Invited talk at TLDI'12

Towards Concurrent Type Theory

Luís Caires
Universidade Nova de Lisboa
luis.caires@di.fct.unl.pt

Frank Pfenning
Carnegie Mellon University
fp@cs.cmu.edu

Bernardo Toninho
Carnegie Mellon University &
Universidade Nova de Lisboa
btoninho@cs.cmu.edu

Abstract

We review progress in a recent line of research that provides a concurrent computational interpretation of (intuitionistic) linear logic. Propositions are interpreted as session types, sequent proofs as processes in the π-calculus, cut reductions as process reductions, and vice versa. The strong proof-theoretic foundation of this type system provides immediate opportunities for uniform generalization, specifically, to embed terms from a functional type theory. The resulting system satisfies the properties of type preservation, progress, and termination, as expected from a language derived via a Curry-Howard isomorphism. While very expressive, the language is strictly stratified so that dependent types for functional terms can be enforced during communication, but neither processes nor channels can appear in functional terms. We briefly speculate on how this limitation might be overcome to arrive at a fully dependent concurrent type theory.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory

General Terms languages, theory, verification

Keywords type theory, dependent types, session types, π-calculus

1. Introduction

Church and Rosser [1936] developed the λ-calculus as a pure calculus of functional computation. As such it has been tremendously successful and has remained essentially unchanged since its inception. Yet if viewed as a logic it is inconsistent, so Church [1940] turned to types. His *simply-typed λ-calculus* supports a higher-order logic, suitable for the formalization of much of classical mathematics. Both syntax and computational mechanisms of the original untyped λ-calculus and its simply-typed version are the same, confirming its fundamental nature. Howard [1969] established a tight correspondence between proofs in natural deduction and Church's simply-typed λ-calculus.[1] But the analogy does not end there. Proof reductions as defined by Prawitz [1965] correspond to the computational reductions of the typed λ-calculus. This isomorphism has been the source of an incredibly rich variety of type structures that simultaneously have a *logical* and an *operational* meaning. One important benefit is that a language designed around logical principles provides intrinsic means for reasoning about its programs, an idea crystalized in constructive type theory [Martin-Löf 1980].

At present, there is no similarly compelling story for *concurrent* computation. Milner et al. [1992] developed the π-calculus as an untyped, foundational calculus for concurrent computation analogous to the untyped λ-calculus for functional computation. But defining a typed version intimately related to logic via a propositions-as-types/proofs-as-programs interpretation has been elusive. In a series of recent papers [Caires and Pfenning 2010; Toninho et al. 2011a; Pfenning et al. 2011; Toninho et al. 2011b; Pérez et al. 2011], we have explored just such a connection from a variety of angles. In this paper, we provide an introduction to our approach and discuss where it succeeds, and where it still falls short of the goal of a constructive concurrent type theory.

Given the line of research on linear logic [Girard 1987], the geometry of interaction [Girard 1989], and game semantics [Hyland and Ong 1995], it should not come as a surprise that linear logic plays a crucial role in our endeavor. Unlike in functional programming, the isomorphism is not with natural deduction but with sequent calculus. This is because we need to decompose substitution, the workhorse of the λ-calculus, into the name-passing interactions of the π-calculus. Like the step from the untyped to the typed λ-calculus, we have to be prepared to give up a lot, such as the possibility to express processes that may deadlock or have nonterminating interactions. As in functional programming, some expressive power can be recovered in a principled way in a full type theory using inductive and coinductive types, but we will not address practical language design here. It turns out that a concurrent interpretation of linear sequent calculus provides a logical reconstruction of *session types* [Honda 1993; Honda et al. 1998]. As in functional programming, a logical foundation provides an immediate and rich opportunity for extension and generalization. One of these directions points towards type theories in which functional and concurrent computation can co-exist harmoniously.

2. Judgmental Principles

In a functional setting, the basic judgment is generally of the form $M : A$, meaning either that M is a proof of A, or M is a term of type A. In the setting of communicating processes, it is unclear what it would mean to say that "*P is a process of type A*". Processes communicate with their environment, so we write instead $P :: x : A$, meaning P is a process offering service A along channel x. The channel here is considered a variable with scope P, so we can rename it as $P\{y/x\} :: y : A$ if y is not already used in P. As is customary, we will perform such renamings silently as appropriate.

[1] A related correspondence between combinators and axiomatic proofs had been noted earlier by Curry [1934], whence the name "Curry-Howard isomorphism".

Processes are uninteresting unless they are placed in a context where they not only *offer* services, but also *use* services offered by other processes. We write a *sequent*

$$x_1{:}A_1, \ldots, x_n{:}A_n \Rightarrow P :: x : A$$

to express that process P offers service A along x when composed with processes P_i providing A_i along x_i for $1 \leq i \leq n$. All the variables x_i must be distinct. We denote contexts of the form above by Δ.

Offering and using services are counterparts, but they are not the same. Therefore, formally, the judgment $x{:}A$ on the right of the sequent and the judgment $x_i{:}A_i$ on the left should be considered different. Since we can always tell by position which one is meant, we use the same notation for both.

Processes evolve through interactions along channels. Interacting on a channel x_i therefore engenders a change of state, and the same channel cannot be used again with the same type. Therefore the sequent arrow "$\Rightarrow$" denotes a *linear hypothetical judgment* [Chang et al. 2003] where each antecedent must be used exactly once. Therefore the context is not subject to weakening or contraction, but reordering is permitted since antecedents are identified by unique names.

Even without defining any particular kinds of services, some principles should hold for the judgments in general. We discuss these first, because they are an important guide to the rest of the development.

2.1 Cut as Composition

When a process P *offers* service A along x, and another process Q *uses* a service A along x, the two can be composed so that they communicate along x.

$$\frac{\Delta \Rightarrow P :: x : A \quad \Delta', x{:}A \Rightarrow Q :: z : C}{\Delta, \Delta' \Rightarrow (\boldsymbol{\nu} x)(P \mid Q) :: z : C}\ \mathsf{cut}$$

The process expression for the composition puts P and Q in parallel, sharing x as a *private* channel, as indicated by the *name restriction* $(\boldsymbol{\nu} x)$. Note that this rule entails some implicit renaming, because the channel along which P offers A must be equated with the channel along which Q uses A.

Viewed from the purely logical perspective, this is simply the rule of *cut* in linear logic:

$$\frac{\Delta \Rightarrow A \quad \Delta', A \Rightarrow C}{\Delta, \Delta' \Rightarrow C}\ \mathsf{cut}$$

It is a little unusual that we use an intuitionistic version of linear logic [Barber 1997] with a singleton right-hand side in a sequent formulation. This is not absolutely essential (see Abramsky [1993] for a related classical counterpart), but it streamlines the judgmental justification of the system. It also reflects an intrinsic asymmetry between offering and using a service, even though we will see they are strongly related. It will also be beneficial in developing a dependent type theory. Finally, as discussed briefly in prior work [Caires and Pfenning 2010] it enforces locality of shared names [Merro and Sangiorgi 2004] as desired for our process interpretation.

2.2 Identity as Forwarding

One way to fulfill the promise of A along x is to just use a channel y that in turn promises A. This just corresponds to an initial sequent, which we call the identity rule.

$$\frac{}{y{:}A \Rightarrow [y \leftrightarrow x] :: x : A}\ \mathsf{id}$$

There is no standard notation for forwarding in the π-calculus, so we write $[y \leftrightarrow x]$ to forward between y and x. We can *implement* the forwarding behavior in the untyped calculus. For example, if service A were to require an input along x, we could define $[y \leftrightarrow x] = x(w).y\langle w\rangle.\mathbf{0}$. In order to make the correspondence between proofs and processes as direct as possible, we prefer to use the new notation.

In accordance with the linear hypothetical judgment, the antecedent $y{:}A$ must be the only one. In purely logical form:

$$\frac{}{A \Rightarrow A}\ \mathsf{id}$$

In summary, cut and identity connect offers and uses of services. Cut composes a process P that offers A along x with a process that uses A along x, making x a private channel $(\boldsymbol{\nu} x)(P \mid Q)$. Identity uses a channel y supplying A to satisfy its own offer of x along A, $[y \leftrightarrow x]$. These general principles are not connected to any particular types of services, which will be associated with particular logical connectives.

2.3 Composing Cut and Identity

Cut and identity are two ways to relate the judgments on the left and right of a sequent. They should be inverses of each other in the sense that composition with the identity should have no effect. We show the compositions directly with the process terms, where $\{y/x\}$ is our notation for capture-avoiding name substitution.

$$\frac{\dfrac{}{y{:}A \Rightarrow [y \leftrightarrow x] :: x : A}\ \mathsf{id} \quad \Delta, x{:}A \Rightarrow Q :: z : C}{\Delta, y{:}A \Rightarrow (\boldsymbol{\nu} x)([y \leftrightarrow x] \mid Q) :: z : C}\ \mathsf{cut}$$

$$\longrightarrow$$

$$\Delta, y{:}A \Rightarrow Q\{y/x\} :: z : C$$

$$\frac{\Delta \Rightarrow P :: x : A \quad x{:}A \Rightarrow [x \leftrightarrow z] :: z : A}{\Delta \Rightarrow (\boldsymbol{\nu} x)(P \mid [x \leftrightarrow z]) :: z : A}\ \mathsf{cut}$$

$$\longrightarrow$$

$$\Delta \Rightarrow P\{z/x\} :: z : A$$

So composition with identity results in renaming. Logically, this transformation takes place at the level of the judgments: no particular propositions are involved. They are therefore of a somewhat different character than the computational reductions in the remainder and might be considered "structural reductions". If we don't have the exact typing premises, we need some conditions on free names of processes, denoted by $\mathsf{fn}(-)$.

$$\begin{array}{lcll} (\boldsymbol{\nu} x)([y \leftrightarrow x] \mid Q) & \longrightarrow & Q\{y/x\} & y \notin \mathsf{fn}(Q) \\ (\boldsymbol{\nu} x)(P \mid [x \leftrightarrow z]) & \longrightarrow & P\{z/x\} & z \notin \mathsf{fn}(P) \end{array}$$

2.4 Composing Cuts

Multiple parallel compositions should be required to synchronize only to the extent that their interactions require it. Proof-theoretically, this means that the order of consecutive cuts should be insignificant. The corresponding laws have no inherent orientation as rewrite rules, so we think of them as structural proof equivalences. We leave it to the reader to write out the simple proof figures. On the process calculus we obtain a corresponding structural equivalences.

$$\begin{array}{l} (\boldsymbol{\nu} x)((\boldsymbol{\nu} y)(P \mid Q) \mid R) \equiv (\boldsymbol{\nu} y)(P \mid (\boldsymbol{\nu} x)(Q \mid R)) \\ \qquad\qquad \text{provided } x \notin \mathsf{fn}(P),\ y \notin \mathsf{fn}(R) \\ (\boldsymbol{\nu} x)(P \mid (\boldsymbol{\nu} y)(Q \mid R)) \equiv (\boldsymbol{\nu} y)(Q \mid (\boldsymbol{\nu} x)(P \mid R)) \\ \qquad\qquad \text{provided } x \notin \mathsf{fn}(Q),\ y \notin \mathsf{fn}(P) \end{array}$$

These can be derived from more fundamental structural equivalences of associativity of parallel composition and scope extrusion.

3. Input

We approach the individual logical connectives by thinking of their meaning as defined by their *right rules* in the sequent calculus. In their process interpretation, we have to analyze what it means to *offer* a particular service along a channel. Linear implication $A \multimap B$ is true if B is true under the (linear) assumption A.

$$\frac{\Delta, A \Rightarrow B}{\Delta \Rightarrow A \multimap B} \multimap R$$

Reading the premise under the process interpretation

$$\Delta, y{:}A \Rightarrow P :: x : B$$

it says that P offers B along x if provided with the opportunity to use A along y. So a process offering $A \multimap B$ must *input* an A and then offer B.

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

Note that we use the same channel name x for the service $A \multimap B$ and then the service B. This is possible because channels are *linear*. Once we input an A along x, we can not input another along x. Instead, the channel x changes state. This is the origin of the term *session types* (or, more broadly, *behavioral types*) for this kind of system. The type $A \multimap B$ describes the type of a session that inputs an A and then behaves like B.

Under this definition, how can we *use* a channel x of type $A \multimap B$? We have to *output* an A along x. In return, we assume that x now offers B. But what precisely does it mean to "*output an* A"? In the π-calculus we do not pass processes, we pass *names* that provide access to processes. So we must have a process P offering A along y and then output that y along x.

$$\frac{\Delta \Rightarrow P :: y : A \quad \Delta', x{:}B \Rightarrow Q :: z : C}{\Delta, \Delta', x{:}A \multimap B \Rightarrow (\boldsymbol{\nu} y)x\langle y\rangle.(P \mid Q) :: z : C} \multimap L$$

Note that each channel in the context must be used either in P or in Q, but not both. This is essential to maintain the linearity in the use of channels. Also, the channel y along which P must offer A must be *bound* in the conclusion, so that there cannot be confusion with any other channel. In the π-calculus this is called a *bound output*, which always outputs a fresh name. Note also that P and Q do not share any names, so they do not communicate with each other directly.

If we strip the process expressions, we obtain just the usual left rule for linear implication in the linear sequent calculus.

$$\frac{\Delta \Rightarrow A \quad \Delta', B \Rightarrow C}{\Delta, \Delta', A \multimap B \Rightarrow C} \multimap L$$

3.1 Reduction

We can define right and left rules in the sequent calculus at will, but to form a coherent logical system they must match appropriately. As a global theorem about a logic, the theorems witnessing such coherence are cut elimination and identity. These decompose into two local properties, separately for each connective, namely *cut reduction* and *identity expansion*. Cut reduction corresponds to process reduction, while identity expansion shows how to reduce channel forwarding at complex types to forwarding at simpler types.

We first consider cut reduction in the sequent calculus. We want to show that the right and left rules match. This means that if we have a cut where the cut formula was just introduced by its left and right rules, then we can reduce it to cuts on subformulas. For linear implication we have

$$\frac{\dfrac{\Delta, A \Rightarrow B}{\Delta \Rightarrow A \multimap B} \multimap R \quad \dfrac{\Delta_1 \Rightarrow A \quad \Delta_2, B \Rightarrow C}{\Delta_1, \Delta_2, A \multimap B \Rightarrow C} \multimap L}{\Delta, \Delta_1, \Delta_2 \Rightarrow C} \mathsf{cut}$$

$$\longrightarrow$$

$$\frac{\dfrac{\Delta_1 \Rightarrow A \quad \Delta, A \Rightarrow B}{\Delta, \Delta_1 \Rightarrow B} \mathsf{cut} \qquad \Delta_2, B \Rightarrow C}{\Delta, \Delta_1, \Delta_2 \Rightarrow C} \mathsf{cut}$$

To obtain a process interpretation of this reduction, we first assign names and processes to the three premises:

$$\begin{array}{l}\Delta, y{:}A \Rightarrow P_1 :: x : B \\ \Delta_1 \Rightarrow P_2 :: w : A \\ \Delta_2, x{:}B \Rightarrow Q :: z : C\end{array}$$

The annotated conclusions before and after the reduction then are:

$$\Delta, \Delta_1, \Delta_2 \Rightarrow (\boldsymbol{\nu} x)(x(y).P_1 \mid (\boldsymbol{\nu} w)(x\langle w\rangle.(P_2 \mid Q))) :: z : C$$

$$\longrightarrow$$

$$\Delta, \Delta_1, \Delta_2 \Rightarrow (\boldsymbol{\nu} x)((\boldsymbol{\nu} w)(P_2 \mid P_1\{w/y\}) \mid Q) :: z : C$$

In order to read this more easily, we can apply the structural congruences of the π-calculus, extruding the bindings on x and w and exchanging P_1 and P_2:

$$\Delta, \Delta_1, \Delta_2 \Rightarrow (\boldsymbol{\nu} x)(\boldsymbol{\nu} w)(x(y).P_1 \mid x\langle w\rangle.(P_2 \mid Q)) :: z : C$$

$$\longrightarrow$$

$$\Delta, \Delta_1, \Delta_2 \Rightarrow (\boldsymbol{\nu} x)(\boldsymbol{\nu} w)(P_1\{w/y\} \mid P_2 \mid Q) :: z : C$$

We see that in the presence of the structural rules, the cut reduction is mirrored by a process reduction, matching an input with a corresponding output. In general, it is an instance of the reduction

$$(x(y).P \mid x\langle w\rangle.Q) \longrightarrow (P\{w/y\} \mid Q)$$

3.2 Expansion

The local reduction discussed above shows the interaction between an offer and a matching use of a service, in this case an input typed as $A \multimap B$. Conversely, when we forward a channel along another one, we should be able to decompose this into forwarding of channels of the component types. First, in the pure sequent calculus:

$$\frac{}{A \multimap B \Rightarrow A \multimap B}\ \mathsf{id} \implies \frac{\dfrac{\dfrac{}{A \Rightarrow A}\ \mathsf{id} \quad \dfrac{}{B \Rightarrow B}\ \mathsf{id}}{A \multimap B, A \Rightarrow B} \multimap L}{A \multimap B \Rightarrow A \multimap B} \multimap R$$

With processes:

$$\frac{}{x{:}A \multimap B \Rightarrow [x \leftrightarrow z] :: z : A \multimap B}\ \mathsf{id}$$

$$\Longrightarrow$$

$$\frac{\dfrac{\dfrac{}{y{:}A \Rightarrow [y \leftrightarrow w] :: w : A}\ \mathsf{id} \quad \dfrac{}{x{:}B \Rightarrow [x \leftrightarrow z] :: z : B}\ \mathsf{id}}{x{:}A \multimap B, y{:}A \Rightarrow (\boldsymbol{\nu} w)x\langle w\rangle.([y \leftrightarrow w] \mid [x \leftrightarrow z]) :: z : B} \multimap L}{x{:}A \multimap B \Rightarrow z(y).(\boldsymbol{\nu} w)x\langle w\rangle.([y \leftrightarrow w] \mid [x \leftrightarrow z]) :: z : A \multimap B} \multimap R$$

In words: we can either forward $z : A \multimap B$ directly to $x : A \multimap B$, or, equivalently, we can input a $y : A$ along z and output a new w to x, where w mimics y at type A and z now mimics x at type B. This kind of interpretation appears inherently typed and difficult to explain or realize in the untyped π-calculus.

4. Output

We have already seen that $A \multimap B$, offering an input, also requires output in order to use the offer. But how do we offer an output? A

natural candidate is $A \otimes B$, which is true if we can prove A and B separately, each using a portion of the available linear hypotheses.

$$\frac{\Delta \Rightarrow A \quad \Delta' \Rightarrow B}{\Delta, \Delta' \Rightarrow A \otimes B}\ \otimes R$$

At first glance, it would appear process offer $A \otimes B$ along x should output two channels, one offering A and another offering B.

$$\frac{\Delta \Rightarrow P :: y : A \quad \Delta' \Rightarrow Q :: z : B}{\Delta, \Delta' \Rightarrow P\ ?\ Q :: x : A \otimes B}\ \otimes R?$$

But we see that this doesn't quite work: P offers along y and Q offers along z, but neither y nor z are even mentioned in the final sequent. Looking at the pattern of $\multimap R$ we see that we need to exploit the linearity of the channel x along which $A \otimes B$ is offered. So we output only a single new channel of type A along x and then offer B, again along x. We could equally well output a new channel of type B and then behave as A, but it seems more natural if the continuation is on the right of the connective.

$$\frac{\Delta \Rightarrow P :: y : A \quad \Delta' \Rightarrow Q :: x : B}{\Delta, \Delta' \Rightarrow (\boldsymbol{\nu} y) x\langle y \rangle.(P \mid Q) :: x : A \otimes B}\ \otimes R$$

Again, this is a form of bound output, but this time as part of an offer.

Conversely, to communicate with a process offering $A \otimes B$ along x we have to input an A along x, after which we still have to communicate with x now of type B.

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

When we remove the process we recognize the usual left rule.

$$\frac{\Delta, A, B \Rightarrow C}{\Delta, A \otimes B \Rightarrow C}\ \otimes L$$

4.1 Reduction

We should verify the cut reduction property and see which form of process reduction it suggests. First, on bare sequents:

$$\frac{\dfrac{\Delta_1 \Rightarrow A \quad \Delta_2 \Rightarrow B}{\Delta_1, \Delta_2 \Rightarrow A \otimes B}\ \otimes R \quad \dfrac{\Delta, A, B \Rightarrow C}{\Delta, A \otimes B \Rightarrow C}\ \otimes L}{\Delta, \Delta_1, \Delta_2 \Rightarrow C}\ \mathsf{cut}$$

$$\longrightarrow$$

$$\frac{\Delta_2 \Rightarrow B \qquad \dfrac{\Delta_1 \Rightarrow A \quad \Delta, A, B \Rightarrow C}{\Delta, \Delta_1, B \Rightarrow C}\ \mathsf{cut}}{\Delta, \Delta_1, \Delta_2 \Rightarrow C}\ \mathsf{cut}$$

We assign names and processes to the open premises:

$$\begin{array}{l}\Delta_1 \Rightarrow P_1 :: y : A \\ \Delta_2 \Rightarrow P_2 :: x : B \\ \Delta, w{:}A, x{:}B \Rightarrow Q :: z : C\end{array}$$

With this process assignment, the reduction viewed on processes becomes:

$$\Delta, \Delta_1, \Delta_2 \Rightarrow (\boldsymbol{\nu} x)((\boldsymbol{\nu} y) x\langle y \rangle.(P_1 \mid P_2) \mid x(w).Q) :: z : C$$
$$\longrightarrow$$
$$\Delta, \Delta_1, \Delta_2 \Rightarrow (\boldsymbol{\nu} x)(P_2 \mid (\boldsymbol{\nu} y)(P_1 \mid Q\{y/w\})) :: z : C$$

Again, this is a simple instance of the usual input/output reduction of the π-calculus, modulo some structural congruences.

4.2 Expansion

The identity expansion for $A \otimes B$:

$$\frac{}{A \otimes B \Rightarrow A \otimes B}\ \mathsf{id} \Longrightarrow \frac{\dfrac{}{A \Rightarrow A}\ \mathsf{id} \quad \dfrac{}{B \Rightarrow B}\ \mathsf{id}}{A, B \Rightarrow A \otimes B}\ \otimes R \Big/ \dfrac{}{A \otimes B \Rightarrow A \otimes B}\ \otimes L$$

If we annotate the sequents with names and processes as

$$x{:}A \otimes B \Rightarrow [x \leftrightarrow z] :: z : A \otimes B$$

we obtain the an expansion symmetric to $A \multimap B$, just reversing the roles of x and z, with input along x and output along z:

$$[x \leftrightarrow z] \Longrightarrow x(y).(\boldsymbol{\nu} w)(z\langle w \rangle.[y \leftrightarrow w] \mid [x \leftrightarrow z])$$

4.3 On Asymmetry

A process offering $A \otimes B$ along x outputs a fresh y offering A and then offers B along x. This asymmetry may seem puzzling, but in fact, $A \otimes B$ and $B \otimes A$ are isomorphic in that we can use one to realize the other. In analogy with the expansion, we construct:

$$x{:}A \otimes B \Rightarrow x(y).(\boldsymbol{\nu} w) z\langle w \rangle.([x \leftrightarrow w] \mid [y \leftrightarrow z]) :: z : B \otimes A$$

In functional languages we have a similar situation: a product $A \times B$ is different from a product $B \times A$, but the two types are isomorphic. Here, too, we can establish an isomorphism, not just a mutual linear entailment [Pérez et al. 2011].

5. Termination

The logical constant $\mathbf{1}$ means we have no antecedents. Consequently, the linear antecedent $\mathbf{1}$ is just replaced by the empty collection of antecedents, which means it is just erased.

$$\frac{}{\cdot \Rightarrow \mathbf{1}}\ \mathbf{1}R \qquad \frac{\Delta \Rightarrow C}{\Delta, \mathbf{1} \Rightarrow C}\ \mathbf{1}L$$

5.1 A Faithful Interpretation

A fruitful way to think of this is as the nullary version of $\otimes$, since it is in fact its unit. We would output nothing and have no continuation:

$$\frac{}{\cdot \Rightarrow x\langle\rangle.\mathbf{0} :: x : \mathbf{1}}\ \mathbf{1}R$$

Conversely, we can expect nothing from an offer of $\mathbf{1}$ along x, except to terminate the connection.

$$\frac{\Delta \Rightarrow Q :: z : C}{\Delta, x{:}\mathbf{1} \Rightarrow x().Q :: z : C}\ \mathbf{1}L$$

The two match perfectly, with the following cut reduction.

$$\frac{\dfrac{}{\cdot \Rightarrow \mathbf{1}}\ \mathbf{1}R \quad \dfrac{\Delta \Rightarrow C}{\Delta, \mathbf{1} \Rightarrow C}\ \mathbf{1}L}{\Delta \Rightarrow C}\ \mathsf{cut}$$

$$\longrightarrow$$

$$\Delta \Rightarrow C$$

On process expressions, labeling the open premise as $\Delta \Rightarrow Q :: z : C$:

$$(\boldsymbol{\nu} x)(x\langle\rangle.\mathbf{0} \mid x().Q) \longrightarrow Q$$

This can be seen as an interaction in the polyadic π-calculus. For completeness, the expansion for $\mathbf{1}$:

$$\frac{}{\mathbf{1} \Rightarrow \mathbf{1}}\ \mathsf{id} \Longrightarrow \frac{\dfrac{}{\cdot \Rightarrow \mathbf{1}}\ \mathbf{1}R}{\mathbf{1} \Rightarrow \mathbf{1}}\ \mathbf{1}L$$

And on proof terms:

$$[x \leftrightarrow z] \Longrightarrow x().z\langle\rangle.\mathbf{0} \qquad \text{for } x{:}\mathbf{1} \text{ and } z{:}\mathbf{1}$$

5.2 Enabling More Parallelism

Even though the assignment above would perfectly fulfill our goal of a Curry-Howard isomorphism, we have proposed an alternative in order to achieve more parallelism in the composition of independent processes. In general, a process P that does not offer anything is typed as $\Delta \Rightarrow P :: x : \mathbf{1}$. Once composed with appropriate processes as required by Δ, it should be a closed process that evolves by internal actions only. If we have a second such process, $\Delta' \Rightarrow Q :: z : \mathbf{1}$ each should be able to evolve independently, without interaction. We could achieve this by adding $x{:}\mathbf{1}$ to Δ' and then using cut.

$$\dfrac{\Delta \Rightarrow P :: x : \mathbf{1} \quad \dfrac{\Delta' \Rightarrow Q :: z : \mathbf{1}}{\Delta', x{:}\mathbf{1} \Rightarrow x().Q :: z : \mathbf{1}}\ \mathbf{1}L}{\Delta, \Delta' \Rightarrow (\boldsymbol{\nu} x)(P \mid x().Q) :: z : \mathbf{1}}\ \mathsf{cut}$$

However, because of the action prefix $x()$ guarding Q, this corresponds to a *sequential composition* (P before Q) rather than a parallel composition. If we cut the other way, we have Q before P. In other words, we can not compose noninteracting processes in a truly parallel manner.

In order to achieve independent parallel composition we could allow some reductions under prefixes. In the inference above, x does not occur in Q, so reducing underneath the $x()$ prefix never creates any difficulties. Going back to, say, inputs, a prefix $x(y).a(b).P$ where x, y, a and b are all distinct could safely input on x first or on a first. In proof theory, this phenomenon is well understood. An exchange of the two prefixes would correspond to an exchange between two consective $\otimes L$ rules, a so-called *commuting conversion*. These commuting conversions can be analyzed as observational equivalences on session-typed π-calculus processes [Pérez et al. 2011], thus justifying their use here. However, it represents a significant departure from the operational reading of the π-calculus, so we will not pursue this idea further here.

We could also write $x().\mathbf{0} \mid Q$ as the proof term for $\mathbf{1}L$, recovering parallelism, but we would still need a commuting conversion in the proof theory to justify reduction. There is yet another alternative for recovering additional parallelism, at least for the $\mathbf{1}L$ and $\mathbf{1}R$ rules, which we have proposed in previous work [Caires and Pfenning 2010]. Instead of maintaining a full isomorphism between proofs and processes, we *contract* proofs to processes, making the $\mathbf{1}L$ rule *silent*. We never use a channel $x{:}\mathbf{1}$ from the context in a process expression but eliminate it with the $\mathbf{1}L$ rule. This must be matched by the right rule, whose process no longer has an action prefix but is simply $\mathbf{0}$.

$$\dfrac{}{\cdot \Rightarrow \mathbf{0} :: x : \mathbf{1}}\ (\mathbf{1}R) \qquad \dfrac{\Delta \Rightarrow Q :: z : C}{\Delta, x{:}\mathbf{1} \Rightarrow Q :: z : C}\ (\mathbf{1}L)$$

Then cut reduction is just a structural congruence on the π-calculus side.

$$(\boldsymbol{\nu} x)(\mathbf{0} \mid Q) \equiv Q \qquad x \notin \mathsf{fn}(Q)$$

We can now derive a symmetric independent composition rule, using the structural congruence $(\boldsymbol{\nu} x)P \equiv P$, provided $x \notin \mathsf{fn}(P)$.

$$\dfrac{\Delta \Rightarrow P :: x : \mathbf{1} \quad \dfrac{\Delta' \Rightarrow Q :: z : 1}{\Delta', x{:}\mathbf{1} \Rightarrow Q :: z : 1}\ (\mathbf{1}L)}{\Delta, \Delta' \Rightarrow (P \mid Q) :: z : 1}\ \mathsf{cut}$$

The expansion

$$x{:}\mathbf{1} \Rightarrow [x \leftrightarrow z] :: z : \mathbf{1} \Longrightarrow \dfrac{\dfrac{}{\cdot \Rightarrow \mathbf{0} :: z : \mathbf{1}}\ (\mathbf{1}R)}{x{:}\mathbf{1} \Rightarrow \mathbf{0} :: z : \mathbf{1}}\ (\mathbf{1}L)$$

yields $[x \leftrightarrow z] \Longrightarrow \mathbf{0}$ for $x{:}\mathbf{1}$, $z{:}\mathbf{1}$.

5.3 An Example

We now illustrate the fragment of our system we have presented thus far through a simple example. We will further develop the example throughout the presentation, as we incrementally introduce the connectives that make up the full system.

Our example consists of a simple PDF indexing service. The high-level concept is a server that receives a PDF file from its client and then returns an indexed version of the file (a file containing a word index for searches). The system we have developed up to this point allows us to model such a server with the following type:

$$\mathsf{Indexer} \triangleq \mathsf{file} \multimap (\mathsf{file} \otimes \mathbf{1})$$

We abstract away the details of what constitutes a proper PDF file with the type file. The type Indexer describes the communication behavior that is expected of the server: it will first perform an input (of the file), followed by an output of a file, after which it terminates. A process that implements such a service along channel x is:

$$\mathsf{Srv} \triangleq x(f).(\boldsymbol{\nu} y)x\langle y\rangle.(P \mid \mathbf{0}) :: x : \mathsf{Indexer}$$

We abstract away the details of the actual indexing of the file in the process P. A possible client for the server is given below (where the process Q represents the clients use of the indexed files):

$$\mathsf{Client} \triangleq (\boldsymbol{\nu} \mathit{pdf})x\langle \mathit{pdf}\rangle.x(\mathit{idx}).(Q \mid \mathbf{0})$$

and we can compose the server and the client as follows:

$$\dfrac{\cdot \Rightarrow \mathsf{Srv} :: x : \mathsf{Indexer} \quad x : \mathsf{Indexer} \Rightarrow \mathsf{Client} :: z : \mathbf{1}}{\cdot \Rightarrow (\nu x)(\mathsf{Srv} \mid \mathsf{Client}) :: z : \mathbf{1}}\ \mathsf{cut}$$

6. Taking Stock, Part I

Let's look at the fragment of the logic so far. If we take the faithful interpretation of $\mathbf{1}$, we have on the process side:

Types	A, B, C	::=	$A \multimap B$	input
		$\mid$	$A \otimes B$	output
		$\mid$	$\mathbf{1}$	termination
Processes	P, Q	::=	$[x \leftrightarrow z]$	forwarding
		$\mid$	$(P \mid Q)$	parallel composition
		$\mid$	$(\boldsymbol{\nu} x)P$	name restriction
		$\mid$	$x(y).P$	input
		$\mid$	$(\nu y)x\langle y\rangle.P$	bound output
		$\mid$	$x().P$	(suspension)
		$\mid$	$x\langle\rangle.\mathbf{0}$	(termination)

We also have the following reductions on well-typed processes, mimicking cut reductions. The typing requirement implies some straightforward conditions on the occurrences of bound variables which we do not detail separately.

Structural	$(\boldsymbol{\nu} x)([y \leftrightarrow x] \mid Q) \longrightarrow Q\{y/x\}$
	$(\boldsymbol{\nu} x)(P \mid [x \leftrightarrow z]) \longrightarrow P\{z/x\}$
Interaction	$(\boldsymbol{\nu} x)(x(y).P \mid (\nu w)x\langle w\rangle.Q)$
	$\longrightarrow (\boldsymbol{\nu} x)(\boldsymbol{\nu} w)(P\{w/y\} \mid Q)$
(Termination)	$(\boldsymbol{\nu} x)(x\langle\rangle.\mathbf{0} \mid x().Q) \longrightarrow Q$

and the following structural congruences, in addition to standard α-conversion:

$$\begin{array}{ll} P \mid (Q \mid R) \equiv (P \mid Q) \mid R & \\ P \mid Q \equiv Q \mid P & \\ P \mid (\boldsymbol{\nu} x)Q \equiv (\boldsymbol{\nu} x)(P \mid Q) & \text{for } x \notin \mathsf{fn}(P) \\ (\boldsymbol{\nu} x)(\boldsymbol{\nu} y)P \equiv (\boldsymbol{\nu} y)(\boldsymbol{\nu} x)P & \end{array}$$

If we use the optimized form of $\mathbf{1}$, we replace $x().P$ and $x\langle\rangle.\mathbf{0}$ just by $\mathbf{0}$ and remove *termination* as a form of interaction. We need to add the congruences $P \mid \mathbf{0} \equiv P$ and $(\boldsymbol{\nu} x)\mathbf{0} \equiv \mathbf{0}$.

In both versions we have relatively standard constructs of the π-calculus, where forwarding can be implemented as sketched earlier using a new channel. We can also obtain regular (not bound) output using forwarding. The following is a deduction which shows we achieve the effect of outputting an existing channel $w{:}A$ along $x : A \otimes B$ and then offer B along x.

$$\dfrac{\dfrac{}{w{:}A \Rightarrow [w \leftrightarrow y] :: y : A}\ \mathsf{id} \quad \Delta \Rightarrow Q :: x : B}{\Delta, w{:}A \Rightarrow (\boldsymbol{\nu} y)x\langle y\rangle.([w \leftrightarrow y] \mid Q) :: x : A \otimes B}\ \otimes R$$

A pleasing aspect of this process interpretation is that process reduction matches up exactly with proof reduction, modulo structural congruence on processes. Typing is a bit unusual, because we have to type processes modulo structural congruence for the purpose of type preservation. This is not too difficult, but we conjecture that in the design of programming language constructs based on our system, one can take a more restrictive view on syntax and relegate the structural congruences to the runtime state of communicating processes. A related departure from the usual Curry-Howard isomorphism is that the process terms assigned to particular inference rules mix multiple different constructs. For example, the $\multimap L$ rule requires a name restriction, an output, and a parallel composition in its process term. However, the particular combination of these constructs is idiomatic and has been observed multiple times in the literature.

In the version of $\mathbf{1}$ with more intrinsic parallelism, we take some liberty with the isomorphism because the $\mathbf{1}$ rule is silent, that is, has no trace in the process expression. As a consequence, some cut reductions are structural congruences on the process side.

Another possibility of restoring independent parallel composition is to generally allow interactions under independent prefixes. *Independence* here refers to syntactic conditions on the names used in action prefixes. For example, a process $x(y).u(w).P$ might input first on x or on u as long as all four names are distinct. Allowing such exchanges mirrors *commuting conversion* of the sequent calculus and could potentially uncover significant latent parallelism. Proof theory is a strong guide, and we have already developed the necessary theoretical tools to analyze such an extension [Pérez et al. 2011].

7. Branching

Branching, or *external choice*, means to offer both A and B along a channel and let the client decide which one to use. We add a binary guarded choice to the process calculus [Sangiorgi and Walker 2001] and additive conjunction $A \mathbin{\&} B$ to our fragment of linear logic. Since only either A or B will be used (at the client's discretion), the context is propagated to both premises. Conversely, if $A \mathbin{\&} B$ is in the environment, we have to chose either A or B.

$$\dfrac{\Delta \Rightarrow A \quad \Delta \Rightarrow B}{\Delta \Rightarrow A \mathbin{\&} B}\ \mathbin{\&} R$$

$$\dfrac{\Delta, A \Rightarrow C}{\Delta, A \mathbin{\&} B \Rightarrow C}\ \mathbin{\&} L_1 \qquad \dfrac{\Delta, B \Rightarrow C}{\Delta, A \mathbin{\&} B \Rightarrow C}\ \mathbin{\&} L_2$$

When assigning process terms we again exploit linearity of channels. We indicate a choice (inl or inr) on a channel and then continue on the same channel with either A or B, respectively.

$$\dfrac{\Delta \Rightarrow P :: x : A \quad \Delta \Rightarrow Q :: x : B}{\Delta \Rightarrow x.\mathsf{case}(P, Q) :: x : A \mathbin{\&} B}\ \mathbin{\&} R$$

$$\dfrac{\Delta, x{:}A \Rightarrow Q :: z : C}{\Delta, x{:}A \mathbin{\&} B \Rightarrow x.\mathsf{inl}; Q :: z : C}\ \mathbin{\&} L_1$$

$$\dfrac{\Delta, x{:}B \Rightarrow Q :: z : C}{\Delta, x{:}A \mathbin{\&} B \Rightarrow x.\mathsf{inr}; Q :: z : C}\ \mathbin{\&} L_2$$

Reduction is also straightforward. In particular, process reduction and proof reduction match perfectly. We leave the details for the reader to reconstruct.

$$\begin{array}{l} (\boldsymbol{\nu} x)(x.\mathsf{case}(P, Q) \mid x.\mathsf{inl}; R) \longrightarrow (\boldsymbol{\nu} x)(P \mid R) \\ (\boldsymbol{\nu} x)(x.\mathsf{case}(P, Q) \mid x.\mathsf{inr}; R) \longrightarrow (\boldsymbol{\nu} x)(Q \mid R) \end{array}$$

For the expansion, from $x{:}A \mathbin{\&} B \Rightarrow [x \leftrightarrow z] :: z : A \mathbin{\&} B$ we obtain

$$[x \leftrightarrow z] \Longrightarrow z.\mathsf{case}(x.\mathsf{inl}; [x \leftrightarrow z], x.\mathsf{inr}; [x \leftrightarrow z])$$

8. Choice

Choice, more precisely *internal choice*, means to offer either A or B along a channel x, the offering process is the one to decide. A party using x must therefore be prepared for both A and B. We do not need to extend the process language for this, having already added binary guarded choice, but we need $A \oplus B$ as an appropriate logical connective.

$$\dfrac{\Delta \Rightarrow A}{\Delta \Rightarrow A \oplus B}\ \oplus R_1 \qquad \dfrac{\Delta \Rightarrow B}{\Delta \Rightarrow A \oplus B}\ \oplus R_2$$

$$\dfrac{\Delta, A \Rightarrow C \quad \Delta, B \Rightarrow C}{\Delta, A \oplus B \Rightarrow C}\ \oplus L$$

We present the process assignment with the guarded choice.

$$\dfrac{\Delta \Rightarrow P :: x : A}{\Delta \Rightarrow x.\mathsf{inl}; P :: x : A \oplus B}\ \oplus R_1$$

$$\dfrac{\Delta \Rightarrow P :: x : B}{\Delta \Rightarrow x.\mathsf{inr}; P :: x : A \oplus B}\ \oplus R_2$$

$$\dfrac{\Delta, x{:}A \Rightarrow P :: z : C \quad \Delta, x{:}B \Rightarrow Q :: z : C}{\Delta, x{:}A \oplus B \Rightarrow x.\mathsf{case}(P, Q) :: z : C}\ \oplus L$$

The reduction was already discussed for branching. The expansion from $x{:}A \oplus B \Rightarrow [x \leftrightarrow z] :: z : A \oplus B$ is

$$[x \leftrightarrow z] \Longrightarrow x.\mathsf{case}(z.\mathsf{inl}; [x \leftrightarrow z], z.\mathsf{inr}; [x \leftrightarrow z])$$

9. Revisiting the Example

We now extend our PDF indexer to incorporate branching and choice, embodied in the types $\mathbin{\&}$ and $\oplus$. The server so far only implements a single service: indexing. Conceivably, a server implements multiple services. For instance, we might want the server to check whether an indexed PDF is indeed indexed correctly. This verification service can be modelled through the following type:

$$\mathsf{Verif} \triangleq \mathsf{file} \multimap ((\mathsf{valid} \otimes \mathbf{1}) \oplus (\mathsf{invalid} \otimes \mathbf{1}))$$

The verification service inputs the pdf file that is to be verified and then will either output a message signaling the file was valid, or output a message signaling otherwise. This form of choice that is internal to the server is modelled by the use of $\oplus$. A process

implementing the verification service is (along channel x):

$$\cdot \Rightarrow x(f).\mathsf{Check} :: x : \mathsf{Verif}$$

with $\mathsf{Check} :: x : ((\mathsf{valid} \otimes \mathbf{1}) \oplus (\mathsf{invalid} \otimes \mathbf{1}))$ depending on the validity of the received file. To include this service in our original example, we make use of $\&$ to obtain:

$$\mathsf{2Services} \triangleq \mathsf{Indexer} \,\&\, \mathsf{Verif}$$

And the process offering the two services becomes:

$$\cdot \Rightarrow x.\mathsf{case}(\mathsf{Srv}, x(f).\mathsf{Check}) :: x : \mathsf{2Services}$$

A client that wishes to use the verification service must then choose accordingly (and branch on the two possibilities, abstracted here by the processes P_{ok} and P_{nok}):

$$\mathsf{ClientVerif} \triangleq x.\mathsf{inr}; (\boldsymbol{\nu} pdf)x\langle pdf\rangle.x.\mathsf{case}(P_{\mathsf{ok}}, P_{\mathsf{nok}})$$

10. Replication

So far, the logic we have considered is purely linear. In order to capture process replication, we will need the proposition $!A$ as a type. A process offering a service $!A$ along x is offering the service A *persistently*: clients may use A as many times as they would like, including not at all. The judgmentally sound way to capture this logically is the sequent formulation of *dual intuitionistic linear logic* [Barber 1997; Chang et al. 2003], which goes back to Andreoli's *dyadic* formulation of classical linear logic [Andreoli 1992]. We distinguish another judgment on antecedents, namely that A is *persistently available*. It is customary to sort the linear and persistent antecedents into separate zones, so that their judgmental status is obvious from their position in the sequent.

$$\underbrace{B_1, \ldots, B_k}_{\Gamma} \quad ; \quad \underbrace{A_1, \ldots A_n}_{\Delta} \quad \Rightarrow C$$

persistently true — linearly true

When we label the sequents with channels and processes, not much changes. We use a different kind of letter, u, to stand for *shared* channels. They may occur arbitrarily often in the process P, unlike the linear channels.

$$u_1{:}B_1, \ldots, u_k{:}B_k \;;\; x_1{:}A_1, \ldots, x_n{:}A_n \Rightarrow P :: z : C$$

All the rules we have shown so far are extended to allow a context Γ of persistent antecedents in the conclusion and all premises. This context satisfies weakening, contraction, and exchange, according to the persistent nature of the assumptions in them. We will use these properties silently.

10.1 Judgmental Rules

Persistent truth is defined as truth not depending on linear assumptions. Therefore we have a new rule of cut to eliminate a persistent antecedent A from a sequent, where one premise is a proof of A *with no linear antecedents*. This allows us to use this premise each time the persistent A is used, without violating any linearity constraints.

$$\frac{\Gamma \;;\; \cdot \Rightarrow A \quad \Gamma, A \;;\; \Delta \Rightarrow C}{\Gamma \;;\; \Delta \Rightarrow C}\;\mathsf{cut!}$$

It is straightforward to annotate the premises with channels and processes, but what about the conclusion?

$$\frac{\Gamma \;;\; \cdot \Rightarrow P :: x : A \quad \Gamma, u{:}A \;;\; \Delta \Rightarrow Q :: z : C}{\Gamma \;;\; \Delta \Rightarrow \,? :: z : C}\;\mathsf{cut!}$$

The idea is for a shared channel u to represent the offer of a *replicated input*. What u receives is a *channel x of type A* along which a single session of type A can proceed. Meanwhile, u remains available to receive another channel of type A. Each use of A is satisfied by a different copy of P, as we will see. So:

$$\frac{\Gamma \;;\; \cdot \Rightarrow P :: x : A \quad \Gamma, u{:}A \;;\; \Delta \Rightarrow Q :: z : C}{\Gamma \;;\; \Delta \Rightarrow (\boldsymbol{\nu} u)(!u(x).P \mid Q) :: z : C}\;\mathsf{cut!}$$

Based on this intuition we have to send u a new channel $y{:}A$ in order to use u. This is realized in the copy rule. Logically, it is justified by saying that persistent truth entails linear truth.

$$\frac{\Gamma, u{:}A \;;\; \Delta, y{:}A \Rightarrow P :: z : C}{\Gamma, u{:}A \;;\; \Delta \Rightarrow (\boldsymbol{\nu} y)u\langle y\rangle.P :: z : C}\;\mathsf{copy}$$

What happens if a cut! meets a copy? Logically, this is straightforward: we spawn two new cuts.

$$\frac{\Gamma \;;\; \cdot \Rightarrow A \quad \dfrac{\Gamma, A \;;\; \Delta, A \Rightarrow C}{\Gamma, A \;;\; \Delta \Rightarrow C}\;\mathsf{copy}}{\Gamma \;;\; \Delta \Rightarrow C}\;\mathsf{cut!}$$

$$\longrightarrow$$

$$\frac{\Gamma \;;\; \cdot \Rightarrow A \quad \dfrac{\Gamma \;;\; \cdot \Rightarrow A \quad \Gamma, A \;;\; \Delta, A \Rightarrow C}{\Gamma \;;\; \Delta, A \Rightarrow C}\;\mathsf{cut!}}{\Gamma \;;\; \Delta \Rightarrow C}\;\mathsf{cut}$$

The first one is considered smaller because the right subproof is shorter; the second one takes place on a smaller judgment (A linearly true vs. A persistently true). If we annotate these with processes, we have the open premises

$$\Gamma \;;\; \cdot \Rightarrow P :: x : A$$
$$\Gamma, u{:}A \;;\; \Delta, y{:}A \Rightarrow Q :: z : C$$

Replaying the proof reduction on the assigned processes yields:

$$\begin{array}{rl} & (\boldsymbol{\nu} u)(!u(x).P \mid (\boldsymbol{\nu} y)u\langle y\rangle.Q) \\ \longrightarrow & (\boldsymbol{\nu} y)(P\{y/x\} \mid (\boldsymbol{\nu} u)(!u(x).P \mid Q)) \end{array}$$

With some structural congruences we obtain a familiar reduction for replicated input, in this case interacting with a bound output.

We also have to consider some additional interactions of the new judgmental rules with the old ones, specifically cut and identity. These are commuting conversions or structural equivalences between proofs, exchanging different forms of cut; on the π-calculus side they are behavioral equivalences called *sharpened replication theorems* [Sangiorgi and Walker 2001]. We omit the details which can be found in Caires and Pfenning [2010].

So far, all the rules and reductions arose entirely from judgmental considerations—we did not yet introduce the $!A$ connective!

10.2 Sharing

After preparing the grounds by analyzing the judgment of persistent truth associated with shared channels, we can now internalize the judgment with the corresponding proposition $!A$. The $!R$ rule just requires the proof of the premise not to depend on linear assumptions. The $!L$ rule just promotes the linear antecedent $!A$ to the persistent antecedent A.

$$\frac{\Gamma \;;\; \cdot \Rightarrow A}{\Gamma \;;\; \cdot \Rightarrow\, !A}\;!R \qquad \frac{\Gamma, A \;;\; \Delta \Rightarrow C}{\Gamma \;;\; \Delta, !A \Rightarrow C}\;!L$$

From the process perspective, $!R$ introduces a replicated input, while $!L$ just corresponds to promoting a channel from linear to shared status.

$$\frac{\Gamma \;;\; \cdot \Rightarrow P :: y : A}{\Gamma \;;\; \cdot \Rightarrow\, !x(y).P :: x : !A}\;!R$$

$$\frac{\Gamma, u{:}A \;;\; \Delta \Rightarrow Q :: z : C}{\Gamma \;;\; \Delta, x{:}!A \Rightarrow x/u.Q :: z : C}\;!L$$

In order to retain an isomorphism between proofs and processes, and also respect the linearity of channels x, we have an explicit

renaming construct $x/u.Q$ which binds u with scope Q. This is just an explicit substitution, for typing purposes, mapping into $Q\{x/u\}$.

We have to check that these rule match up. Fortunately, a cut at type $!A$ turns into a cut! at type A.

$$\dfrac{\dfrac{\Gamma\,;\cdot \Rightarrow A}{\Gamma\,;\cdot \Rightarrow\, !A}\;!R \quad \dfrac{\Gamma, A\,;\Delta \Rightarrow C}{\Gamma\,;\Delta, !A \Rightarrow C}\;!L}{\Gamma\,;\Delta \Rightarrow C}\;\mathsf{cut}$$

$$\longrightarrow$$

$$\dfrac{\Gamma\,;\cdot \Rightarrow A \quad \Gamma, A\,;\Delta \Rightarrow C}{\Gamma\,;\Delta \Rightarrow C}\;\mathsf{cut!}$$

If we annotate these with process expressions, we see that it just renames a bound variable and removes an explicit renaming construct.

$$(\boldsymbol{\nu} x)(!x(y).P \mid x/u.Q) \longrightarrow (\boldsymbol{\nu} u)(!u(y).P \mid Q)$$

In order to recover some additional parallelism, we can proceed similarly to the $\mathbf{1}L$ rule. We can either permit reduction and interaction under the x/u prefix, or we can carry out the substitution to obtain $Q\{x/u\}$. Either one is another small departure from a perfect Curry-Howard isomorphism, contracting several distinct proofs to the same program, and collapsing some proof reductions into congruences. The $!L/!R$ reduction above would turn into α-conversion.

$$(\boldsymbol{\nu} x)(!x(y).P \mid Q\{x/u\}) \equiv (\boldsymbol{\nu} u)(!u(y).P \mid Q)$$

10.3 Expansion

The expansion for $!A$:

$$\cdot\,;\, !A \Rightarrow\, !A \quad\Longrightarrow\quad \dfrac{\dfrac{\dfrac{\dfrac{}{A\,;A \Rightarrow A}\;\mathsf{id}}{A\,;\cdot \Rightarrow A}\;\mathsf{copy}}{A\,;\cdot \Rightarrow\, !A}\;!R}{\cdot\,;\, !A \Rightarrow\, !A}\;!L$$

Annotating the sequent as $\cdot\,;\, w{:}!A \Rightarrow [w \leftrightarrow x] :: x : !A$ we obtain the expansion

$$[w \leftrightarrow x] \Longrightarrow w/u.!x(z).(\boldsymbol{\nu} y)(u\langle y\rangle.[y \leftrightarrow z])$$

10.4 Replication and Sharing in the Example

Our PDF indexing service is still only interesting insofar as it can only be used by a client once. To make the service persistent, or replicated, we augment the type with the exponential:

$$\mathsf{PersistentS} \triangleq\, !\mathsf{2Services}$$

Since our implementation uses no ambient linear resources, we need only add a replicated input to the server code to obey the specification:

$$\cdot;\cdot \Rightarrow\, !y(x).x.\mathsf{case}(\mathsf{Srv}, x(f).\mathsf{Check}) :: y : \mathsf{PersistentS}$$

Clients of the persistent service must now trigger a replication of the server. For instance, a client for the verification service must be extended as follows:

$$\mathsf{ClientVerifP} \triangleq (\boldsymbol{\nu} x)y\langle x\rangle.\mathsf{ClientVerif}$$

11. Taking Stock, Part II

The extension of the calculus by internal and external choice and replication does not create any new difficulties. Much of the computational content for replication is associated with the judgmental rules, rather than the proposition $!A$ itself.

In order to retain a full isomorphism we need a construct reminiscent of an explicit substitution that promotes a linear variable to be persistent. As for $\mathbf{1}L$, we justify either carrying out this substitution, or reducing under the promotion prefix. In the first case we stay within the π-calculus, but not all proof reductions correspond to process reductions. This is analogous to the tradeoff we made for the $\mathbf{1}L$ rule.

We extend the syntax and the reductions.

Types	A, B, C	$::=$	$\ldots$	
		$\mid$	$A \,\&\, B$	external choice
		$\mid$	$A \oplus B$	internal choice
		$\mid$	$!A$	replication
Processes	P, Q	$::=$	$\ldots$	
		$\mid$	$x.\mathsf{inl}; P \mid x.\mathsf{inr}; P$	selection
		$\mid$	$x.\mathsf{case}(P, Q)$	branching
		$\mid$	$!u(x).P$	replicating input
		$\mid$	$x/u.P$	(promotion)

We also have the following new reductions

(Structural)	$(\boldsymbol{\nu} x)(!x(y).P \mid x/u.Q) \longrightarrow (\boldsymbol{\nu} u)(!u(y).P \mid Q)$
Choice	$(\boldsymbol{\nu} x)(x.\mathsf{case}(P, Q) \mid x.\mathsf{inl}; R) \longrightarrow (\boldsymbol{\nu} x)(P \mid R)$
	$(\boldsymbol{\nu} x)(x.\mathsf{case}(P, Q) \mid x.\mathsf{inr}; R) \longrightarrow (\boldsymbol{\nu} x)(Q \mid R)$
Replication	$(\boldsymbol{\nu} u)(!u(x).P \mid (\boldsymbol{\nu} y)u\langle y\rangle.Q)$
	$\longrightarrow (\boldsymbol{\nu} y)(P\{y/x\} \mid (\boldsymbol{\nu} u)(!u(x).P \mid Q))$

The new structural reduction and the promotion construct disappear if we annotate the conclusion of the $!L$ rule with $Q\{x/u\}$ instead of $x/u.Q$.

12. Term Passing

So far, we have stayed very close to the π-calculus, establishing an interpretation of linear propositions as session types, sequent proofs as processes, and cut reduction as process reduction. Structural congruence arises from structural equivalences between sequent proofs.

Next, we would like to work towards a *type theory* which integrates both ordinary functional computation and concurrency. We can see two possible paths. We can *embed* functional programming, using typed analogues to translations by Milner [1992]. This is indeed possible, since variants of Milner's translations can be typed in our system [Toninho et al. 2011b], but we expect an extension to dependent types and a full type theory to be far from straightforward. Essentially, we would have to reason about functional programs by reasoning about their implementations via session-typed concurrent processes. A second possibility is to extend the calculus with terms from a type theory. This is somewhat reminiscent of the applied π-calculus [Abadi and Fournet 2001] with a rich, dependently typed term language. This is what we pursue here.

12.1 Judgments

Terms and values are drawn from a possibly dependent type theory. This type theory is open-ended, since its interaction with the calculus developed so far is strictly controlled. We go so far as not to specify whether its operational semantics should be call-by-name or call-by-value, but we will isolate the assumptions we make about it in order to guarantee that the combined system is meaningful.

We have a basic judgment $M : \tau$, where τ is a type and M is a term. We generally say "*term*" to refer to a term from the functional language and will continue to use "*process*" for process expressions. We use the basic judgment in its hypothetical form,

$$\Psi \vdash M : \tau,$$

where Ψ is a collection of hypotheses $x_i{:}\tau_i$. As usual, all the x_i are distinct term variables, and we assume that it is a hypothetical judgment and therefore satisfies substitution properties and the hypothesis rule. We assume that hypotheses Ψ are not linear, for

the sake of simplicity. In its simply-typed version, the judgment $\Psi \vdash M : \tau$ would be related to intuitionistic propositional logic via the standard form of the Curry-Howard isomorphism [Howard 1969].

We now generalize all the sequent judgments so far to add new hypotheses Ψ, assigning types to term variables, written as

$$\Psi \,;\, \Gamma \,;\, \Delta \Rightarrow P :: x : A$$

Like the shared names in Γ, the typing assumptions in Ψ are propagated to all premises in all rules we have presented so far. For example, the identity and cut rules are now

$$\frac{}{\Psi \,;\, \Gamma \,;\, x{:}A \Rightarrow [x \leftrightarrow z] :: z : A}\ \mathsf{id}$$

$$\frac{\Psi \,;\, \Gamma \,;\, \Delta \Rightarrow P :: x : A \quad \Psi \,;\, \Gamma \,;\, \Delta', x{:}A \Rightarrow Q :: z : C}{\Psi \,;\, \Gamma \,;\, \Delta, \Delta' \Rightarrow (\boldsymbol{\nu} x)(P \mid Q) :: z : C}\ \mathsf{cut}$$

The generalization of sequents is straightforward, but we must internalize term typing within process typing in order to use them. We will do this in a minimalistic way, using only two new propositions, $\forall x{:}\tau.A$ and $\exists x{:}\tau.A$.

12.2 Term Input

Input of terms is modeled simply by universal quantification, taking our cue from the usual functional interpretation of type theory.

$$\frac{\Psi, y{:}\tau \,;\, \Gamma \,;\, \Delta \Rightarrow P :: x : A}{\Psi \,;\, \Gamma \,;\, \Delta \Rightarrow x(y).P :: x : \forall y{:}\tau.A}\ \forall R$$

As before, the type of channel x evolves through interaction. In order for cut reduction to work correctly, the $\forall L$ rule must provide a matching output.

$$\frac{\Psi \vdash M : \tau \quad \Psi \,;\, \Gamma \,;\, \Delta', x{:}A\{M/y\} \Rightarrow Q :: z : C}{\Psi \,;\, \Gamma \,;\, \Delta', x{:}\forall y{:}\tau.A \Rightarrow x\langle M\rangle.Q :: z : C}\ \forall L$$

Again, as before, we reuse the name x in the premise without conflict since x is linear. Note that y must be chosen fresh so that the new context $\Psi, y{:}\tau$ is well-formed. This can always be achieved by renaming. We extend our notation of substitution of names for names to terms for names, writing $\{M/y\}$ for the capture-avoiding substitution of M for y in A. By the substitution property of the type theory, the result will always be well-typed.

12.3 Reduction

Applying cut to the right and left rules as formulated above yields the conclusion

$$\Psi \,;\, \Gamma \,;\, \Delta, \Delta' \Rightarrow (\boldsymbol{\nu} x)(x(y).P \mid x\langle M\rangle.Q) :: z : C$$

To applying the usual reduction step from the sequent calculus, we must substitute M for y in the premise of the $\forall R$. We see that we need the substitution property for hypotheses in Ψ to justify reduction. After that we obtain the following cut:

$$\frac{\Psi \,;\, \Gamma \,;\, \Delta \Rightarrow P\{M/y\} :: x : A\{M/y\} \quad \Psi \,;\, \Gamma \,;\, \Delta', x{:}A\{M/y\} \Rightarrow Q :: z : C}{\Psi \,;\, \Gamma \,;\, \Delta, \Delta' \Rightarrow (\boldsymbol{\nu} x)(P\{M/y\} \mid Q) :: z : C}\ \mathsf{cut}$$

from which we read off the reduction

$$(\boldsymbol{\nu} x)(x(y).P \mid x\langle M\rangle.Q) \longrightarrow (\boldsymbol{\nu} x)(P\{M/y\} \mid Q)$$

In other words, we just use term passing instead of name passing in the π-calculus.

12.4 Expansion

Expanding the judgment $x{:}\forall y{:}\tau.A \Rightarrow [x \leftrightarrow z] :: z : \forall y{:}\tau.A$ yields

$$\dfrac{\dfrac{\dfrac{}{y{:}\tau \vdash y{:}\tau}\ \mathsf{hyp} \quad \dfrac{}{y{:}\tau \,;\, \cdot \,;\, x{:}A \Rightarrow [x \leftrightarrow z] :: z : A}\ \mathsf{id}}{y{:}\tau \,;\, \cdot \,;\, x{:}\forall y{:}\tau.A \Rightarrow x\langle y\rangle.[x \leftrightarrow z] :: z : A}\ \forall L}{\cdot \,;\, \cdot \,;\, x{:}\forall y{:}\tau.A \Rightarrow z(y).x\langle y\rangle.[x \leftrightarrow z] :: z : \forall y{:}\tau.A}\ \forall R$$

We see that we need the hypothesis rule in the type theory to justify the expansion.

$$[x \leftrightarrow z] \Longrightarrow z(y).x\langle y\rangle.[x \leftrightarrow z]$$

12.5 Term Output

A channel $x : \exists y{:}\tau.A$ offers to output a term M of type τ along x and then offer $A\{M/y\}$. This is symmetric to term input as described for $\forall y{:}\tau.A$. So even though our logic is intuitionistic, we obtain a strong duality between universal and existential quantification.

$$\frac{\Psi \vdash M : \tau \quad \Psi \,;\, \Gamma \,;\, \Delta \Rightarrow P :: x : A\{M/y\}}{\Psi \,;\, \Gamma \,;\, \Delta \Rightarrow x\langle M\rangle.P :: x : \exists y{:}\tau.A}\ \exists R$$

$$\frac{\Psi, y{:}\tau \,;\, \Gamma \,;\, \Delta', x{:}A \Rightarrow Q :: z : C}{\Psi \,;\, \Gamma \,;\, \Delta', x{:}\exists y{:}\tau.A \Rightarrow x(y).Q :: z : C}\ \exists L$$

Applying cut to these two rules yields the conclusion

$$\Psi \,;\, \Gamma \,;\, \Delta, \Delta' \Rightarrow (\boldsymbol{\nu} x)(x\langle M\rangle.P \mid x(y).Q) :: z : C$$

Straightforward substitution into the premise of $\exists L$ and cutting the result with the premise of $\exists R$ yields

$$\frac{\Psi \,;\, \Gamma \,;\, \Delta \Rightarrow P :: x : A\{M/y\} \quad \Psi \,;\, \Gamma \,;\, \Delta', x{:}A\{M/y\} \Rightarrow Q\{M/y\} :: z : C}{\Psi \,;\, \Gamma \,;\, \Delta, \Delta' \Rightarrow (\boldsymbol{\nu} x)(P \mid Q\{M/y\}) :: z : C}\ \mathsf{cut}$$

So modulo symmetry of parallel composition, the reduction is the same as for the $\forall R/\forall L$ pair:

$$(\boldsymbol{\nu} x)(x\langle M\rangle.P \mid x(y).Q) \longrightarrow (\boldsymbol{\nu} x)(P \mid Q\{M/y\})$$

A similar observation can be made about the expansion, which for a judgment $x{:}\exists y{:}\tau.A \Rightarrow [x \leftrightarrow z] :: z : \exists y{:}\tau.A$ yields

$$[x \leftrightarrow z] \Longrightarrow x(y).z\langle y\rangle.[x \leftrightarrow z]$$

The new constructs of the type theory and process language including terms and term passing are incorporated into the overall summary in Figures 1, 2, and 3.

12.6 Incorporating Dependencies in the Example

While we can claim that our running example of the PDF indexing service models two services (an indexer and a verifier), the types we have presented only really specify communications (an input followed by an output or an input followed by a choice, for example). Using dependent types, we can enrich the protocol description to have it enforce properties on the actual data that is communicated. A common idiom that arises when using dependencies in this manner is writing $\forall x{:}\tau.A$ where x does not occur in A. We emphasize the non-dependence with the abbreviation $\tau \supset A$ for the universal quantifier and $\tau \wedge A$ for the existential quantifier ($\exists x{:}\tau.A$ for x not in A). Consider the original indexing service. We can specify that the communicated objects are indeed valid PDF files as follows (using the idioms mentioned above):

$$\mathsf{IndexD} \triangleq \forall f{:}\mathsf{file}.\ \mathsf{pdf}(f) \supset \exists r{:}\mathsf{file}.\ \mathsf{pdf}(r) \wedge \mathbf{1}$$

The type IndexD specifies that the process is to input a file and a *proof* that the file is a valid PDF. Afterwards, the process must

output a second file and a proof that this file is also a valid PDF. A process implementing such a service along channel x might be

$$\mathsf{SrvD} \triangleq x(f).x(p).x\langle\pi_1(\mathsf{genIndex}\, f\, p)\rangle.\\ x\langle\pi_2(\mathsf{genIndex}\, f\, p)\rangle.\mathbf{0}$$

where genIndex is a dependently typed function implementing the index generation, with type $\Pi f{:}\mathsf{file}.\, \Pi p{:}\mathsf{pdf}(f).\ \Sigma r{:}\mathsf{file}.\ \mathsf{pdf}(r)$.

13. Proof Irrelevance

One particularly important aspect of a dependent type theory is that proofs are first-class objects. This means that proofs can be offered and requested during interactions on sessions. For example, a process of type $x : \mathsf{Fix}$ persistently offers a service computing a fixed point of a given function if it exists, or a proof that there is no fixed point.

$$\mathsf{Fix}_1 \triangleq x : !\forall f{:}\mathsf{nat} \to \mathsf{nat}.\ (\exists p{:}(\Sigma m{:}\mathsf{nat}.\ m \doteq f(m)).\ \mathbf{1})\\ \oplus\ (\exists q{:}(\neg\Sigma m{:}\mathsf{nat}.\ m \doteq f(m)).\ \mathbf{1})$$

We assume here that the type theory provides function types $\Pi x{:}\tau.\sigma$ (which may be written $\tau \to \sigma$ if there are no dependencies), dependent sum types $\Sigma x{:}\tau.\sigma$ (inhabited by dependently typed pairs, which may be written $\tau \times \sigma$ if there are no dependencies), negation $\neg\tau$ (inhabited by proofs of contradiction from τ), and equality $M \doteq N$ (inhabited by equational proofs).

In practice, we may not care about the proof that a fixed point does not exist; we are happy to trust the server. In that case we can employ *proof irrelevance* (for example, in the form of *squash types* [Constable et al. 1986] or *bracket types* [Awodey and Bauer 2004]). Briefly, the type $[\tau]$ is inhabited by terms $[M]$ where $M : \tau$, but M can be shown not to be relevant to the outcome of a functional computation. The rules are elided here—they are reviewed in Toninho et al. [2011a]; Pfenning et al. [2011]. The type theory guarantees that witnesses for bracket types need not be transmitted during sessions, and we assume that session participants agree on that. Then we can rewrite the specification above as

$$\mathsf{Fix}_2 \triangleq x : !\forall f{:}\mathsf{nat} \to \mathsf{nat}.\ (\exists p{:}(\Sigma m{:}\mathsf{nat}.m \doteq f(m)).\mathbf{1})\\ \oplus\ (\exists q{:}[\neg\Sigma m{:}\mathsf{nat}.m \doteq f(m)].\mathbf{1})$$

We can even go a step further and realize that no proof of the equality $m \doteq f(m)$ is needed, since we can do this check ourselves by applying f to the witness provided by the server. We obtain:

$$\mathsf{Fix}_3 \triangleq x : !\forall f{:}\mathsf{nat} \to \mathsf{nat}.\ (\exists p{:}(\Sigma m{:}\mathsf{nat}.[m \doteq f(m)]).\mathbf{1})\\ \oplus\ \exists q{:}[\neg\Sigma m{:}\mathsf{nat}.m \doteq f(m)].\mathbf{1})$$

After erasure of information that is not transmitted, this type is isomorphic to

$$\mathsf{Fix}_4 \triangleq x : !\forall f{:}\mathsf{nat} \to \mathsf{nat}.\ (\exists p{:}\mathsf{nat}.\mathbf{1}) \oplus \mathbf{1}$$

so we can optimize both computation and communication if we agree that the optimizations are applied at both ends of sessions. Of course, this last specification is much less informative than the previous ones.

13.1 Example with Proof Irrelevance

While it is often necessary to have processes explicitly communicate proof objects, there are situations where the extra communication overhead may not be desirable, or even necessary. For instance, in our example, proofs for the validity of the exchanged PDF files are communicated explicitly. This verification can potentially be made by the client and the server (with some extra computational overhead) and the communication of the proofs may be suppressed. To capture this at the level of types, we make use of proof irrelevance as follows:

$$\mathsf{IndexID} \triangleq \forall f{:}\mathsf{file}.\ [\mathsf{pdf}(f)] \supset \exists r{:}\mathsf{file}.\ [\mathsf{pdf}(r)] \wedge \mathbf{1}$$

Note that during type-checking, the proof objects are required to exist (and so the contract is still enforced by typing) but the actual objects are suppressed at runtime and thus their communication may be erased.

$$\mathsf{SrvID} \triangleq x(f).x([p]).x\langle\pi_1(\mathsf{genIndex}\, f\, p)\rangle.\\ x\langle[\pi_2(\mathsf{genIndex}\, f\, p)]\rangle.\mathbf{0})$$

The process above is what one would expect during type-checking: all terms are present to ensure the desired properties. The optimization procedure would then produce the following:

$$\mathsf{SrvID}_{\mathsf{opt}} \triangleq x(f).x\langle\pi_1(\mathsf{genIndex}\, f\, p)\rangle.\mathbf{0}$$

14. Digital Signatures

A common way mobile code is certified in practice is through *digital signatures*. We now sketch how to extend our type theory to capture them, so that trust in mobile code can be acquired either through explicit proof, or via digital signatures, or some combination of these techniques. This means that type checking at channel interfaces in the distributed setting must have the ability to check digital signature for authenticity, say, using some public key infrastructure.

On the logical side, we have the new judgment of affirmation [Garg et al. 2006] by a principal K, written as $\Psi \vdash M :_K \tau$ (K affirms that M has type τ). The rule that defines it is

$$\frac{\Psi \vdash M : \tau}{\Psi \vdash \langle M{:}\tau\rangle_K :_K \tau}\ \mathsf{affirm}$$

The notation $\langle M{:}\tau\rangle_K$ literally refers to a certificate that M has type τ, signed by K. We internalize this judgment in the type theory with a new principal-indexed family of modal operators $\Diamond_K\tau$. It is defined by the following two rules:

$$\frac{\Psi \vdash M :_K \tau}{\Psi \vdash M : \Diamond_K\tau}\ \Diamond I \qquad \frac{\Psi \vdash M : \Diamond_K\tau \quad \Psi, x{:}\tau \vdash N :_K \sigma}{\Psi \vdash \mathbf{let}\ \langle x{:}\tau\rangle_K = M\ \mathbf{in}\ N :_K \sigma}\ \Diamond E$$

So far, the digital signature only serves to cut down proof-checking time: if we trust the signer, we do not need to the check proof. When combined with proof irrelevance, however, it becomes much more significant: since irrelevant proofs of type $[\tau]$ are not transmitted, signing a statement of the form $\langle[M]{:}[\tau]\rangle$ actually becomes $\langle[]{:}[\tau]\rangle$ in transmission. By putting our trust in the signer, we avoid transmitting the proof.

Resuming the fixed point example, if we call the principal offering the service F, then we can express that the non-existence of a fixed point must be certified by F as:

$$\mathsf{Fix}_5 \triangleq x : !\forall f{:}\mathsf{nat} \to \mathsf{nat}.\ (\exists p{:}(\Sigma m{:}\mathsf{nat}.[m \doteq f(m)]).\mathbf{1})\\ \oplus\ \exists q{:}\Diamond_F[\neg\Sigma m{:}\mathsf{nat}.m \doteq f(m)].\mathbf{1})$$

In contrast, the proof that x is a fixed point can be omitted outright, no signature required, because we can easily check the fixed point property.

We replay the interaction on $x : \mathsf{Fix}_5$ on a simple example.

Client :	$x\langle y\rangle$	send fresh channel y
Client :	$y\langle\lambda n{:}\mathsf{nat}.\ n+1\rangle$	send function
Server :	$y.\mathsf{inr}$	select "no fixed point"
Server :	$y\langle\langle[]{:}[\neg\Sigma m{:}\mathsf{nat}.\ m \doteq m+1]\rangle_F\rangle$	send signed affirmation

14.1 Example with Digital Signatures

Our indexing service guarantees that the communicated files are valid PDF files (either by requiring and supplying the necessary proof objects or by eliding them at runtime). However, nothing yet guarantees that the file sent by the service is actually correctly indexed. We integrate this in the type of the service by having it

include an extra proof object that attests such a fact:

$$\mathsf{IdxA} \triangleq \forall f{:}\mathsf{file}.[\mathsf{pdf}(f)] \supset \exists r{:}\mathsf{file}.\,[\mathsf{pdf}(r)] \wedge [\mathsf{agree}(f,r)] \wedge \mathbf{1}$$

Given that the proof objects can be suppressed, we wish to enforce some form of accountability. In particular, we want the indexing service to *sign* the proof certificate, attesting agreement between f and r. We can do so by changing the type accordingly (assuming the identification of the service is X):

$$\mathsf{IdxSign} \triangleq \forall f{:}\mathsf{file}.[\mathsf{pdf}(f)] \supset \exists r{:}\mathsf{file}.\,[\mathsf{pdf}(r)] \wedge \Diamond_X[\mathsf{agree}(f,r)] \wedge \mathbf{1}$$

The actual server code can be easily extended to implement this service by outputting an extra, signed, proof object. In an actual implementation, the digital signature would involve cryptography through some underlying public key infrastructure.

15. Conclusions

We have developed a Curry-Howard isomorphism between propositions in linear logic and session types, between proofs in the intuitionstic linear sequent calculus and π-calculus processes, and between proof reductions and process reductions. The sequential nature of sequent proofs impedes some expected parallelism, some of which can be recovered by small local improvements of the type assignment [Caires and Pfenning 2010]. More sweeping changes would exploit a recently developed theory of observational equivalences based on linear logical relations [Pérez et al. 2011]. The basis in logic suggest straightforward generalizations, such as embedding and passing terms from a functional type theory [Toninho et al. 2011b]. This functional type theory must satisfy only some minimal requirements, which allows us to add support for proof erasure and digital signatures [Pfenning et al. 2011]. In total, we have an expressive, extensible basis for concurrent systems, including such architectures as proof-carrying code. In all the system we obtain type preservation, global progress (which implies freedom from deadlock), and termination, guided by proof-theoretic properties and the Curry-Howard correspondence.

Currently, the underlying functional type theory can be dependent, but not the concurrent part. This means we can reason precisely about the terms being passed, but not about properties of the channels or processes themselves beyond those guaranteed by the simple types. We speculate that there are three plausible paths towards a concurrent constructive type theory. One might be to encapsulate the concurrent part in a monad and work within a linear dependent type theory such as CLF [Watkins et al. 2004], although the monad in CLF plays a very different role from what is needed here. A second one might be to use a typed translation from a functions to processes and embed the functional language in the process calculus [Milner 1992; Toninho et al. 2011b]. This requires an understanding how dependent types fare under the type-directed translation and, in particular, which process equalities model judgmental equality in the functional type theory. We have been developing a theory of typed observational equivalences [Pérez et al. 2011], which would likely play a key role in this approach. A third approach might be to use a proof-theoretic form of the a *resource semantics* [Reed 2009] in order to circumvent the usual obstacles to linear dependent types. In this case the interaction between the resource semantics and the operational semantics would seem to be crucial.

Acknowledgments

Support for this research was provided by the Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) through the Carnegie Mellon Portugal Program, under grants SFRH / BD / 33763 / 2009 and INTERFACES NGN-44 / 2009, and CITI.

References

M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *28th Symposium on Principles of Programming Languages*, POPL'01, pages 104–115, London, United Kingdom, 2001. ACM.

S. Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, 111:3–57, 1993.

J.-M. Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):197–347, 1992.

S. Awodey and A. Bauer. Propositions as [types]. *Journal of Logic and Computation*, 14(4):447–471, 2004.

A. G. Barber. *Linear Type Theories, Semantics and Action Calculi*. PhD thesis, University of Edinburgh, 1997. Available as Technical Report ECS-LFCS-97-371.

L. Caires and F. Pfenning. Session types as intuitionistic linear propositions. In *Proceedings of the 21st International Conference on Concurrency Theory (CONCUR 2010)*, pages 222–236, Paris, France, Aug. 2010. Springer LNCS 6269.

B.-Y. E. Chang, K. Chaudhuri, and F. Pfenning. A judgmental analysis of linear logic. Technical Report CMU-CS-03-131R, Carnegie Mellon University, Department of Computer Science, Dec. 2003.

A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

A. Church and J. Rosser. Some properties of conversion. *Transactions of the American Mathematical Society*, 39(3):472–482, May 1936.

R. L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

H. B. Curry. Functionality in combinatory logic. *Proceedings of the National Academy of Sciences, U.S.A.*, 20:584–590, 1934.

D. Garg, L. Bauer, K. Bowers, F. Pfenning, and M. Reiter. A linear logic of affirmation and knowledge. In D. Gollman, J. Meier, and A. Sabelfeld, editors, *Proceedings of the 11th European Symposium on Research in Computer Security (ESORICS'06)*, pages 297–312, Hamburg, Germany, Sept. 2006. Springer LNCS 4189.

J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

J.-Y. Girard. Towards a geometry of interaction. In J. W. Gray and A. Scedrov, editors, *Categories in Computer Science and Logic*, pages 69–108. American Mathematical Society, 1989. Contemporary Mathematics, Volume 92.

K. Honda. Types for dyadic interaction. In *4th International Conference on Concurrency Theory*, CONCUR'93, pages 509–523. Springer LNCS 715, 1993.

K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *7th European Symposium on Programming Languages and Systems*, ESOP'98, pages 122–138. Springer LNCS 1381, 1998.

W. A. Howard. The formulae-as-types notion of construction. Unpublished note. An annotated version appeared in: *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, 479–490, Academic Press (1980), 1969.

J. M. E. Hyland and C.-H. L. Ong. Pi-calculus, dialogue games and PCF. In *7th Conference on Functional Programming Languages and Computer Architecture*, FPCA'95, pages 96–107, La Jolla, California, June 1995. ACM.

P. Martin-Löf. Constructive mathematics and computer programming. In *Logic, Methodology and Philosophy of Science VI*, pages 153–175. North-Holland, 1980.

M. Merro and D. Sangiorgi. On asynchrony in name-passing calculi. *Mathematical Structures in Computer Science*, 14(5):715–767, 2004.

R. Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2(2):119–141, 1992.

$$\frac{}{\Psi\ ;\ \Gamma\ ;\ x{:}A \Rightarrow [x \leftrightarrow z] :: z : A}\ \mathsf{id} \qquad \frac{}{\Psi\ ;\ \Gamma\ ;\ \cdot \Rightarrow \mathbf{0} :: x : \mathbf{1}}\ (\mathbf{1}R) \qquad \frac{\Psi\ ;\ \Gamma\ ;\ \Delta \Rightarrow P :: z : C}{\Psi\ ;\ \Gamma\ ;\ \Delta, x{:}\mathbf{1} \Rightarrow P :: z : C}\ (\mathbf{1}L)$$

$$\frac{\Psi\ ;\ \Gamma\ ;\ \cdot \Rightarrow P :: y : A}{\Psi\ ;\ \Gamma\ ;\ \cdot \Rightarrow !z(y).P :: z : !A}\ !R \qquad \frac{\Psi\ ;\ \Gamma, u{:}A\ ;\ \Delta \Rightarrow P :: z : C}{\Psi\ ;\ \Gamma\ ;\ \Delta, x{:}!A \Rightarrow P\{x/u\} :: z : C}\ (!L) \qquad \frac{\Psi\ ;\ \Gamma, u{:}A\ ;\ \Delta, y{:}A \Rightarrow P :: z : C}{\Psi\ ;\ \Gamma, u{:}A\ ;\ \Delta \Rightarrow (\boldsymbol{\nu} y)u\langle y\rangle.P :: z : C}\ \mathsf{copy}$$

$$\frac{\Psi\ ;\ \Gamma\ ;\ \Delta \Rightarrow P :: z : A \quad \Psi\ ;\ \Gamma\ ;\ \Delta \Rightarrow Q :: z : B}{\Psi\ ;\ \Gamma\ ;\ \Delta \Rightarrow z.\mathsf{case}(P, Q) :: z : A \,\&\, B}\ \&R \qquad \frac{\Psi\ ;\ \Gamma\ ;\ \Delta, x{:}A \Rightarrow P :: z : C}{\Psi\ ;\ \Gamma\ ;\ \Delta, x{:}A \,\&\, B \Rightarrow x.\mathsf{inl}; P :: z : C}\ \&L_1$$

$$\frac{\Psi\ ;\ \Gamma\ ;\ \Delta, x{:}B \Rightarrow P :: z : C}{\Psi\ ;\ \Gamma\ ;\ \Delta, x{:}A \,\&\, B \Rightarrow x.\mathsf{inr}; P :: z : C}\ \&L_2 \qquad \frac{\Psi\ ;\ \Gamma\ ;\ \Delta_1 \Rightarrow P :: y : A \quad \Psi\ ;\ \Gamma\ ;\ \Delta_2 \Rightarrow Q :: z : B}{\Psi\ ;\ \Gamma\ ;\ \Delta_1, \Delta_2 \Rightarrow (\boldsymbol{\nu} y)z\langle y\rangle.(P \mid Q) :: z : A \otimes B}\ \otimes R$$

$$\frac{\Psi\ ;\ \Gamma\ ;\ \Delta, y{:}A, x{:}B \Rightarrow P :: z : C}{\Psi\ ;\ \Gamma\ ;\ \Delta, x{:}A \otimes B \Rightarrow x(y).P :: z : C}\ \otimes L \qquad \frac{\Psi\ ;\ \Gamma\ ;\ \Delta \Rightarrow P :: z : A}{\Psi\ ;\ \Gamma\ ;\ \Delta \Rightarrow z.\mathsf{inl}; P :: z : A \oplus B}\ \oplus R_1$$

$$\frac{\Psi\ ;\ \Gamma\ ;\ \Delta \Rightarrow P :: z : B}{\Psi\ ;\ \Gamma\ ;\ \Delta \Rightarrow z.\mathsf{inr}; P :: z : A \oplus B}\ \oplus R_2 \qquad \frac{\Psi\ ;\ \Gamma\ ;\ \Delta, x{:}A \Rightarrow P :: z : C \quad \Psi\ ;\ \Gamma\ ;\ \Delta, x{:}B \Rightarrow Q :: z : C}{\Psi\ ;\ \Gamma\ ;\ \Delta, x{:}A \oplus B \Rightarrow x.\mathsf{case}(P, Q) :: z : C}\ \oplus L$$

$$\frac{\Psi, x{:}\tau\ ;\ \Gamma\ ;\ \Delta \Rightarrow P :: z : A}{\Psi\ ;\ \Gamma\ ;\ \Delta \Rightarrow z(x).P :: z : \forall x{:}\tau.A}\ \forall R \qquad \frac{\Psi \vdash N : \tau \quad \Psi\ ;\ \Gamma\ ;\ \Delta, x{:}A\{N/y\} \Rightarrow P :: z : C}{\Psi\ ;\ \Gamma\ ;\ \Delta, x{:}\forall y{:}\tau.A \Rightarrow x\langle N\rangle.P :: z : C}\ \forall L$$

$$\frac{\Psi \vdash N : \tau \quad \Psi\ ;\ \Gamma\ ;\ \Delta \Rightarrow P : A\{N/x\}}{\Psi\ ;\ \Gamma\ ;\ \Delta \Rightarrow z\langle N\rangle.P :: z : \exists x{:}\tau.A}\ \exists R \qquad \frac{\Psi, y{:}\tau\ ;\ \Gamma\ ;\ \Delta, x{:}A \Rightarrow P :: z : C}{\Psi\ ;\ \Gamma\ ;\ \Delta, x{:}\exists y{:}\tau.A \Rightarrow x(y).P :: z : C}\ \exists L$$

$$\frac{\Psi\ ;\ \Gamma\ ;\ \Delta_1 \Rightarrow P :: x : A \quad \Psi\ ;\ \Gamma\ ;\ \Delta_2, x{:}A \Rightarrow Q :: z : C}{\Psi\ ;\ \Gamma\ ;\ \Delta_1, \Delta_2 \Rightarrow (\boldsymbol{\nu} x)(P \mid Q) :: z : C}\ \mathsf{cut} \qquad \frac{\Psi\ ;\ \Gamma\ ;\ \cdot \Rightarrow P :: x : A \quad \Psi\ ;\ \Gamma, u{:}A\ ;\ \Delta \Rightarrow Q :: z : C}{\Psi\ ;\ \Gamma\ ;\ \Delta \Rightarrow (\nu u)((!u(x).P) \mid Q) :: z : C}\ \mathsf{cut!}$$

Figure 1. Dependently-typed sessions.

$$\frac{}{\Psi\ ;\ \Gamma\ ;\ \cdot \Rightarrow x\langle\rangle.\mathbf{0} :: x : \mathbf{1}}\ \mathbf{1}R \qquad \frac{\Psi\ ;\ \Gamma\ ;\ \Delta \Rightarrow P :: z : C}{\Psi\ ;\ \Gamma\ ;\ \Delta, x{:}\mathbf{1} \Rightarrow x().P :: z : C}\ \mathbf{1}L \qquad \frac{\Psi\ ;\ \Gamma, u{:}A\ ;\ \Delta \Rightarrow P :: z : C}{\Psi\ ;\ \Gamma\ ;\ \Delta, x{:}!A \Rightarrow x/u.P :: z : C}\ !L$$

$$x\langle\rangle.\mathbf{0} \mid x().Q \longrightarrow Q$$
$$(\boldsymbol{\nu} x)(!x(y).P \mid x/u.Q) \longrightarrow (\boldsymbol{\nu} u)(!u(y).P \mid Q)$$

Figure 2. Alternative faithful process assignment, with additional reductions.

$$\begin{array}{ll} P \mid \mathbf{0} \equiv P & P \equiv_\alpha Q \Rightarrow P \equiv Q \\ P \mid (Q \mid R) \equiv (P \mid Q) \mid R & P \mid Q \equiv Q \mid P \\ x \notin \mathsf{fn}(P) \Rightarrow P \mid (\boldsymbol{\nu} x)Q \equiv (\boldsymbol{\nu} x)(P \mid Q) & (\boldsymbol{\nu} x)\mathbf{0} \equiv \mathbf{0} \\ (\boldsymbol{\nu} x)(\boldsymbol{\nu} y)P \equiv (\boldsymbol{\nu} y)(\boldsymbol{\nu} x)P & \end{array}$$

$$\begin{array}{ll} x\langle y\rangle.P \mid x(z).Q \longrightarrow P \mid Q\{y/z\} & u\langle y\rangle.P \mid !u(z).Q \longrightarrow P \mid Q\{y/z\} \mid !u(z).Q \\ x\langle M\rangle.P \mid x(n).Q \longrightarrow P \mid Q\{M/n\} & x.\mathsf{inl}; P \mid x.\mathsf{case}(Q, R) \longrightarrow P \mid Q \\ x.\mathsf{inr}; P \mid x.\mathsf{case}(Q, R) \longrightarrow P \mid R & y \notin \mathsf{fn}(Q) \Rightarrow (\boldsymbol{\nu} x)([y \leftrightarrow x] \mid Q) \longrightarrow Q\{y/x\} \\ z \notin \mathsf{fn}(P) \Rightarrow (\boldsymbol{\nu} x)(P \mid [x \leftrightarrow z]) \longrightarrow Q\{z/x\} & Q \longrightarrow Q' \Rightarrow P \mid Q \longrightarrow P \mid Q' \\ P \longrightarrow Q \Rightarrow (\boldsymbol{\nu} y)P \longrightarrow (\boldsymbol{\nu} y)Q & P \equiv P' \wedge P' \longrightarrow Q' \wedge Q' \equiv Q \Rightarrow P \longrightarrow Q \end{array}$$

$$\begin{array}{lcl} [x \leftrightarrow z]_{A \multimap B} & \Longrightarrow & z(y).(\boldsymbol{\nu} w)(x\langle w\rangle.([y \leftrightarrow w]_A \mid [x \leftrightarrow z]_B)) \\ [x \leftrightarrow z]_{\mathbf{1}} & \Longrightarrow & \mathbf{0} \\ [x \leftrightarrow z]_{A \oplus B} & \Longrightarrow & x.\mathsf{case}(z.\mathsf{inl}; [x \leftrightarrow z]_A, z.\mathsf{inr}; [x \leftrightarrow z]_B) \\ [x \leftrightarrow z]_{\forall y:\tau.A} & \Longrightarrow & z(y).x\langle y\rangle.[x \leftrightarrow z]_A \end{array} \qquad \begin{array}{lcl} [x \leftrightarrow z]_{A \otimes B} & \Longrightarrow & x(y).(\boldsymbol{\nu} w)(z\langle w\rangle.([y \leftrightarrow w]_A \mid [x \leftrightarrow z]_B)) \\ [x \leftrightarrow z]_{A \,\&\, B} & \Longrightarrow & z.\mathsf{case}(x.\mathsf{inl}; [x \leftrightarrow z]_A, x.\mathsf{inr}; [x \leftrightarrow z]_B) \\ [w \leftrightarrow x]_{!A} & \Longrightarrow & !x(z).(\boldsymbol{\nu} y)(w\langle y\rangle.[y \leftrightarrow z]_A) \\ [x \leftrightarrow z]_{\exists y:\tau.A} & \Longrightarrow & x(y).z\langle y\rangle.[x \leftrightarrow z]_A \end{array}$$

Figure 3. Processes: structural congruence, reduction and expansion

R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Information and Computation*, 100(1):1–77, Sept. 1992. Parts I and II.

J. A. Pérez, L. Caires, F. Pfenning, and B. Toninho. Termination in session-based concurrency via linear logical relations. Submitted, Oct. 2011.

F. Pfenning, L. Caires, and B. Toninho. Proof-carrying code in a session-typed process calculus. In *1st International Conference on Certified Programs and Proofs*, CPP'11, Kenting, Taiwan, Dec. 2011. Springer LNCS. To appear.

D. Prawitz. *Natural Deduction*. Almquist & Wiksell, Stockholm, 1965.

J. C. Reed. *A Hybrid Logical Framework*. PhD thesis, Carnegie Mellon University, Sept. 2009. Available as Technical Report CMU-CS-09-155.

D. Sangiorgi and D. Walker. *The π-Calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.

B. Toninho, L. Caires, and F. Pfenning. Dependent session types via intuitionistic linear type theory. In *Proceedings of the 13th International Conference on Principles and Practice of Declarative Programming*, PPDP'11, pages 161–172, Odense, Denmark, July 2011a. ACM.

B. Toninho, L. Caires, and F. Pfenning. Functions as session-typed processes. Submitted, Oct. 2011b.

K. Watkins, I. Cervesato, F. Pfenning, and D. Walker. Specifying properties of concurrent computations in CLF. In C.Schürmann, editor, *Proceedings of the 4th International Workshop on Logical Frameworks and Meta-Languages (LFM'04)*, Cork, Ireland, July 2004. Electronic Notes in Theoretical Computer Science (ENTCS), vol 199, pp. 133–145, 2008.