta_1 \vdash \mathcal{C}_2 :: \Delta_2}{\Delta_0 \vdash \mathcal{C}_1, \mathcal{C}_2 :: \Delta_2}\ \mathsf{join} \qquad \frac{}{\Delta \vdash (\cdot) :: \Delta}\ \mathsf{empty}$$

At this point we notice an emerging conflict. On one hand we think of configurations as multisets in the sense that the order of the individual processes is relevant. On the other, for a typing derivation we require some ordering. As we can see from the join rule, the typing derivation requires that each provider precedes its client. Furthermore, we need to make sure that this relation is uniquely determined so we stipulate that each channel in a configuration with

$$\Delta \vdash \mathcal{C} :: \Delta'$$

occurs either in Δ, or in Δ' (or both), or has exactly one provider and exactly one client in $\mathcal{C}$. When we start from an initial configuration with a single main process, this will be true, and all the rules can be seen to preserve this property. The only doubt one might have is about cut, but the new channel is chosen such that it does not already occur in the whole configuration. Moreover, this new channel has exactly one provider and exactly one client.

Coming back to the ordering, the join operator as combining two *derivations* is associative with unit empty, but it is not commutative. For a configuration to be well-typed we require that there is an ordering of the processes that can be typed with the given rules. This ordering is not unique: satisfying the provider-before-client requirement still may leave many options. Which of these possibilities we pick is irrelevant. A key property only briefly mentioned in lecture is the following exchange lemma.

Lemma 1 (Exchange) *If process P provides a channel a which is not used by the following process Q in the configuration typing, then the two processes can be exchanged.*

Proof: By inspection of the two typing derivations, since the first represents the most general case of two consecutive processes satisfying the given condition. Given

$$\dfrac{\dfrac{\Delta_P \vdash P :: (a : A)}{\Delta', \Delta_P \vdash \mathsf{proc}(P) :: (\Delta', a : A)}\ \mathsf{proc} \quad \dfrac{\Delta_Q \vdash Q :: (c : C)}{\Delta', a : A, \Delta_Q \vdash \mathsf{proc}(Q) :: (\Delta', a : A, c : C)}\ \mathsf{proc}}{\Delta', \Delta_P, \Delta_Q \vdash \mathsf{proc}(P), \mathsf{proc}(Q) :: (\Delta', a : A, c : C)}\ \mathsf{join}$$

we construct

$$\dfrac{\dfrac{\Delta_Q \vdash Q :: (c : C)}{\Delta', \Delta_P, \Delta_Q \vdash \mathsf{proc}(Q) :: (\Delta', \Delta_P, c : C)}\ \mathsf{proc} \quad \dfrac{\Delta_P \vdash P :: (a : A)}{\Delta', \Delta_P, c : A \vdash \mathsf{proc}(P) :: (\Delta', a : A, c : C)}\ \mathsf{proc}}{\Delta', \Delta_P, \Delta_Q \vdash \mathsf{proc}(Q), \mathsf{proc}(P) :: (\Delta', a : A, c : C)}\ \mathsf{join}$$

□

We can iterate the exchange so that a process P that provides a channel a can always be moved to the right until it is next to its client. If it does not have a client, then it can be moved to be the rightmost process in a configuration.

4 Preservation

Unlike pure logic where cut elimination always terminates in a cut-free proof, computation may run forever due to the presence of recursion. So rather than cut elimination (or admissibility of cut as the key lemma) we prove that as computation proceeds the configuration remains well-typed. Since the computation rules are (mostly) derived from principal cut reductions, patterns from the proof of the admissibility of cut recur.

We have already remarked on a fundamental property, namely that the type of channels evolves as communication takes place. So it what sense are types actually preserved? What happens is that the types of *internal channels* in a configuration changes consistently between client and provider, but the types of *externally visible channels* remain invariant.

Theorem 2 (Preservation) *If $\Delta \vdash \mathcal{C} :: \Delta'$ and $\mathcal{C} \longrightarrow \mathcal{D}$ then $\Delta \vdash \mathcal{D} :: \Delta'$.*

Theorem 3 *The proof proceeds by cases over the forms of the reduction. There are four kinds of cases: spawn (= cut), fwd (= identity), call, and interactions. We show two representative cases. We analyze the configuration in the order of its typing derivation.*

Cut: $\mathcal{C} = (\mathcal{C}_L, \mathsf{proc}(x_A \leftarrow P(x) \;;\; Q(x)), \mathcal{C}_R)$ *where*

$$\begin{array}{l} \Delta \vdash \mathcal{C}_L :: \Delta_L \\ \Delta_L \vdash \mathsf{proc}(x_A \leftarrow P(x) \;;\; Q(x)) :: \Delta_R \\ \Delta_R \vdash \mathcal{C}_R :: \Delta' \end{array}$$

and

$$\mathcal{D} = (\mathcal{C}_L, \mathsf{proc}(P(a)), \mathsf{proc}(Q(a)), \mathcal{C}_R)$$

for a globally fresh channel a.

In order to construct a typing derivation for $\mathcal{D}$ as required, we apply inversion *to the typing*

$$\Delta_L \vdash \mathsf{proc}(x_A \leftarrow P(x) \;;\; Q(x)) :: \Delta_R$$

This means we analyze the typing rules for processes and consider what we might say about this typing derivation. We find that for some Δ'_L, Δ_P, Δ_Q, and $c : C$, we must have

$$\begin{array}{l}\Delta_L = (\Delta'_L, \Delta_P, \Delta_Q) \\ \Delta_P \vdash P(x) :: (x : A) \\ \Delta_Q, x : A \vdash Q(x) :: (c : C) \\ \Delta_R = (\Delta'_L, c : C)\end{array}$$

Because a is globally fresh, we can substitute it for x in the two typing derivation in the middle, and conclude that

$$\begin{array}{l}\Delta_L = (\Delta'_L, \Delta_P, \Delta_Q) \\ \Delta_P \vdash P(a) :: (a : A) \\ \Delta_Q, a : A \vdash Q(a) :: (c : C) \\ \Delta_R = (\Delta'_L, c : C)\end{array}$$

Now we can construct a typing derivation for $\mathcal{D} = (\mathcal{C}_L, \mathsf{proc}(P(a)), \mathsf{proc}(Q(a)), \mathcal{C}_R)$ from

$$\begin{array}{l}\Delta \vdash \mathcal{C}_L :: \Delta_L \\ \Delta_L = (\Delta'_L, \Delta_P, \Delta_Q) \\ \Delta'_L, \Delta_P, \Delta_Q \vdash \mathsf{proc}(P(a)) :: (\Delta'_L, \Delta_Q, a : A) \\ \Delta'_L, \Delta_Q, a : A \vdash \mathsf{proc}(Q(a)) :: (\Delta'_L, c : C) \\ (\Delta'_L, c : C) = \Delta_R \\ \Delta_R \vdash \mathcal{C}_R :: \Delta'\end{array}$$

This concludes this case of type preservation.

$\otimes R/\otimes L$: $\mathcal{C} = (\mathcal{C}_L, \mathsf{proc}(\mathbf{send}\ a\ b\ ;\ P), \mathsf{proc}(\mathbf{recv}\ a\ (y \Rightarrow Q(y))), \mathcal{C}_R)$ *where*

$$\begin{array}{l}\Delta \vdash \mathcal{C}_L :: \Delta_L \\ \Delta_L \vdash \mathsf{proc}(\mathbf{send}\ a\ b\ ;\ P), \mathsf{proc}(\mathbf{recv}\ a\ (y \Rightarrow Q(y))) :: \Delta_R \\ \Delta_R \vdash \mathcal{C}_R :: \Delta\end{array}$$

and

$$\mathcal{D} = (\mathcal{C}_L, \mathsf{proc}(P), \mathsf{proc}(Q(b)), \mathcal{C}_R)$$

Here we have taken advantage the exchange lemma to restrict ourselves to the case where the provider and client are immediately adjacent in the typing derivation. Again we apply inversion *to analyze the possible typing derivations for the middle line and find that for some Δ'_L, Δ_P, $a : A$, $b : B$, and $c : C$ we must have*

$$\begin{array}{l}\Delta_L = (\Delta'_L, \Delta_P, b : B, \Delta_Q) \\ \Delta_P \vdash P :: (a : A) \\ \Delta_Q, a : A, b : B \vdash Q(b) :: (c : C) \\ \Delta_R = (\Delta'_L, c : C)\end{array}$$

A critical step here is examine the typing rules for $\mathbf{send}\ a\ b\ ;\ P$ *and* $\mathbf{recv}\ a\ (y \Rightarrow Q(y))$ *for which there is only one each once we know the first process is the provider (which comes from their relative position in the typing derivation for* $\mathcal{C}$*)*

From these pieces we can assemble a typing derivation for

$$\mathcal{D} = (\mathcal{C}_L, \mathsf{proc}(P), \mathsf{proc}(Q(b)), \mathcal{C}_R)$$

as follows:

$$\begin{array}{l}
\Delta \vdash \mathcal{C}_L :: \Delta_L \\
\Delta_L = (\Delta'_L, \Delta_P, b : B, \Delta_Q) \\
\Delta'_L, \Delta_P, b : B, \Delta_Q \vdash \mathsf{proc}(P) :: (\Delta'_L, a : A, b : B, \Delta_Q) \\
\Delta'_L, a : A, b : B, \Delta_Q \vdash Q(b) :: (\Delta'_L, c : C) \\
(\Delta'_L, c : C) = \Delta_R \\
\Delta_R \vdash \mathcal{C}_R :: \Delta'
\end{array}$$

There is a lot of bureaucracy in the proof of preservation, but ultimately the core reasoning step in the communication steps is that cut reduction preserves the conclusion of the cut (in particular, the antecedents and the succedent).

5 Progress

Preservation means that in any computation $\mathcal{C}_1 \longrightarrow \mathcal{C}_2 \longrightarrow \cdots$ the interface to the configuration never changes. Cut elimination would also predict that reduction always terminates, but that's not true in the presence of recursion unless we make some restrictions. Instead, we would like to prove that "we never get stuck": either we can take a step, or the configuration is *final* in a well-defined way. We are looking for analogue to the statement that in functional languages every expression e either can take a step or it is a value already. But what's the analogue of value? In our language of *synchronous communication* (that is, both sender and receiver proceed in lock-step when a message is exchanged) a configuration is *final* if all processes attempt to communicate along an external channel. Such a channel does not have a second endpoint, so such a process can legitimately not make further progress.

To keep the argument simple we assume that the configuration is closed on the left, that is,

$$\cdot \vdash \mathcal{C} :: \Delta$$

In other words, $\mathcal{C}$ *provides* some external channels but does not use any. That's analogous to the usual assumption that in a functional language we only evaluate *closed* expressions, that is, expressions without free variables.

Theorem 4 (Progress) *If* $\cdot \vdash \mathcal{C} :: \Delta$ *then either* $\mathcal{C}$ *is final or* $\mathcal{C} \longrightarrow \mathcal{D}$ *for some* $\mathcal{D}$*.*

Proof: This time we do a right-to-left induction over the structure the given typing derivation (which we associate to the left). So $\mathcal{C} = (\mathcal{C}_L, \mathsf{proc}(P))$ for some process P with $\cdot \vdash \mathcal{C}_L :: \Delta'$ and $\Delta' \vdash \mathsf{proc}(P) :: \Delta$.

By induction hypothesis, either $\mathcal{C}_L \longrightarrow \mathcal{D}_L$ for some $\mathcal{D}_L$ or $\mathcal{C}_L$ is final.

In the first case $\mathcal{C} \longrightarrow (\mathcal{D}_L, \mathsf{proc}(P))$ by definition of reduction.

In the second case, all processes in $\mathcal{C}_L$ will try to communicate along the channel that they provide. Now we distinguish cases based on the process P.

Cut/Spawn: $P = (x_A \leftarrow P_1(x)\ ;\ P_2(x))$ for some P_1 and P_2. Then $\mathsf{proc}(P) \longrightarrow \mathsf{proc}(P_1(a)), \mathsf{proc}(P_2(a))$ for a fresh a, and therefore also $\mathcal{C} \longrightarrow (\mathcal{C}_L, \mathsf{proc}(P_1), \mathsf{proc}(P_2))$.

Receive Channel: $P = (\mathbf{recv}\ a\ (y \Rightarrow P'(y)))$ for some P'. If P provides a (that is, $a : A \in \Delta$ for some A) then all of $\mathcal{C}$ is final.

Otherwise P uses a and must (by inversion) end in the $\otimes L$ rule. That is the typing derivation of $\Delta' \vdash \mathsf{proc}(P) :: \Delta$ looks like

$$\dfrac{\dfrac{\Delta_P, a : A, y : B \vdash P'(y) :: (c : C)}{\Delta_P, a : B \otimes A \vdash (\mathbf{recv}\ a\ (y \Rightarrow P'(y))) :: (c : C)}\ \otimes L}{\Delta' \vdash \mathsf{proc}(P) :: \Delta}\ \mathsf{proc}$$

where $\Delta' = (\Delta_L, \Delta_P, a : B \otimes A)$ and $\Delta = (\Delta_L, c : C)$

Because $\mathcal{C}_L :: (\Delta_L, \Delta_P, a : B \otimes A)$ there must be a process in $\mathcal{C}_L$ providing $a : B \otimes A$. In particular, it cannot be part of the antecedents of $\mathcal{C}_L$ because these must be empty.

By inversion on the typing of a we find that there must be an object $\mathsf{proc}(Q)$ in $\mathcal{C}_L$ that provides $a : B \otimes A$. Moreover, since $\mathcal{C}_L$ is final, this process must be trying communicate along a, so it must have form $Q = (\mathbf{send}\ a\ b\ ;\ Q')$. By the rule for sending and receiving a channel Q and P can interact, and therefore $\mathcal{C} \longrightarrow \mathcal{D}$ for some $\mathcal{D}$.

□

Again, there is a lot of bureaucracy, but in the end the progress theorem comes down to the fact that during the proof of admissibility of cut all the principal cases could be reduced (= make progress).

6 Observation

We think of a closed configuration $\cdot \vdash \mathcal{C} :: \Delta$ as a collection of processes that provide all the channels in Δ. As we have seen, the external interface Δ will never change during the computation. Moreover, when $\mathcal{C}$ is final, every process in $\mathcal{C}$ is trying to communicate along a channel in Δ. Because our language is synchronous (both send and receive block), this means none of the processes in $\mathcal{C}$ can take a step.

The question is how do we observe the outcome of the computation? Unlike functional languages, the value is not presented to us a whole. For example, if we have a final configuration

$$\cdot \vdash \mathsf{proc}(P) :: (a : \mathsf{nat})$$

where $\mathsf{nat} = \oplus\{\mathsf{zero} : \mathbf{1}, \mathsf{succ} : \mathsf{nat}\}$, then we do not yet have any information except that the process P terminated.

So we need to *receive* a label from the channel a in order to observe the outcome. As soon as we interact with P, it will resume computation with its continuation until is once again blocks with a send.

An observation may actually change the type of the channel a at the interface. For example, if we received zero along a then afterwards we have $P' :: (a : \mathbf{1})$. If instead we received succ, then afterwards the continuation process P' will again provide $a : \mathsf{nat}$.

A similar interaction protocol holds for all positive types ($A \otimes B$, $\mathbf{1}$, $\oplus\{\ell : A_\ell\}_{\ell \in L}$). For a negative type like $\&\{\ell : A_\ell\}_{\ell \in L}$ the situation is different. As pointed out in the introduction, we cannot actually observe the process that is trying to receive along the channel it provides. The best thing we could do at this point is send it (separately) each of the labels ℓ in the set L and observe the continuation A_ℓ in each case. This strategy breaks down when we encounter $a : B \multimap A$ because we cannot possibly send it a channel B that explores all possible behaviors along a. For example, if $B = \mathsf{nat}$, there would be infinitely many.

This means when we encounter a channel of negative type we stop our observation process. An analogous decision is made in functional languages such as ML or Haskell: values of function types are simply not directly observable, although we can probe their behavior by applying them to different arguments.

The implementation of the `exec P` in MPASS observes the outcome of a computation just as described above and prints the observed messages. When a negative type is encountered it prints just a dash.

7 Refactoring the Dynamics

It is often convenient to treat all the send and receive actions in a uniform way. In order to support this, we can refactor the syntax and also the dynamics with the following definitions.

$$\begin{array}{llcll}
\text{Processes} & P, Q & ::= & x_A \leftarrow P(x) \;;\; Q(x) & (\mathsf{cut}) \\
 & & \mid & \mathbf{fwd}\; x\; y & (\mathsf{id}) \\
 & & \mid & \mathbf{send}\; x\; m \;;\; P & \text{(positive right or negative left rules)} \\
 & & \mid & \mathbf{recv}\; x\; K & \text{(positive left or negative right rules)} \\
 & & \mid & \mathbf{call}\; p\; x\; \overline{y} & \text{(possibly recursive process } p) \\[1ex]
\text{Messages} & m & ::= & (\,) & (\mathbf{1}) \\
 & & \mid & k & (\oplus, \&) \\
 & & \mid & y & (\otimes, \multimap) \\[1ex]
\text{Continuations} & K & ::= & (\,) \Rightarrow P & (\mathbf{1}) \\
 & & \mid & (\ell \Rightarrow P_\ell)_{\ell \in L} & (\oplus, \&) \\
 & & \mid & (y \Rightarrow P(y)) & (\otimes, \multimap)
\end{array}$$

In the dynamics, we pass a message to a continuation $m \triangleright K$ to obtain a process.

$$\begin{array}{lclcll}
(\,) & \triangleright & ((\,) \Rightarrow P) & = & P & \\
k & \triangleright & (\ell \Rightarrow P_\ell)_{\ell \in L} & = & P_k & (k \in L) \\
b & \triangleright & (y \Rightarrow P(y)) & = & P(b) &
\end{array}$$

The computation rules then simplify.

$$\begin{array}{lcl}
\mathsf{proc}(x_A \leftarrow P(x) \;;\; Q(x)) & \longrightarrow & \mathsf{proc}(P(a)), \mathsf{proc}(Q(a)) \quad (a \text{ fresh}) \\
\mathsf{proc}(P(b)), \mathsf{proc}(\mathbf{fwd}\; a\; b) & \longrightarrow & \mathsf{proc}(P(a)) \\
\mathsf{proc}(\mathbf{send}\; a\; m \;;\; P), \mathsf{proc}(\mathbf{recv}\; a\; K) & \longrightarrow & \mathsf{proc}(P), \mathsf{proc}(m \triangleright K) \\
\mathsf{proc}(\mathbf{call}\; p\; a\; \overline{b}) & \longrightarrow & \mathsf{proc}(P(a, \overline{b})) \\
 & & \text{where } p\; (x : A)\; \overline{(y : B)} = P(x, \overline{y}) \in \Sigma
\end{array}$$

There are some possible variations on the identity rules that are sometimes useful. For an implementation, for example, we might enforce that $P(b)$ actually tries to communicate along b so it is expecting to interact. There is also a symmetric rule to the given one where $\mathsf{proc}(\mathbf{fwd}\; a\; b)$ interacts with its client $\mathsf{proc}(P(a))$ to yield $\mathsf{proc}(P(b))$. Such variations are consistent with logical cut reduction but we are not forced to specialize or generalize the rule above.

References

Gerhard Gentzen. Die Widerspuchsfreiheit der reinen Zahlentheorie. *Mathematische Annalen*, 112:493–565, 1936. English translation in M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 132–213, North-Holland, 1969.