

# A Concurrent Interpretation of Intuitionistic Linear Logic

Luís Caires

Departamento de Informática and CITI  
FCT, Universidade Nova de Lisboa, Portugal

Frank Pfenning

Department of Computer Science  
Carnegie Mellon University, USA

## Abstract

*Several type disciplines for  $\pi$ -calculi have been proposed, in which linearity frequently plays a key role. In this paper, we introduce a type system for the  $\pi$ -calculus that exactly corresponds to the standard sequent calculus proof system for dual intuitionistic linear logic, based on an interpretation of linear propositions as session types. Perhaps surprisingly, the induced type discipline reveals a close connection between intuitionistic linear logic, linear and session types for mobile processes, and forms of concurrent functional evaluation. We also show that our type system ensures session fidelity and the absence of deadlocks, and a tight operational correspondence between  $\pi$ -calculus reductions and cut elimination steps.*

## 1 Introduction

Since its appearance, linear logic has been intensively explored in the analysis of  $\pi$ -calculus models for communicating and mobile system, given its essential ability to deal with resources, effects, and non-interference. The fundamental way it provides for analyzing notions of sharing versus uniqueness, captured by the exponential “!”, seems to have been a source of inspiration for Milner when introducing replication in the  $\pi$ -calculus [20]. Following the early works of Abramsky [1], several authors have exploited the  $\pi$ -calculus as a programming language to express proof reductions (e.g., [5]) or game semantics (e.g., [17]) in systems of linear logic. Different research directions have drawn inspiration from linear logic for developing type-theoretic analyses of mobile processes, motivated by the works of Kobayashi, Pierce, and Turner [19]; a similar influence is already noticeable in the first publications by Honda on session types [15]. Many expressive type disciplines for  $\pi$ -calculi in which linearity frequently plays a key role have been proposed since then (e.g., [18, 16, 24]).

Linearity has been usually employed in such systems in indirect ways, exploiting its basic type context management techniques, or the assignment of usage multiplicities

to channels [19], rather than the deeper type-theoretic significance of linear logical operators.

In this paper we present a type system for the  $\pi$ -calculus that exactly corresponds to the standard sequent calculus proof system for dual intuitionistic linear logic. The key to our correspondence is a new, perhaps surprising, interpretation of intuitionistic linear logic formulas as a form of session types [15, 16], in which the programming language is a session-typed  $\pi$ -calculus, and the type structure consists precisely of the connectives of intuitionistic linear logic, retaining their standard proof-theoretic interpretation.

In session-based concurrency, processes communicate through so-called session channels, connecting exactly two subsystems, and communication is disciplined by session protocols so that actions always occur in dual pairs: when one partner sends, the other receives; when one partner offers a selection, the other chooses; when a session terminates, no further interaction may occur. New sessions may be dynamically created by invocation of shared servers. Such a model exhibits concurrency in the sense that several sessions, not necessarily causally related, may be executing simultaneously. Mobility is also present, since both session and server names may be passed around (delegated) in communications. Session types [16] discipline interactions in sessions. It turns out that the connectives of intuitionistic linear logic suffice to express finite session disciplines.

While in the linear  $\lambda$ -calculus types are assigned to terms (denoting values, functions), in our interpretation types are assigned to names (denoting communication channels) and describe their session protocol. The essence of our interpretation may already be found in the interpretation of the linear logic multiplicatives as behavioral prefix operators. Traditionally, an object of type  $A \multimap B$  denotes a linear function that given an object of type  $A$  returns an object of type  $B$  [14]. In our interpretation, an object of type  $A \multimap B$  denotes a session  $x$  that first inputs a session channel of type  $A$ , and then behaves as  $B$ , where  $B$  specifies again an interactive behavior, rather than a closed value. Linearity of  $\multimap$  is essential, otherwise the behavior of the input session after communication could not be ensured. An object of type  $A \otimes B$  denotes a session that first sends a session channel of

type  $A$  and afterwards behaves as  $B$ . But notice that objects of type  $A \otimes B$  really consist of two objects: the sent session of type  $A$  and the continuation session, of type  $B$ . These two objects are separate and non-interfering, as enforced by the canonical semantics of the linear multiplicative conjunction ( $\otimes$ ). Our interpretation of  $A \otimes B$  appears asymmetric, but the proof of  $A \otimes B \vdash B \otimes A$ , realized by an appropriately typed process, coerces any session of type  $A \otimes B$  to a session of type  $B \otimes A$ . The other linear constructors are then given compatible interpretations, in particular, the  $!A$  type is naturally interpreted as a type of shared servers.

We briefly summarize the contributions of the paper. We describe a type system for the  $\pi$ -calculus (Section 3) that corresponds to the sequent calculus for dual intuitionistic linear logic DILL (Section 4). The correspondence is bidirectional and tight, in the sense that (a) any  $\pi$ -calculus computation can be simulated by proof reductions on typing derivations (Theorem 5.3), thus establishing a strong form of subject reduction (Theorem 6.1), and (b) that any proof reduction or conversion corresponds either to a computation step or to a process equivalence on the  $\pi$ -calculus side (Theorems 5.4 and 5.5). An intrinsic consequence of the logical typing is a global progress property, that ensures the absence of deadlock for systems with an arbitrary number of open sessions (Theorem 6.3). Finally, we illustrate the expressiveness of our system (Section 7) with some examples: typing session systems, encoding typed data, and an embedding of the linear  $\lambda$ -calculus with exponential types.

## 2 Process Model

We briefly introduce the syntax and operational semantics of the process model: the synchronous  $\pi$ -calculus (see [22]) extended with (binary) guarded choice.

**Definition 2.1 (Processes)** *Given an infinite set  $\Lambda$  of names  $(x, y, z, u, v)$ , the set of processes  $(P, Q, R)$  is defined by*

$$\begin{array}{l|l|l}
 P ::= & \mathbf{0} & | P \mid Q \\
 & (\nu y)P & | x\langle y \rangle.P \\
 & x(y).P & | !x(y).P \\
 & x.\text{inl}; P & | x.\text{inr}; P \quad | \quad x.\text{case}(P, Q)
 \end{array}$$

The operators  $\mathbf{0}$  (inaction),  $P \mid Q$  (parallel composition), and  $(\nu y)P$  (name restriction) comprise the static fragment of any  $\pi$ -calculus. We then have  $x\langle y \rangle.P$  (send  $y$  on  $x$  and proceed as  $P$ ),  $x(y).P$  (receive a name  $z$  on  $x$  and proceed as  $P$  with the input parameter  $y$  replaced by  $z$ ), and  $!x(y).P$  which denotes replicated (or persistent) input. The remaining three operators define a minimal labeled choice mechanism, comparable to the  $n$ -ary branching constructs found in standard session  $\pi$ -calculi (see e.g., [16]). For the sake of minimality and without loss of generality we restrict our model to binary choice. In restriction  $(\nu y)P$  and

input  $x(y).P$  the distinguished occurrence of the name  $y$  is binding, with scope the process  $P$ . For any process  $P$ , we denote the set of *free names* of  $P$  by  $fn(P)$ . A process is *closed* if it does not contain free occurrences of names. We identify process up to consistent renaming of bound names, writing  $\equiv_\alpha$  for this congruence. We write  $P\{x/y\}$  for the process obtained from  $P$  by capture avoiding substitution of  $x$  for  $y$  in  $P$ . Structural congruence expresses basic identities on the structure of processes.

**Definition 2.2** *Structural congruence ( $P \equiv Q$ ), is the least congruence relation on processes such that*

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

Next, we define the behavior of processes under reduction.

**Definition 2.3** *Reduction ( $P \rightarrow Q$ ), is the binary relation on processes inductively defined by:*

$$\begin{array}{ll}
 x\langle y \rangle.Q \mid x(z).P \rightarrow Q \mid P\{y/z\} & (RC) \\
 x\langle y \rangle.Q \mid !x(z).P \rightarrow Q \mid P\{y/z\} \mid !x(z).P & (R!) \\
 x.\text{inl}; P \mid x.\text{case}(Q, R) \rightarrow P \mid Q & (RL) \\
 x.\text{inr}; P \mid x.\text{case}(Q, R) \rightarrow P \mid R & (RR) \\
 Q \rightarrow Q' \Rightarrow P \mid Q \rightarrow P \mid Q' & (R|) \\
 P \rightarrow Q \Rightarrow (\nu y)P \rightarrow (\nu y)Q & (R\nu) \\
 P \equiv P', P' \rightarrow Q', Q' \equiv Q \Rightarrow P \rightarrow Q & (R\equiv)
 \end{array}$$

Notice that reduction is closed (by definition) under structural congruence. Reduction specifies the computations a process performs on its own. To characterize the interactions a process may perform with its environment, we introduce a labeled transition system. It coincides with the standard early transition system for the  $\pi$ -calculus [22], extended with appropriate labels and transition rules for the choice constructs. A transition  $P \xrightarrow{\alpha} Q$  denotes that process  $P$  may evolve to process  $Q$  by performing the action represented by the label  $\alpha$ . Transition labels are given by

$$\alpha ::= \overline{x\langle y \rangle} \mid x(y) \mid (\nu y)x\langle y \rangle \mid x.\text{inl} \mid x.\text{inr} \mid \overline{x.\text{inl}} \mid \overline{x.\text{inr}}$$

Basic actions are the input  $x(n)$ , the left/right offers  $x.\text{inl}$  and  $x.\text{inr}$ , and their matching co-actions, respectively the output  $x\langle y \rangle$  and bound output  $(\nu y)x\langle y \rangle$  actions, and the left/right selections  $\overline{x.\text{inl}}$  and  $\overline{x.\text{inr}}$ . The bound output action  $(\nu y)x\langle y \rangle$  denotes extrusion of a fresh name  $y$  along (channel)  $x$ . Internal action is represented by  $\tau$ , while in general an action  $\alpha(\bar{\alpha})$  requires a matching action  $\bar{\alpha}$  ( $\alpha$ ) in the environment to enable progress, as specified by the transition rules. For a label  $\alpha$ , we define the sets  $fn(\alpha)$  and  $bn(\alpha)$  of free and bound names, respectively, as usual. We denote by  $s(\alpha)$  the subject of  $\alpha$  (e.g.,  $x$  in  $x\langle y \rangle$ ).

$$\begin{array}{c}
\frac{P \xrightarrow{\alpha} Q}{(\nu y)P \xrightarrow{\alpha} (\nu y)Q} \text{(res)} \quad \frac{P \xrightarrow{\alpha} Q}{P \mid R \xrightarrow{\alpha} Q \mid R} \text{(par)} \quad \frac{P \xrightarrow{\alpha} P' \quad Q \xrightarrow{\alpha} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'} \text{(com)} \quad \frac{P \xrightarrow{(\nu y)x(y)} P' \quad Q \xrightarrow{x(y)} Q'}{P \mid Q \xrightarrow{\tau} (\nu y)(P' \mid Q')} \text{(close)} \\
\frac{P \xrightarrow{x(y)} Q}{(\nu y)P \xrightarrow{(\nu y)x(y)} Q} \text{(open)} \quad x(y).P \xrightarrow{x(y)} P \text{(out)} \quad x(y).P \xrightarrow{x(z)} P\{z/y\} \text{(in)} \quad !x(y).P \xrightarrow{x(z)} P\{z/y\} \mid !x(y).P \text{(rep)} \\
x.\text{inl}; P \xrightarrow{x.\text{inl}} P \text{(lout)} \quad x.\text{inr}; P \xrightarrow{x.\text{inr}} P \text{(rout)} \quad x.\text{case}(P, Q) \xrightarrow{x.\text{inl}} P \text{(lin)} \quad x.\text{case}(P, Q) \xrightarrow{x.\text{inr}} Q \text{(rin)}
\end{array}$$

Figure 1. Labeled Transition System.

**Definition 2.4 (Labeled Transition System)** *The relation of labeled transition ( $P \xrightarrow{\alpha} Q$ ) is defined by the rules in Figure 1, subject to the side conditions: in rule (res), we require  $y \notin \text{fn}(\alpha)$ ; in rule (par), we require  $\text{bn}(\alpha) \cap \text{fn}(R) = \emptyset$ ; in rule (close), we require  $y \notin \text{fn}(Q)$ . We omit the symmetric versions of rules (par), (com), and (close).*

We recall some basic facts relating reduction, structural congruence, and labeled transition, namely: closure of labeled transitions under structural congruence, and coincidence of  $\tau$ -labeled transition and reduction [22]. Given these facts, we will take processes up to structural congruence w.r.t. the reduction or labeled transition semantics.

**Lemma 2.5 (Harmony)** *We have*

1. Let  $P \equiv^{\alpha} Q$ . Then  $P \xrightarrow{\alpha} Q$ .
2.  $P \rightarrow Q$  if and only if  $P \xrightarrow{\tau} Q$ .

N.B.: we write  $R_1 R_2$  for relation composition (e.g.,  $\xrightarrow{\tau} \equiv$ ).

### 3 Type System

We first describe our type structure, which coincides with intuitionistic linear logic [14, 3], omitting atomic formulas and the usual additive constants  $\top$  and  $\mathbf{0}$ .

**Definition 3.1 (Types)** *Types ( $A, B, C$ ) are given by*

$$\begin{array}{l}
A, B ::= \mathbf{1} \quad | \quad !A \quad | \quad A \otimes B \\
\quad | \quad A \multimap B \quad | \quad A \oplus B \quad | \quad A \& B
\end{array}$$

Types are assigned to (channel) names, and may be conveniently interpreted as a form of session types; an assignment  $x : A$  enforces that the process will use  $x$  according to the discipline  $A$ .  $A \otimes B$  is the type of a session channel that first performs an output (sending a session channel of type  $A$ ) to its partner before proceeding as specified by  $B$ . In a similar way, the type  $A \multimap B$  types a session channel that first performs an input (receiving a session channel of type  $A$ ) from its partner, before proceeding as specified by  $B$ . The type  $\mathbf{1}$  means that the session terminated, no further interaction will take place on it. Notice that names of type

$\mathbf{1}$  may still be passed around in sessions, as opaque values.  $A \oplus B$  types a session that either selects “left” and then proceed as specified by  $A$ , or else selects “right”, and then proceeds as specified by  $B$ . Dually, the type  $A \& B$  types a session channel that offers its partner a choice between an  $A$  typed behavior (the “left” choice) and a  $B$  typed behavior (the “right” choice). The type  $!A$  types a non-session (non-linearized) channel (called *standard channel* in [13]), to be used by a server for spawning an arbitrary number of new sessions (possibly zero), each one conforming to type  $A$ .

A type environment is a collection of type assignments, of the form  $x : A$  where  $x$  is a name and  $A$  a type, the names being pairwise disjoint. Following the insights behind dual intuitionistic linear logic, which goes back to Andreoli’s *dyadic* system for classical linear logic [2], we distinguish two kinds of type environments subject to different structural properties: a *linear* part  $\Delta$  and an *unrestricted* part  $\Gamma$ , where weakening and contraction principles hold for  $\Gamma$  but not for  $\Delta$ . A judgment of our system has then the form

$$\Gamma; \Delta \vdash P :: z:C$$

where name declarations in  $\Gamma$  are always propagated unchanged to all premises in the typing rules, while name declarations in  $\Delta$  are handled multiplicatively or additively, depending on the nature of the type being defined. The domains of  $\Gamma, \Delta$  and  $z:C$  are required to be pairwise disjoint.

Intuitively, such a judgment asserts:  $P$  is ensured to safely provide a usage of name  $z$  according to the behavior (session) specified by type  $C$ , whenever composed by any process environment providing usages of names according to the behaviors specified by names in  $\Gamma; \Delta$ . As shown in Section 6, in our case safety ensures that the behavior is free of communication errors and deadlock.

We present the rules of our type system  $\pi\text{DILL}$  in Fig. 2. We use  $T, S$  for right hand side singleton environments (e.g.,  $z:C$ ). The interpretation of the various rules should be clear, given the explanation of types given above. Notice that since in  $\otimes R$  the sent name is always fresh, our typed calculus conforms to a session-based internal mobility discipline [21, 7], without loss of expressiveness (Section 7.1).

$$\begin{array}{c}
\frac{\Gamma; \Delta \vdash P :: T}{\overline{\Gamma; \Delta, x:\mathbf{1} \vdash P :: T}} \text{ (T1L)} \qquad \frac{}{\overline{\Gamma; \cdot \vdash 0 :: x:\mathbf{1}}} \text{ (T1R)} \\
\frac{\Gamma; \Delta, y:A, x:B \vdash P :: T}{\overline{\Gamma; \Delta, x:A \otimes B \vdash x(y).P :: T}} \text{ (T}\otimes\text{L)} \qquad \frac{\Gamma; \Delta \vdash P :: y:A \quad \Gamma; \Delta' \vdash Q :: x:B}{\overline{\Gamma; \Delta, \Delta' \vdash (\nu y)x\langle y \rangle.(P \mid Q) :: x:A \otimes B}} \text{ (T}\otimes\text{R)} \\
\frac{\Gamma; \Delta \vdash P :: y:A \quad \Gamma; \Delta', x:B \vdash Q :: T}{\overline{\Gamma; \Delta, \Delta', x:A \multimap B \vdash (\nu y)x\langle y \rangle.(P \mid Q) :: T}} \text{ (T}\multimap\text{L)} \qquad \frac{\Gamma; \Delta, y:A \vdash P :: x:B}{\overline{\Gamma; \Delta \vdash x(y).P :: x:A \multimap B}} \text{ (T}\multimap\text{R)} \\
\frac{\Gamma; \Delta \vdash P :: x:A \quad \Gamma; \Delta', x:A \vdash Q :: T}{\overline{\Gamma; \Delta, \Delta' \vdash (\nu x)(P \mid Q) :: T}} \text{ (Tcut)} \qquad \frac{\Gamma; \cdot \vdash P :: y:A \quad \Gamma, u:A; \Delta \vdash Q :: T}{\overline{\Gamma; \Delta \vdash (\nu u)(!u(y).P \mid Q) :: T}} \text{ (Tcut}^!\text{)} \\
\frac{\Gamma, u:A; \Delta, y:A \vdash P :: T}{\overline{\Gamma, u:A; \Delta \vdash (\nu y)u\langle y \rangle.P :: T}} \text{ (Tcopy)} \qquad \frac{\Gamma; \cdot \vdash Q :: y:A}{\overline{\Gamma; \cdot \vdash !x(y).Q :: x:A}} \text{ (T!R)} \qquad \frac{\Gamma, u:A; \Delta \vdash P\{u/x\} :: T}{\overline{\Gamma; \Delta, x:A \vdash P :: T}} \text{ (T!L)} \\
\frac{\Gamma; \Delta, x:A \vdash P :: T}{\overline{\Gamma; \Delta, x:A \& B \vdash x.\text{inl}; P :: T}} \text{ (T}\&\text{L}_1\text{)} \qquad \frac{\Gamma; \Delta, x:B \vdash P :: T}{\overline{\Gamma; \Delta, x:A \& B \vdash x.\text{inr}; P :: T}} \text{ (T}\&\text{L}_2\text{)} \\
\frac{\Gamma; \Delta \vdash P :: x:A \quad \Gamma; \Delta \vdash Q :: x:B}{\overline{\Gamma; \Delta \vdash x.\text{case}(P, Q) :: x:A \& B}} \text{ (T}\&\text{R)} \qquad \frac{\Gamma; \Delta, x:A \vdash P :: T \quad \Gamma; \Delta, x:B \vdash Q :: T}{\overline{\Gamma; \Delta, x:A \oplus B \vdash x.\text{case}(P, Q) :: T}} \text{ (T}\oplus\text{L)} \\
\frac{\Gamma; \Delta \vdash P :: x:A}{\overline{\Gamma; \Delta \vdash x.\text{inl}; P :: x:A \oplus B}} \text{ (T}\oplus\text{R}_1\text{)} \qquad \frac{\Gamma; \Delta \vdash P :: x:B}{\overline{\Gamma; \Delta \vdash x.\text{inr}; P :: x:A \oplus B}} \text{ (T}\oplus\text{R}_2\text{)}
\end{array}$$

**Figure 2. The Type System  $\pi\text{DILL}$ .**

The composition rules (cut and cut<sup>!</sup>) follow the “composition plus hiding” principle [1], extended to a name passing setting. More familiar linear typing rules for parallel composition (e.g., as in [19]) are derivable (Section 7.1).

Since we are considering  $\pi$ -calculus terms up to structural congruence, typability is closed under  $\equiv$  by definition.  $\pi\text{DILL}$  enjoys the usual properties of equivariance, weakening in  $\Gamma$  and contraction in  $\Gamma$ . The coverage property also holds: if  $\Gamma; \Delta \vdash P :: z:A$  then  $\text{fn}(P) \subseteq \Gamma \cup \Delta \cup \{z\}$ . In the presence of type-annotated restrictions  $(\nu x:A)P$ , as usual in typed  $\pi$ -calculi [22], type-checking is decidable.

We illustrate the type system with a simple example, frequently used to motivate session based interactions (see e.g., [13]). A client may choose between a “buy” operation, in which it indicates a product name and a credit card number to receive a receipt, and a “quote” operation, in which it indicates a product name, to obtain the product price. From the client perspective, the session protocol exposed by the server may be specified by the type

$$\text{ServerProto} \triangleq (N \multimap I \multimap (N \otimes \mathbf{1})) \& (N \multimap (I \otimes \mathbf{1}))$$

We assume that  $N$  and  $I$  are types representing shareable values (e.g., strings  $N$  and integers  $I$ ). To simplify, we set  $N = I = \mathbf{1}$ . Assuming  $s$  to be the name of the session channel connecting the client and server, consider the code

$$\text{QClntBody}_s \triangleq s.\text{inr}; (\nu \text{tea})s\langle \text{tea} \rangle.s\langle \text{pr} \rangle.\mathbf{0}$$

$\text{QClntBody}_s$  specifies a client that asks for the price of tea (we simply abstract away from what the client might do with the price after reading it). It first selects the quoting

operation on the server ( $s.\text{inr}$ ), then sends the  $id$  of the product to the server ( $s\langle \text{tea} \rangle$ ), then receives the price  $s\langle \text{pr} \rangle$  from the server and finally terminates the session ( $\mathbf{0}$ ). Then

$$\cdot; s : \text{ServerProto} \vdash \text{QClntBody}_s :: -:\mathbf{1}$$

is derivable (by  $T\mathbf{1R}$ ,  $T\otimes\text{L}$ ,  $T\multimap\text{L}$  and  $T\&\text{L}_2$ ). Here we wrote  $-$  for an anonymous variable that does not appear in  $\text{QClntBody}_s$ . This is possible even in a linear type discipline since the inactive process  $\mathbf{0}$  is typed by  $x:\mathbf{1}$  and does not use  $x$ . Concerning the server code, let

$$\text{SrvBody}_s \triangleq s.\text{case}(s\langle \text{pn} \rangle.s\langle \text{cn} \rangle.(\nu \text{rc})s\langle \text{rc} \rangle.\mathbf{0}, s\langle \text{pn} \rangle.(\nu \text{pr})s\langle \text{pr} \rangle.\mathbf{0})$$

Then  $\cdot; \cdot \vdash \text{SrvBody}_s :: s:\text{ServerProto}$  is derivable, by  $T\&\text{R}$ . By  $T\text{cut}$  we obtain for the composition

$$\text{QSimple} \triangleq (\nu s)(\text{SrvBody}_s \mid \text{QClntBody}_s)$$

the typing  $\cdot; \cdot \vdash \text{QSimple} :: -:\mathbf{1}$ .

## 4 Dual Intuitionistic Linear Logic

Our type constructors correspond directly to linear logic connectives. Typing judgments directly correspond to sequents in dual intuitionistic linear logic, by erasing processes [3, 10]. In Figure 3 we present the  $\text{DILL}$  sequent calculus. In our presentation,  $\text{DILL}$  is conveniently equipped with a faithful proof term assignment, so sequents have the form  $\Gamma; \Delta \vdash D : C$  where  $\Gamma$  is the unrestricted context,  $\Delta$  the linear context,  $C$  a formula (= type) and  $D$  the proof term that faithfully represents the derivation of  $\Gamma; \Delta \vdash C$ . Our use of names in the proof system will be consistent with

$$\begin{array}{c}
\frac{\Gamma; \Delta \vdash D : C}{\Gamma; \Delta, x : \mathbf{1} \vdash \mathbf{1L} x D : C} \text{ (1L)} \qquad \frac{}{\Gamma; \cdot \vdash \mathbf{1R} : \mathbf{1}} \text{ (1R)} \\
\frac{\Gamma; \Delta, y : A, x : B \vdash D : C}{\Gamma; \Delta, x : A \otimes B \vdash \otimes L x (y.x. D) : C} \text{ (\otimes L)} \qquad \frac{\Gamma; \Delta \vdash D : A \quad \Gamma; \Delta' \vdash E : B}{\Gamma; \Delta, \Delta' \vdash \otimes R D E : A \otimes B} \text{ (\otimes R)} \\
\frac{\Gamma; \Delta \vdash D : A \quad \Gamma; \Delta', x : B \vdash E : C}{\Gamma; \Delta, \Delta', x : A \multimap B \vdash \multimap L x D (x. E) :: C} \text{ (\multimap L)} \qquad \frac{\Gamma; \Delta, y : A \vdash D : B}{\Gamma; \Delta \vdash \multimap R (y. D) : A \multimap B} \text{ (\multimap R)} \\
\frac{\Gamma; \Delta \vdash D : A \quad \Gamma; \Delta', x : A \vdash E : C}{\Gamma; \Delta, \Delta' \vdash \text{cut} D (x. E) : C} \text{ (cut)} \qquad \frac{\Gamma; \cdot \vdash D : A \quad \Gamma, u : A; \Delta \vdash E : C}{\Gamma; \Delta \vdash \text{cut}^! D (u. E) : C} \text{ (cut}^!\text{)} \\
\frac{\Gamma, u : A; \Delta, y : A \vdash D : C}{\Gamma, u : A; \Delta \vdash \text{copy} u (y. D) : C} \text{ (copy)} \qquad \frac{\Gamma; \cdot \vdash D : A}{\Gamma; \cdot \vdash !R D : !A} \text{ (!R)} \qquad \frac{\Gamma, u : A; \Delta \vdash D : C}{\Gamma; \Delta, x : !A \vdash !L x (u.D) : C} \text{ (!L)} \\
\frac{\Gamma; \Delta, x : A \vdash D : C}{\Gamma; \Delta, x : A \& B \vdash \& L_1 x (x. D) : C} \text{ (\&L}_1\text{)} \qquad \frac{\Gamma; \Delta, x : B \vdash D : C}{\Gamma; \Delta, x : A \& B \vdash \& L_2 x (x. D) : C} \text{ (\&L}_2\text{)} \\
\frac{\Gamma; \Delta \vdash D : A \quad \Gamma; \Delta \vdash E : B}{\Gamma; \Delta \vdash \& R D E : A \& B} \text{ (\&R)} \qquad \frac{\Gamma; \Delta x : A \vdash D : C \quad \Gamma; \Delta, x : B \vdash E : C}{\Gamma; \Delta, x : A \oplus B \vdash \oplus L x (x. D) (x. E) : C} \text{ (\oplus L)} \\
\frac{\Gamma; \Delta \vdash D : A}{\Gamma; \Delta \vdash \oplus R_1 D : A \oplus B} \text{ (\oplus R}_1\text{)} \qquad \frac{\Gamma; \Delta \vdash D : B}{\Gamma; \Delta \vdash \oplus R_2 D : A \oplus B} \text{ (\oplus R}_2\text{)}
\end{array}$$

**Figure 3. Dual Intuitionistic Linear Logic** DILL.

the proof discipline,  $u, v, w$  for variables in  $\Gamma$  and  $x, y, z$  for variables in  $\Delta$ . This is consistent with standard usage of names in  $\pi$ -calculi. Given the parallel structure of the two systems, if  $\Gamma; \Delta \vdash D : A$  is derivable in DILL then there is a process  $P$  and a name  $z$  such that  $\Gamma; \Delta \vdash P :: z : A$  is derivable in  $\pi$ DILL, and the converse result also holds: if  $\Gamma; \Delta \vdash P :: z : A$  is derivable in  $\pi$ DILL there is a derivation  $D$  that proves  $\Gamma; \Delta \vdash D : A$ . This correspondence is made explicit by a translation from faithful proof terms to processes, defined in Figure 4: for  $\Delta \vdash D : C$  we write  $\hat{D}^z$  for the translation of  $D$  such that  $\Delta \vdash \hat{D}^z :: z : C$ .

**Definition 4.1 (Typed Extraction)** *We write*

$$\Gamma; \Delta \vdash D \rightsquigarrow P :: z : A \quad (\text{proof } D \text{ extracts to } P)$$

whenever  $\Gamma; \Delta \vdash D :: A$  and  $\Gamma; \Delta \vdash \hat{D}^z :: z : A$ .

Typed extraction is unique up to structural congruence, in the sense that if  $\Gamma; \Delta \vdash D \rightsquigarrow P :: z : A$  and  $\Gamma; \Delta \vdash D \rightsquigarrow Q :: z : A$  then  $P \equiv Q$ , as a consequence of closure of typing under structural congruence. The system DILL as presented does not admit atomic formulas, and hence has no true initial sequents. However, the correspondence mentioned above yields an explicit identity theorem:

**Proposition 4.2** *For any type  $A$  and distinct names  $x, y$ , there is a process  $id_A(x, y)$  and a cut-free derivation  $D$  such that  $\cdot; x : A \vdash D \rightsquigarrow id_A(x, y) :: y : A$ .*

The  $id_A(x, y)$  process, with exactly the free names  $x, y$ , implements a synchronous mediator that bidirectionally plays the protocol specified by  $A$  between channels  $x$  and

$y$ . As a small example, we analyze the interpretation of the sequent  $A \otimes B \vdash B \otimes A$ . We have

$$x:A \otimes B \vdash F \rightsquigarrow x(z).(v n)y \langle n \rangle. (P \mid Q) :: y:B \otimes A$$

where  $F = \otimes L x (z.x. \otimes R D E)$ ,  $P = id_B(x, n)$  and  $Q = id_A(z, y)$ . This process is an interactive proxy that coerces a session of type  $A \otimes B$  at  $x$  to a session of type  $B \otimes A$  at  $y$ . It first receives a session of type  $A$  (bound to  $z$ ) and after sending on  $y$  a session of type  $B$  (played by copying the continuation of  $x$  to  $n$ ), it progresses with a session of type  $A$  on  $y$  (copying the continuation of  $z$  to  $y$ ).

As processes are related by structural and computational rules, namely those involved in the definition of  $\equiv$  and  $\rightarrow$ , derivations in DILL are related by structural and computational rules, that express certain sound proof transformations that arise in cut-elimination. The reductions (Figure 5) generally take place when a right rule meets a left rule for the same connective, and correspond to reduction steps in the process term assignment. On the left, we show the usual reductions for cuts; on the right, we show the corresponding reductions (if any) of the process terms, modulo structural congruence. Since equivalences depend on occurrences of variables, we write  $D_x$  if  $x$  may occur in  $D$ .

The structural conversions in Figure 6 correspond to structural equivalences in the  $\pi$ -calculus, since they just change the order of cuts, e.g.,  $(\text{cut}/-\text{cut}_1)$  translates to

$$(\nu x)(\hat{D}^x \mid (\nu y)(\hat{E}^y \mid \hat{F}^z)) \equiv (\nu y)((\nu x)(\hat{D}^x \mid \hat{E}^y) \mid \hat{F}^z)$$

In addition, we have two special conversions. Among those,  $(\text{cut}/\mathbf{1R}/\mathbf{1L})$  is not needed in order to simulate the

$D$	$\rightsquigarrow$	$\hat{D}^z$
$1R$	$\rightsquigarrow$	$0$
$1L \ x \ D$	$\rightsquigarrow$	$\hat{D}^z$
$\otimes R \ D \ E$	$\rightsquigarrow$	$(\nu y) z \langle y \rangle. (\hat{D}^y \mid \hat{E}^z)$
$\otimes L \ x \ (y.x.D)$	$\rightsquigarrow$	$x(y). \hat{D}^z$
$\multimap R \ (y.D)$	$\rightsquigarrow$	$z(y). \hat{D}^z$
$\multimap L \ x \ D \ (x.E)$	$\rightsquigarrow$	$(\nu y) x \langle y \rangle. (\hat{D}^y \mid \hat{E}^z)$
$\& R \ D \ E$	$\rightsquigarrow$	$z. \text{case}(\hat{D}^z, \hat{E}^z)$
$\& L_1 \ x \ (x.D)$	$\rightsquigarrow$	$x. \text{inl}; \hat{D}^z$
$\& L_2 \ x \ (x.E)$	$\rightsquigarrow$	$x. \text{inr}; \hat{E}^z$
$\oplus R_1 \ D$	$\rightsquigarrow$	$z. \text{inl}; \hat{D}^z$
$\oplus R_2 \ E$	$\rightsquigarrow$	$z. \text{inr}; \hat{E}^z$
$\oplus L \ x \ (x.D) \ (x.E)$	$\rightsquigarrow$	$x. \text{case}(\hat{D}^z, \hat{E}^z)$
$\text{cut} \ D \ (x.E)$	$\rightsquigarrow$	$(\nu x)(\hat{D}^x \mid \hat{E}^z)$
$!R \ D$	$\rightsquigarrow$	$!z(y). \hat{D}^y$
$!L \ x \ (u.D)$	$\rightsquigarrow$	$\hat{D}^z \{x/u\}$
$\text{copy} \ u \ (y.D)$	$\rightsquigarrow$	$(\nu y) u \langle y \rangle. \hat{D}^z$
$\text{cut}^! \ D \ (u.E)$	$\rightsquigarrow$	$(\nu u)((!u(y). \hat{D}^y) \mid \hat{E}^z)$

**Figure 4. Proof  $D$  extracts to process  $\hat{D}^z$ .**

$\pi$ -calculus reduction, while  $(\text{cut}/!R/!L)$  is. During cut-elimination procedures, these are always used from left to right. Here, they are listed as equivalences because the corresponding  $\pi$ -calculus terms are structurally congruent. The root cause for this is that the rules  $1L$  and  $!L$  are *silent*: the extracted terms in the premise and conclusion are the same, modulo renaming. For  $1L$ , this is the case because a terminated process, represented by  $0 :: - : 1$  silently disappears from a parallel composition by structural congruence. For  $!L$ , this is the case because the actual replication of a server process is captured in the copy rule which clones  $u:A$  to  $y:A$ , rule rather than  $!L$ . It is precisely for this reason that the rule commuting a persistent cut ( $\text{cut}^!$ ) over a copy rule ( $\text{copy}$ ) is among the computational conversions.

The structural conversions in Figure 8 propagate  $\text{cut}^!$ . From the proof theoretic perspective, because  $\text{cut}^!$  cuts a persistent variable  $u$ ,  $\text{cut}^!$  may be duplicated or erased. On the  $\pi$ -calculus side, these no longer correspond to structural congruences, but, quite remarkably, to behavioral equivalences, derivable from known properties of typed processes, the (sharpened) Replication Theorems [22], which hold in our language, due to the interpretation of  $!$  types.

Our operational correspondence results also depend on six commuting conversions (Figure 7). The commuting conversions push a cut up (or inside) the  $1L$  and  $!L$  rules. During the usual cut elimination procedures, these are used from left to right. In the correspondence with the sequent calculus, the situation is more complex. Because the  $1L$  and  $!L$  rules do not affect the extracted term, cuts have to be per-

mutated with these two rules in order to simulate  $\pi$ -calculus reduction. From the process calculus perspective, such conversions correspond to identity. There is a second group of commuting conversions (not shown), not necessary for our current development. Those do not correspond to structural congruence nor to strong bisimilarities on  $\pi$ -calculus, as they may not preserve process behavior in the general untyped setting, since they promote an action prefix from a subexpression to the top level. We conjecture that such equations denote behavioral identities under a natural definition of typed observational congruence for our calculus.

We denote by  $\equiv$  the least congruence generated by the structural conversions (I) and commuting conversions (II), and by  $\simeq_s$  the least congruence generated by all structural conversions (I-III). We note by  $\Rightarrow$  the reduction obtained by orienting all conversions in the direction shown in the figures, from left to right or top to bottom. We note by  $\simeq_s$  the congruence generated by  $\equiv$  and  $\simeq$  on typed processes; as referred above  $\simeq_s$  expresses typed behavioral equivalences.

## 5 Computational Correspondence

Theorem 5.3 states the existence of a simulation between reductions in the typed  $\pi$ -calculus and proof conversions / reductions. The proof relies on several auxiliary lemmas, which we mostly omit, among them a sequence of reduction lemmas from which we select two typical examples.

**Lemma 5.1** *Assume*

- (a)  $\Gamma; \Delta_1 \vdash D \rightsquigarrow P :: x:A_1 \otimes A_2$  with  $P \xrightarrow{(\nu y)x \langle y \rangle} P'$ ;
  - (b)  $\Gamma; \Delta_2, x:A_1 \otimes A_2 \vdash E \rightsquigarrow Q :: z:C$  with  $Q \xrightarrow{x \langle y \rangle} Q'$ .
- Then*
- (c)  $\text{cut} \ D \ (x.E) \Rightarrow \equiv F$  for some  $F$ ;
  - (d)  $\Gamma; \Delta_1, \Delta_2 \vdash F \rightsquigarrow R :: z : C$  for  $R \equiv (\nu x)(P' \mid Q')$ .

**Lemma 5.2** *Assume*

- (a)  $\Gamma; \cdot \vdash D \rightsquigarrow P :: x:A$ ;
  - (b)  $\Gamma, u:A; \Delta_2 \vdash E \rightsquigarrow Q :: z:C$  with  $Q \xrightarrow{(\nu y)u \langle y \rangle} Q'$ .
- Then*
- (a)  $\text{cut}^! \ D \ (u.E) \Rightarrow \equiv F$  for some  $F$ ;
  - (b)  $\Gamma; \Delta \vdash F \rightsquigarrow R :: z : C$   
for some  $R \equiv (\nu u)(!u(x).P \mid (\nu y)(P\{y/x\} \mid Q'))$ .

**Theorem 5.3** *Let  $\Gamma; \Delta \vdash D \rightsquigarrow P :: z:A$  and  $P \rightarrow Q$ . Then there is  $E$  such that  $D \Rightarrow \equiv E$  and  $\Gamma; \Delta \vdash E \rightsquigarrow Q :: z:A$*

*Proof.* By induction on the structure of derivation  $D$ . The possible cases for  $D$  are  $D = 1L \ y \ D'$ ,  $D = !L \ x \ (u.D')$ ,  $D = \text{cut} \ D_1 \ (x.D_2)$ , and  $D = \text{cut}^! \ D_1 \ (x.D_2)$ , in all other cases  $P \not\rightarrow$ . The key cases are the cuts, where we rely on a series of reduction lemmas, one for each type  $C$  of cut formula, which assign certain proof conversions to process labeled transitions. For example, for  $C = C_1 \otimes C_2$ , we rely

(cut/⊗R/⊗L)	$\text{cut} (\otimes R D_1 D_2) (x. \otimes L x (y.x. E))$ $\Rightarrow$ $\text{cut} D_1 (y. \text{cut} D_2 (x. E))$	$\rightsquigarrow$ $\xrightarrow{\sim}$ $(\nu x)((\nu y) x \langle y \rangle. (\hat{D}_1^y \mid \hat{D}_2^x)) \mid x \langle y \rangle. \hat{E}^z$ $\rightsquigarrow$ $(\nu x)(\nu y)(\hat{D}_1^y \mid \hat{D}_2^x \mid \hat{E}^z)$
(cut/¬oR/¬oL)	$\text{cut} (\neg o R (y. D)) (x. \neg o L x E_1 (x. E_2))$ $\Rightarrow$ $\text{cut} (\text{cut} E_1 (y. D)) (x. E_2)$	$\rightsquigarrow$ $\xrightarrow{\sim}$ $(\nu x)((x \langle y \rangle. \hat{D}^x) \mid (\nu y) x \langle y \rangle. (\hat{E}_1^y \mid \hat{E}_2^z))$ $\rightsquigarrow$ $(\nu x)(\nu y)(\hat{D}^x \mid \hat{E}_1^y \mid \hat{E}_2^z)$
(cut/&R/&L <sub>1</sub> )	$\text{cut} (\& R D_1 D_2) (x. \& L_1 x (x. E_1))$ $\Rightarrow$ $\text{cut} D_1 (x. E_1)$	$\rightsquigarrow$ $\xrightarrow{\sim}$ $(\nu x)(x. \text{case}(\hat{D}_1^x, \hat{D}_2^x) \mid x. \text{inl}; \hat{E}_1^z)$ $\rightsquigarrow$ $(\nu x)(\hat{D}_1^x \mid \hat{E}_1^z)$
(cut/&R/&L <sub>2</sub> )	$\text{cut} (\& R D_1 D_2) (x. \& L_2 x (x. E_2))$ $\Rightarrow$ $\text{cut} D_2 (x. E_2)$	$\rightsquigarrow$ $\xrightarrow{\sim}$ $(\nu x)(x. \text{case}(\hat{D}_1^x, \hat{D}_2^x) \mid x. \text{inr}; \hat{E}_2^z)$ $\rightsquigarrow$ $(\nu x)(\hat{D}_2^x \mid \hat{E}_2^z)$
(cut/⊕R <sub>1</sub> /⊕L)	$\text{cut} (\oplus R_1 D_1) (x. \oplus L x (x. E_1) (x. E_2))$ $\Rightarrow$ $\text{cut} D_1 (x. E_1)$	$\rightsquigarrow$ $\xrightarrow{\sim}$ $(\nu x)(x. \text{inl}; \hat{D}_1^x \mid x. \text{case}(\hat{E}_1^z, \hat{E}_2^z))$ $\rightsquigarrow$ $(\nu x)(\hat{D}_1^x \mid \hat{E}_1^z)$
(cut/⊕R <sub>2</sub> /⊕L)	$\text{cut} (\oplus R_2 D_2) (x. \oplus L x (x. E_1) (x. E_2))$ $\Rightarrow$ $\text{cut} D_2 (x. E_2)$	$\rightsquigarrow$ $\xrightarrow{\sim}$ $(\nu x)(x. \text{inr}; \hat{D}_2^x \mid x. \text{case}(\hat{E}_1^z, \hat{E}_2^z))$ $\rightsquigarrow$ $(\nu x)(\hat{D}_2^x \mid \hat{E}_2^z)$
(cut <sup>!</sup> /-/copy)	$\text{cut}^! D (u. \text{copy } u (y. E_{uy}))$ $\Rightarrow$ $\text{cut} D (y. \text{cut}^! D (u. E_{uy}))$	$\rightsquigarrow$ $\xrightarrow{\sim}$ $(\nu u)((!u \langle y \rangle. \hat{D}^y) \mid (\nu y) u \langle y \rangle. \hat{E}^z)$ $\rightsquigarrow$ $(\nu y)(\hat{D}^y \mid (\nu u)((!u \langle y \rangle. \hat{D}^y) \mid \hat{E}^z))$

**Figure 5. Computational Conversions.**

(cut/-/cut <sub>1</sub> )	$\text{cut} D (x. \text{cut} E_x (y. F_y))$	$\equiv$	$\text{cut} (\text{cut} D (x. E_x)) (y. F_y)$
(cut/-/cut <sub>2</sub> )	$\text{cut} D (x. \text{cut} E (y. F_{xy}))$	$\equiv$	$\text{cut} E (y. \text{cut} D (x. F_{xy}))$
(cut/cut <sup>!</sup> /-)	$\text{cut} (\text{cut}^! D (u. E_u)) (x. F_x)$	$\equiv$	$\text{cut}^! D (u. \text{cut} E_u (x. F_x))$
(cut/-/cut <sup>!</sup> )	$\text{cut} D (x. \text{cut}^! E (u. F_{xu}))$	$\equiv$	$(\text{cut}^! E (u. \text{cut} D (x. F_{xu})))$
(cut/1R/1L)	$\text{cut} 1R (x. 1L x D)$	$\equiv$	$D$
(cut!/R!/L)	$\text{cut} (!R D) (x. !L x (u. E))$	$\equiv$	$\text{cut}^! D (u. E)$

**Figure 6. Structural Conversions (I): Cut Conversions.**

(cut/-/1L)	$\text{cut} D (x. 1L y E_x)$	$\equiv$	$1L y (\text{cut} D (x. E_x))$
(cut/-!/L)	$\text{cut} D (x. !L y (u. E_{xu}))$	$\equiv$	$!L y (u. \text{cut} D (x. E_{xu}))$
(cut/1L/-)	$\text{cut} (1L y D) (x. F_x)$	$\equiv$	$1L y (\text{cut} D (x. F_x))$
(cut!/L/-)	$\text{cut} (!L y (u. D_u)) (x. F_x)$	$\equiv$	$!L y (u. \text{cut} D_u (x. F_x))$
(cut <sup>!</sup> /-/1L)	$\text{cut}^! D (u. 1L y E_u)$	$\equiv$	$1L y (\text{cut}^! D (u. E_u))$
(cut <sup>!</sup> /-/!L)	$\text{cut}^! D (u. !L y (v. E_{uv}))$	$\equiv$	$!L y (v. \text{cut}^! D (u. E_{uv}))$

**Figure 7. Structural Conversions (II): Commuting Conversions**

(cut <sup>!</sup> /-/cut)	$\text{cut}^! D (u. \text{cut} E_u (y. F_{uy}))$ $\equiv$ $\text{cut} (\text{cut}^! D (u. E_u)) (y. \text{cut}^! D (u. F_{uy}))$	$\rightsquigarrow$ $\xrightarrow{\sim}$ $(\nu u)(!u \langle y \rangle. \hat{D}^y \mid (\nu y)(\hat{E}^y \mid \hat{F}^z))$ $\rightsquigarrow$ $(\nu y)((\nu u)(!u \langle y \rangle. \hat{D}^y \mid \hat{E}^y) \mid (\nu u)(!u \langle y \rangle. \hat{D}^y \mid \hat{F}^z))$
(cut <sup>!</sup> /-/cut <sup>!</sup> )	$\text{cut}^! D (u. \text{cut}^! E_u (v. F_{uv}))$ $\equiv$ $\text{cut}^! (\text{cut}^! D (u. E_u)) (v. \text{cut}^! D (u. F_{uv}))$	$\rightsquigarrow$ $\xrightarrow{\sim}$ $(\nu u)(!u \langle y \rangle. \hat{D}^y \mid (\nu v)(!v \langle y \rangle. \hat{E}^y \mid \hat{F}^z))$ $\rightsquigarrow$ $(\nu v)((!v \langle y \rangle. (\nu u)(!u \langle y \rangle. \hat{D}^y \mid \hat{E}^y)) \mid (\nu u)(!u \langle y \rangle. \hat{D}^y \mid \hat{F}^z))$
(cut <sup>!</sup> /cut <sup>!</sup> /-)	$\text{cut}^! (\text{cut}^! D (u. E_u)) (v. F_v)$ $\equiv$ $\text{cut}^! D (u. \text{cut}^! E_u (v. F_v))$	$\rightsquigarrow$ $\xrightarrow{\sim}$ $(\nu v)(!v \langle y \rangle. (\nu u)(!u \langle y \rangle. \hat{D}^y \mid \hat{E}^y)) \mid F^z$ $\rightsquigarrow$ $(\nu u)(!u \langle y \rangle. \hat{D}^y \mid (\nu v)(!v \langle y \rangle. \hat{E}^y \mid \hat{F}^z))$
(cut <sup>!</sup> /-/- <sub>0</sub> )	$\text{cut}^! D (u. E)$ $\equiv$ $E$	$\rightsquigarrow$ $\xrightarrow{\sim}$ $(\nu u)(!u \langle y \rangle. \hat{D}^y \mid \hat{E}^z)$ $\rightsquigarrow$ $\hat{E}^z$ (for $u \notin FN(\hat{E}^z)$ )

**Figure 8. Structural Conversions (III): Cut! Conversions.**

on Lemma 5.1. The case of  $\text{cut}^!$ , similar to the case  $C = !C'$ , relies on Lemma 5.2. We show such case in detail. Let  $D = \text{cut}^! D_1 (u. D_2)$ . We have  $P \equiv (\nu u)(!u(w).P_1 \mid P_2)$ ,  $\Gamma; \vdash D_1 \rightsquigarrow P_1 :: x:C$ , and  $\Gamma, u : C; \Delta \vdash D_2 \rightsquigarrow P_2 :: z:A$  by inversion. Since  $P \rightarrow Q$ , there two cases: (1)  $P_2 \rightarrow Q_2$  and  $Q = (\nu u)(!u(w).P_1 \mid Q_2)$ , or (2)  $P_2 \xrightarrow{\alpha} Q_2$  where  $\alpha = \overline{(\nu y)x(y)}$  and  $Q = (\nu u)(!u(w).P_1 \mid (\nu y)(P_1\{y/x\} \mid Q_2))$ .

Case (1): We have  $\Gamma, u : C; \Delta \vdash D_2 \rightsquigarrow Q_2 :: z:A$  for  $E'$  with  $D_2 \equiv \equiv E'$  by i.h. Then  $\text{cut}^! D_1 (u. D_2) \equiv \equiv \text{cut}^! D_1 (u. E')$  by congruence. Let  $E = \text{cut}^! D_1 (u. E')$ . So  $\Gamma; \Delta \vdash E \rightsquigarrow Q :: z:A$  by  $\text{cut}^!$ .

Case (2): By Lemma 5.2,  $\text{cut}^! D_1 (u. D_2) \equiv \equiv E$  for some  $E$ , and  $\Gamma; \Delta \vdash E \rightsquigarrow R :: z:A$  with  $R \equiv Q$ .  $\square$

Theorems 5.4 and 5.5 state that any proof reduction or conversion also corresponds to either a process equivalence or to a reduction step on the  $\pi$ -calculus.

**Theorem 5.4** *Let  $\Gamma; \Delta \vdash D \rightsquigarrow P :: z:A$  and  $D \simeq_s E$ . Then there is  $Q$  where  $P \simeq_s Q$  and  $\Gamma; \Delta \vdash E \rightsquigarrow Q :: z:A$ .*

*Proof.* By a diagram chase using the commuting squares relating  $\equiv$ ,  $\rightsquigarrow$  and  $\simeq$  in Figures 6, 7 and 8.  $\square$

**Theorem 5.5** *Let  $\Gamma; \Delta \vdash D \rightsquigarrow P :: z:A$  and  $D \Rightarrow E$ . Then there is  $Q$  such that  $P \rightarrow Q$  and  $\Gamma; \Delta \vdash E \rightsquigarrow Q :: z:A$ .*

*Proof.* By a diagram chase using the commuting squares relating  $\Rightarrow$ ,  $\rightsquigarrow$  and  $\rightarrow$  in Figure 5.  $\square$

Notice that the simulation of  $\pi$ -calculus reductions by proof term conversions provided by Theorem 5.3, and from which subject reduction follows, is very tight indeed, as reduction is simulated up to structural congruence, which is a very fine equivalence on processes. To that end, structural conversions need to be applied symmetrically (as equations), unlike in a standard proof of cut-elimination, where they are usually considered as directed computational steps. Under the assumptions of Theorem 5.3, we can also prove that there is  $E$  such that  $D \Rightarrow E$  and  $\Gamma; \Delta \vdash E \rightsquigarrow R :: z:A$ , for  $Q \simeq_s R$ . Thus, even if one considers the proof conversions as directed reduction rules ( $\Rightarrow$ ), we still obtain a sound simulation up to typed strong behavioral congruence.

## 6 Type Preservation and Progress

We state type preservation and progress results. The subject reduction property directly follows from Theorem 5.3.

**Theorem 6.1 (Subject Reduction)** *If  $\Gamma; \Delta \vdash P :: z:A$  and  $P \rightarrow Q$  then  $\Gamma; \Delta \vdash Q :: z:A$ .*

Together with direct consequences of linear typing, Theorem 6.1 ensures session fidelity. Our type discipline also enforces a global progress property. For any  $P$ , define

$$\text{live}(P) \text{ iff } P \equiv (\nu \bar{n})(\pi.Q \mid R) \text{ for some } \pi.Q, R, \bar{n}$$

where  $\pi.Q$  is a *non-replicated* guarded process. We first establish the following contextual progress property.

**Lemma 6.2** *Let  $\Gamma; \Delta \vdash D \rightsquigarrow P :: z:C$ . If  $\text{live}(P)$  then there is  $Q$  such that either*

1.  $P \rightarrow Q$ , or
2.  $P \xrightarrow{\alpha} Q$  for some  $\alpha$  where  $s(\alpha) \in (z, \Gamma, \Delta)$ . Moreover, if  $C = !A$  for some  $A$  then  $s(\alpha) \neq z$ .

*Proof.* Induction on the structure of derivation  $D$ . The key cases are  $D = \text{cut} D_1 (y. D_2)$  and  $D = \text{cut}^! D_1 (u. D_2)$ . There, we rely on auxiliary lemmas that characterize the shape of the action the process can do on cut name  $x:C$ , depending on the type  $C$ , to show that a synchronization between dual actions takes place. For  $\text{cut}^!$ , an inversion lemma is needed, stating that the free names of a non-live process can only be typed by  $\mathbf{1}$  or  $!A$  types, allowing the induction to go through while finding a relevant transition (satisfying the second part of 2.).  $\square$

**Theorem 6.3 (Progress)** *Let  $\cdot; \cdot \vdash D \rightsquigarrow P :: x:\mathbf{1}$ . If  $\text{live}(P)$  then there is  $Q$  such that  $P \rightarrow Q$ .*

*Proof.* By Lemma 6.2 and the fact that  $P$  cannot perform any action  $\alpha$  with subject  $s(\alpha) = x$  since  $x:\mathbf{1}$  (by the action shape characterization lemmas).  $\square$

## 7 Discussion and Examples

### 7.1 Session Types

We further compare our linear type system for (finite) session types with more familiar type systems for sessions. An immediate observation is that our types are freely generated, while in [19, 16, 13] there is a stratification of types in “session” and “standard types” (the later corresponding to our  $!A$  types, typing session initiation channels). In our interpretation a session may terminate ( $\mathbf{1}$ ), or become a replicated server ( $!A$ ), which is more general and uniform. In [19, 16, 13] two composition rules can be found, one corresponding to the cancellation of two dual session endpoints (a restriction rule), and another corresponding to independent parallel composition, also found in most linear type systems for mobile processes. In our case, cut combines both principles, and the following rule is derivable:

$$\frac{\Gamma; \Delta \vdash P :: x : \mathbf{1} \quad \Gamma; \Delta' \vdash Q :: T}{\Gamma; \Delta, \Delta' \vdash (P \mid Q) :: T} \text{ (comp)}$$

A consequence of our logical composition rules  $\text{cut}$  and  $\text{cut}^!$  is that typing enforces a global progress, unlike in systems for linear and session types, that usually build on additional information, such as partial orders on channels [18, 11].

Channel “polarities” are captured in our system by the left-right symmetry of sequents, rather than by annotations on channels (cf.  $x^+, x^-$ ). Session and linear type systems [19, 16, 13] also include a typing rule for output of the form

$$\frac{\Gamma; \Delta \vdash P :: x : C}{\bar{\Gamma}; \Delta, y : A \vdash x(y).P :: x : A \otimes C}$$

In our case, an analogous rule may be derived by  $\otimes R$  and the copycat construction, where a “proxy” for the free name  $y$ , copying behavior  $A$ , is linked to  $z$ .

$$\frac{\Gamma; \Delta \vdash P :: x : C}{\Gamma; \Delta, y : A \vdash (\nu z)x\langle z \rangle.(id_A(y, z) \mid P) :: x : A \otimes C}$$

The copycat process  $id_A(y, z)$  plays the role of the “link” processes of [21, 7]. Notice that in our case the definition of the “link” is obtained “for free” by the interpretation of identity axioms (Proposition 4.2). The two processes above may be proven behaviorally equivalent, under an adequate notion of observational equivalence, along the lines of [7].

We now elaborate on the example of Section 3, in order to illustrate sharing and session initiation. Consider now a different client, that picks the “buy” rather than the “quote” operation, and the corresponding composed system.

$$\begin{aligned} BClntBody_s &\triangleq s.inl; (\nu cof)s\langle cof \rangle.(\nu pin)s\langle pin \rangle.s(rc)0 \\ BSimple &\triangleq (\nu s)(SrvBody_s \mid BClntBody_s) \end{aligned}$$

Then, we also have

$$\begin{aligned} & ; s : ServerProto \vdash BClntBody_s :: - : \mathbf{1} \\ & ; \cdot \vdash BSimple :: - : \mathbf{1} \end{aligned}$$

In the examples above, there is a single installed pair client-server, where the session is already initiated, and only known to the two partners. To illustrate sharing, we now consider a replicated server. Such a replicated server is capable of spawning a fresh session instance for each initial invocation, each one conforming to the general behavior specified by  $ServerProto$ , and can be typed by  $!ServerProto$ . Correspondingly, clients must initially invoke the replicated server to instantiate a new session (cf. the  $Tcopy$  rule).

$$\begin{aligned} QClient &\triangleq (\nu s)c\langle s \rangle.QClntBody_s \\ BClient &\triangleq (\nu s)c\langle s \rangle.BClntBody_s \\ Server &\triangleq !c\langle s \rangle.SrvBody_s \\ SharSys &\triangleq (\nu c)(Server \mid BClient \mid QClient) \end{aligned}$$

For the shared server, by  $T!R$ , we type

$$; \cdot \vdash Server :: c : !ServerProto$$

We also have, for the clients, by  $Tcopy$  the typings

$$\begin{aligned} c : ServerProto ; \cdot \vdash BClient &:: - : \mathbf{1} \\ c : ServerProto ; \cdot \vdash QClient &:: - : \mathbf{1} \end{aligned}$$

By (comp),  $T!L$ , and  $Tcut$  we obtain the intended typing for the whole concurrent system

$$; \cdot \vdash SharSys :: - : \mathbf{1}$$

Notice how the session instantiation protocol is naturally explained by the logical interpretation of the  $!A$  operator.

## 7.2 Basic Data

Using known techniques basic data may be encoded as  $\pi$ -calculus processes: in our framework, typed encodings arise directly from the type structure. We illustrate with

types for linear and plain booleans. The type  $LBool$  of linear booleans may be defined  $LBool \triangleq \mathbf{1} \oplus \mathbf{1}$ , the behavioral characterization of the linear boolean values being  $ltrue_x \triangleq x.inl; \mathbf{0}$  and  $lfalse_x \triangleq x.inr; \mathbf{0}$ , with typings  $; \cdot \vdash ltrue_x :: x:LBool$  and  $; \cdot \vdash lfalse_x :: x:LBool$ . A destructor for  $LBool$  can be easily defined; as shown in detail for plain (non-linear) booleans. We define the type  $Bool \triangleq !LBool$  of plain booleans, with values given by  $true_x \triangleq !x(b).ltrue_b$  and  $false_x \triangleq !x(b).lfalse_b$ , typed  $; \cdot \vdash true_x :: x:Bool$  and  $; \cdot \vdash false_x :: x:Bool$ . We set

if  $C_x$  then  $P$  else  $Q \triangleq (\nu x)(C_x \mid (\nu b)x\langle b \rangle.b.case(P, Q))$

where  $x \notin fn(P, Q)$ . The following rule is then admissible:

$$\frac{\Gamma; \Delta_1 \vdash C :: x:Bool \quad \Gamma; \Delta_2 \vdash P :: T \quad \Gamma; \Delta_2 \vdash Q :: T}{\Gamma; \Delta_1, \Delta_2 \vdash \text{if } C_x \text{ then } P \text{ else } Q :: T}$$

## 7.3 Linear typed $\lambda$ -calculus

After the proposal of [20], several encodings of  $\lambda$ -calculus in  $\pi$ -calculus have been studied. A natural question to ask is what kind of encoding one may extract (“for free”) from our logical interpretation, e.g., by reading off the cut-free encoding of  $DILL$  typing rules in natural deduction form [3] in our sequent calculus, and using the decomposition  $A \rightarrow B \triangleq !A \multimap B$  to address the simply-typed case. We illustrate with derivable rules for abstraction and application.

$$\frac{\Gamma; \Delta, x:A \vdash \llbracket P \rrbracket_z :: z:B}{\Gamma; \Delta \vdash \llbracket \lambda x:A.P \rrbracket_z :: z:A \multimap B} \quad \llbracket \lambda x:A.P \rrbracket_z \triangleq z(x).\llbracket P \rrbracket_z$$

$$\frac{\Gamma; \Delta \vdash \llbracket P \rrbracket_y :: y:A \multimap B \quad \Gamma; \Delta' \vdash \llbracket Q \rrbracket_w :: w:A}{\Gamma; \Delta, \Delta' \vdash \llbracket PQ \rrbracket_z :: z:B}$$

where  $\llbracket PQ \rrbracket_z \triangleq (\nu y)(\llbracket P \rrbracket_y)(\nu w)y\langle w \rangle.(\llbracket Q \rrbracket_w \mid id_B(y, z))$ . A distinctive aspect of the encoding thus obtained is that, in application, once the “function”  $P$  reduces to an abstraction, after a reference  $w$  to the “argument”  $Q$  is passed to the body both argument and function body evaluate concurrently. In particular, if  $A$  is a  $!C$  type, the various possible occurrences of the parameter will rely on a single computation of the argument, thus, we are not in presence of CBV or CBN, but rather of a sharing reduction strategy.

## 8 Related Work and Conclusions

We have established a tight correspondence between a session-based type discipline for the  $\pi$ -calculus and intuitionistic linear logic: typing rules correspond to dual intuitionistic linear sequent calculus proof rules, moreover process reduction may be simulated in a type preserving way by proof conversions and reductions, and *vice versa*. As a result, we obtain subject reduction. We also proved that the basic typing discipline ensures global progress, (cf. [11]).

As we have seen, the correspondence is rather strong and natural, but not a true isomorphism between proofs and programs because the  $!L$  and  $!R$  rules do not change the extracted  $\pi$ -calculus term. We plan to investigate if an isomorphism can be obtained by exploiting some results regarding the logical justification of the mix rule in linear logic [10].

A realizability interpretation, based on a  $\pi$ -calculus variant, for a linear logic with temporal modalities was proposed in [4]. Other works have also established connections between linear logic and linear type systems for processes [12]. It is important to notice that our session-based interpretation does not require locality (in which only the output capability of names may be transmitted), which seems required in most works on linearity for  $\pi$ -calculi (e.g., [24]).

We have also analyzed the relation between our type discipline and (finite, deadlock-free) session types. Our intuitionistic interpretation establishes an explicit relationship between session-based concurrency and functional computation in which determinism (no races) and progress (deadlock-freedom) are intrinsic characteristics. Interesting related topics would then be the accommodation of recursive types, and the characterization of observational equivalences and logical relations [8] on our typing discipline.

One important motivation for choosing a purely logical approach to typing is that it often suggests uniform and expressive generalizations. We believe that dependent generalizations of the system of simple linear types proposed here, perhaps along the lines of LLF [9] or CLF [23], may be able to capture many additional properties of communication behavior in a purely logical manner. Already, some systems of session types have dependent character, such as [6] that, among other properties, integrates correspondence assertions into session types.

*Acknowledgments.* This work was supported by FCT / MCTES (CMU-PT/NGN44-2009-12), and the Information and Communications Technology Institute at Carnegie Mellon University.

## References

- [1] S. Abramsky. Computational Interpretations of Linear Logic. *Theor. Comput. Sci.*, 111(1&2), 1993.
- [2] J.-M. Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):197–347, 1992.
- [3] A. Barber and G. Plotkin. Dual Intuitionistic Linear Logic. Technical Report LFCS-96-347, Univ. of Edinburgh, 1997.
- [4] E. Beffara. A Concurrent Model for Linear Logic. *Electr. Notes in Theor. Computer Science*, 155:147–168, 2006.
- [5] G. Bellin and P. Scott. On the  $\pi$ -Calculus and Linear Logic. *Theoretical Computer Science*, 135:11–65, 1994.
- [6] E. Bonelli, A. B. Compagnoni, and E. L. Gunter. Correspondence Assertions for Process Synchronization in Concurrent Communications. *Journal of Functional Programming*, 15(2):219–247, 2005.
- [7] M. Boreale. On the Expressiveness of Internal Mobility in Name-Passing Calculi. *Theoretical Computer Science*, 195(2):205–226, 1998.
- [8] L. Caires. Logical semantics of types for concurrency. In T. Mossakowski, U. Montanari, and M. Haverlaan, editors, *International Conference Algebra and Coalgebra in Computer Science, CALCO 2007*, volume 4624 of *Lecture Notes in Computer Science*, pages 16–35. Springer-Verlag, 2007.
- [9] I. Cervesato and F. Pfenning. A Linear Logical Framework. *Information & Computation*, 179(1):19–75, Nov. 2002.
- [10] B.-Y. E. Chang, K. Chaudhuri, and F. Pfenning. A Judgmental Analysis of Linear Logic. Technical Report CMU-CS-03-131R, Carnegie Mellon University, 2003.
- [11] M. Dezani-Ciancaglini, U. de’ Liguoro, and N. Yoshida. On Progress for Structured Communications. In G. Barthe and C. Fournet, editors, *Third Symposium Trustworthy Global Computing, TGC 2007*, volume 4912 of *Lecture Notes in Computer Science*, pages 257–275. Springer-Verlag, 2008.
- [12] C. Faggian and M. Piccolo. Ludics is a Model for the Finitary Linear Pi-Calculus. In *Typed Lambda Calculi and Applications, TLCA 2007*, Lecture Notes in Computer Science, pages 148–162. Springer-Verlag, 2007.
- [13] S. Gay and M. Hole. Subtyping for Session Types in the Pi Calculus. *Acta Informatica*, 42(2-3):191–225, 2005.
- [14] J.-Y. Girard and Y. Lafont. Linear Logic and Lazy Computation. In H. Ehrig, R. A. Kowalski, G. Levi, and U. Montanari, editors, *Intl. Conf. on Theory and Practice of Software Development, TAPSOFT’87*, volume 250 of *Lecture Notes in Computer Science*, pages 52–66. Springer-Verlag, 1987.
- [15] K. Honda. Types for Dyadic Interaction. In E. Best, editor, *4th International Conference on Concurrency Theory, Concur 1993*, volume 715 of *Lecture Notes in Computer Science*, pages 509–523. Springer-Verlag, 1993.
- [16] K. Honda, V. T. Vasconcelos, and M. Kubo. Language Primitives and Type Discipline for Structured Communication-Based Programming. In C. Hankin, editor, *7th European Symposium on Programming Languages and Systems, ESOP 1998*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer-Verlag, 1998.
- [17] J. M. E. Hyland and C.-H. L. Ong. Pi-Calculus, Dialogue Games and PCF. In *WG2.8 Conference on Functional Programming Languages*, pages 96–107, 1995.
- [18] N. Kobayashi. A Partially Deadlock-Free Typed Process Calculus. *ACM Tr. Progr. Lang. Sys.*, 20(2):436–482, 1998.
- [19] N. Kobayashi, B. C. Pierce, and D. N. Turner. Linearity and the Pi-Calculus. In *23rd Symp. on Principles of Programming Languages, POPL 1996*, pages 358–371. ACM, 1996.
- [20] R. Milner. Functions as processes. *Math. Struc. in Computer Sciences*, 2(2):119–141, 1992.
- [21] D. Sangiorgi. Pi-Calculus, Internal Mobility, and Agent Passing Calculi. *Theoretical Computer Science*, 167(1&2):235–274, 1996.
- [22] D. Sangiorgi and D. Walker. *The  $\pi$ -calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.
- [23] K. Watkins, I. Cervesato, F. Pfenning, and D. Walker. Specifying Properties of Concurrent Computations in CLF. *Electr. Notes in Theor. Computer Science*, 155, 2004.
- [24] N. Yoshida, K. Honda, and M. Berger. Linearity and Bisimulation. *J. Log. Algebr. Program.*, 72(2):207–238, 2007.