From $\mathcal{X}$ to π

Representing the Classical Sequent Calculus in the π-calculus

Extended Abstract

Steffen van Bakel[1], Luca Cardelli[2], and Maria Grazia Vigliotti[1]

1: Department of Computing, Imperial College, 180 Queen's Gate, London SW7 2BZ, UK
2: Microsoft Research Cambridge, 7 J J Thomson Avenue, Cambridge, CB3 0FB, UK
svb@doc.ic.ac.uk, luca@microsoft.com, mgv98@doc.ic.ac.uk

Abstract. We study the π-calculus, enriched with pairing and non-blocking input, and define a notion of type assignment that uses the type constructor $\rightarrow$. We encode the circuits of the calculus $\mathcal{X}$ into this variant of π, and show that all reduction (cut-elimination) and assignable types are preserved. Since $\mathcal{X}$ enjoys the Curry-Howard isomorphism for Gentzen's calculus LK, this implies that all proofs in LK have a representation in π.

Introduction

In this paper we present an encoding of proofs of Gentzen's (implicative) LK [15] into the π-calculus [26] that respects *cut*-elimination, and define a new notion of type assignment for π so that processes will become witnesses for the provable formulae. The encoding of classical logic into π-calculus is attained by using the intuition of the calculus $\mathcal{X}$, which gives a computational meaning to LK (a first version of this calculus was proposed in [32, 34, 33]; the implicative fragment of $\mathcal{X}$ was studied in [8]).

$\mathcal{X}$ enjoys the Curry-Howard property for LK; it achieves the isomorphism by constructing witnesses, called *nets*, for derivable sequents. Nets in $\mathcal{X}$ have multiple named inputs and multiple named outputs, that are collectively called *connectors*. Reduction in $\mathcal{X}$ is expressed via a set of rewrite rules that represent *cut*-elimination, eventually leading to renaming of connectors. It is well known that *cut*-elimination in LK is not confluent, and, since $\mathcal{X}$ is Curry-Howard for LK, neither is reduction in $\mathcal{X}$. These two features –non-confluence and reduction as connection of nets via the exchange of names– lead us to consider the π-calculus as an alternative computational model for *cut*-elimination and proofs in LK.

The relation between process calculi and classical logic is an interesting and very promising area of research (similar attempts we made in the context of natural deduction [24] and linear logic [10]). Our aim is to widen further the path to practical application of classical logic in computation by providing an interpretation of classical logic into process algebra, that fully exploits the non-determinism of both LK and π.

The aim of this paper is to link LK and π via $\mathcal{X}$; the main achievements are:

– an encoding of $\mathcal{X}$ into π is defined, that preserves the operational semantics – to achieve this result, reduction in π is generalised;

– we define a non-standard notion of type assignment for π (types do not contain channel information) that encompasses implication;
– the encoding preserves assignable types, effectively showing that all proofs in LK have a representation in π – to represent LK, π is enriched with pairing [2].

Classical sequents, $\mathcal{X}$, and π

The *sequent calculus* LK, introduced by Gentzen in [15], is a logical system in which the rules only introduce connectives (but on either side of a sequent), in contrast to *natural deduction* (also introduced in [15]) which uses rules that introduce or eliminate connectives in the logical formulae. Natural deduction normally derives statements with a single conclusion, whereas LK allows for multiple conclusions, deriving sequents of the form $A_1,\ldots,A_n \vdash B_1,\ldots,B_m$, where $A_1,\ldots,A_n$ is to be understood as $A_1\wedge\ldots\wedge A_n$ and $B_1,\ldots,B_m$ is to be understood as $B_1\vee\ldots\vee B_m$. The version G_3 of Implicative LK has four rules: *axiom*, *left introduction* of the arrow, *right introduction*, and *cut*.

$$(Ax):\frac{}{\Gamma, A\vdash A,\Delta}\qquad (\Rightarrow L):\frac{\Gamma\vdash A,\Delta\quad \Gamma,B\vdash\Delta}{\Gamma,A\Rightarrow B\vdash\Delta}$$

$$(\Rightarrow R):\frac{\Gamma,A\vdash B,\Delta}{\Gamma\vdash A\Rightarrow B,\Delta}\qquad (cut):\frac{\Gamma\vdash A,\Delta\quad \Gamma,A\vdash\Delta}{\Gamma\vdash\Delta}$$

Since LK has only introduction rules, the only way to eliminate a connective is to eliminate the whole formula in which it appears via an application of the (cut)-rule. Gentzen defined a procedure that eliminates all applications of the (cut)-rule from a proof of a sequent, generating a proof in *normal form* of the same sequent, that is, without a cut. This procedure is defined via local reductions of the proof-tree, which has –with some discrepancies– the flavour of term rewriting [25] or the evaluation of explicit substitutions [14, 1].

The calculus $\mathcal{X}$ achieves a Curry-Howard isomorphism, first discovered for Combinatory Logic [13], for the proofs in LK by constructing *witnesses* (called *nets*) for derivable sequents, without any notion of application. In establishing the isomorphism for $\mathcal{X}$, similar to calculi like $\lambda\mu$ [28] and $\overline{\lambda}\mu\tilde{\mu}$ [12], Roman names are attached to formulae in the left context, and Greek names for those on the right, and syntactic structure is associated to the rules. These correspond to *variables* and *co-variables*, respectively, in [35], or, alternatively, to Parigot's λ- and μ-variables [28] (see also [12]).

Gentzen's proof reductions by cut-elimination become the fundamental principle of computation in $\mathcal{X}$. Cuts in proofs are witnessed by $P\widehat{\alpha}\dagger\widehat{x}Q$ (called the *cut* of P and Q via α and x), and the reduction rules specify how to remove them. Since *cut*-elimination in LK is not confluent, neither is reduction in $\mathcal{X}$; for example, when P does not contain α and Q does not contain x, reducing $P\widehat{\alpha}\dagger\widehat{x}Q$ can lead to both P and Q. Reduction in $\mathcal{X}$ boils down to *renaming*: during reduction nets are re-organised, creating nets that are similar, but with different connector names inside.

$\mathcal{X}$'s notion of multiple inputs and outputs is also found in π, and was the original inspiration for our research. The aim of this work is to find a simple and intuitive encoding of LK-proofs in π, and to devise a notion of type assignment for π so that the types in $\mathcal{X}$ are preserved in π. In this precise sense we view processes in π as giving

an alternative computational meaning to proofs in classical logic. Clearly this implies that we had to define a notion of type assignment that uses the type constructor $\rightarrow$ for π; we managed this without having to linearise the calculus as done in [24], and this is one of the contributions of this paper.

Although the calculi $\mathcal{X}$ and π are, of course, essentially different, the similarities go beyond the correspondence of inputs and output between nets in $\mathcal{X}$ and processes in π. Like $\mathcal{X}$, π is application free, and substitution only takes place on *channel names*, similar to the renaming feature of $\mathcal{X}$, so *cut*-elimination is similar to synchronisation.

Related work

In the past, say before Herbelin's PhD [20] and Urban's PhD [32], the study of the relation between computation, programming languages and logic has concentrated mainly on *natural deduction systems* (of course, exceptions exist [16, 18]). In fact, these carry the predicate '*natural*' deservedly; in comparison with, for example, *sequent style systems*, natural deduction systems are easy to understand and reason about. This holds most strongly in the context of *non-classical* logics; for example, the Curry-Howard relation between *Intuitionistic Logic* and the *Lambda Calculus* (with types) is well studied and understood, and has resulted in a vast and well-investigated area of research, resulting in, amongst others, functional programming languages and much further to system F [17] and the Calculus of Constructions [11]. Abramsky [3, 5] has studied correspondence between multiplicative linear logic and processes, and later moved to the context of game semantics [4]. In fact, all the calculi are *applicative* in that abstraction and application (corresponding to arrow introduction and elimination) are the main constructors in the syntax. The link between Classical Logic and continuations and control was first established for the λ_C-Calculus [19] (where C stands for Felleisen's C operator).

The introduction-elimination approach is easy to understand and convenient to use, but is also rather restrictive: for example, the handling of negation is not as nicely balanced, as is the treatment of contradiction (normally represented by the type $\bot$; for a detailed discussion, see [30]). This imbalance can be observed in Parigot's $\lambda\mu$-calculus [28], an approach for representing classical proofs via a natural deduction system in which there is one main conclusion that is being manipulated and possibly several alternative ones. Adding $\bot$ as pseudo-type (only negation, or $A\rightarrow\bot$, is expressed; $\bot\rightarrow A$ is not a type), the $\lambda\mu$-calculus corresponds to *minimal classical logic* [6].

Herbelin has studied the calculus $\overline{\lambda}\mu\tilde{\mu}$ as a non-applicative extension of $\lambda\mu$, which gives a fine-grained account of manipulation of sequents [20, 12, 21]. The relation between call-by-name and call-by-value in the fragment of LK with negation and conjunction is studied in the Dual Calculus [35]; as in calculi like $\lambda\mu$ and $\overline{\lambda}\mu\tilde{\mu}$, that calculus considers a logic with *active* formulae, so these calculi do not achieve a direct Curry-Howard isomorphism with LK. The relation between $\mathcal{X}$ and $\overline{\lambda}\mu\tilde{\mu}$ has been investigated in [7, 8]; there it was shown that it is straightforward to map $\overline{\lambda}\mu\tilde{\mu}$-terms into $\mathcal{X}$ whilst preserving reduction, but that it is not possible to do the converse.

The π-calculus is equipped with a rich type theory [29]: from the basic type system for counting the arity of channels to sophisticated linear types in [24], which studies a relation between Call-by-Value $\lambda\mu$ and a linear π-calculus. Linearisation is used to

be able to achieve processes that are functions, by allowing output over one channel name only. Moreover, the encoding presented in [24] is type dependent, in that, for each term, there are different π-processes assigned, depending on the original type; this makes the encoding quite cumbersome. By contrast, our encoding is very simple and intuitive by interpreting the cut operationally as a communication. The idea of giving a computational interpretation of the cut as a communication primitive is also used by [5] and [10]. In both papers, only a small fragment of Linear Logic was considered, and the encoding between proofs and π-calculus was left rather implicit.

The type system presented in this paper differs quite drastically from the standard type system presented in [29]: here input and output channels essentially have the type of the data they are sending or receiving, and are separated by the type system by putting all inputs with their types on the left of the sequent, and the outputs on the right. In our paper, types give a logical view to the π-calculus rather than an abstract specification on how channels should behave.

1 The calculus $\mathcal{X}$

In this section we will give the definition of the $\mathcal{X}$-calculus which has been proven to be a fine-grained implementation model for various well-known calculi [7], like the λ-calculus [9], $\lambda\mu$ [28] and $\overline{\lambda}\mu\tilde{\mu}$ [21]. As discussed in the introduction, the calculus $\mathcal{X}$ is inspired by the sequent calculus; the system we will consider in this section has only implication, no structural rules and a changed axiom. $\mathcal{X}$ features two separate categories of 'connectors', *plugs* and *sockets*, that act as input and output channels, and is defined without any notion of substitution or application.

Definition 1 (Syntax). The nets of the $\mathcal{X}$-calculus are defined by the following syntax, where x, y range over the infinite set of *sockets*, α, β over the infinite set of *plugs*.

$$\underset{\textit{capsule}}{\langle x \cdot \alpha \rangle} \mid \underset{\textit{export}}{\widehat{y} P \widehat{\beta} \cdot \alpha} \mid \underset{\textit{import}}{P \widehat{\beta} \, [y] \, \widehat{x} Q} \mid \underset{\textit{cut}}{P \widehat{\alpha} \dagger \widehat{x} Q}$$

The $\widehat{\cdot}$ symbolises that the socket or plug underneath is bound in the net. The notion of bound and free connector (free sockets $fs(P)$, and free plugs $fp(P)$, respectively, and $fc(P) = fs(P) \cup fp(P)$) is defined as usual, and we will identify nets that only differ in the names of bound connectors, as usual. We accept Barendregt's convention on names, which states that no name can occur both free *and* bound in a context; α-conversion is supposed to take place silently, whenever necessary.

The calculus, defined by the reduction rules below, explains in detail how cuts are propagated through nets to be eventually evaluated at the level of capsules, where the renaming takes place. Reduction is defined by specifying both the interaction between well-connected basic syntactic structures, and how to deal with propagating active nodes to points in the net where they can interact.

It is important to know when a connector is introduced, i.e. is connectable, i.e. is exposed and unique; this will play an important role in the reduction rules. Informally, a net P introduces a socket x if P is constructed from sub-nets which do not contain x as free socket, so x only occurs at the "top level." This means that P is either an import

with a middle connector $[x]$ or a capsule with left part x. Similarly, a net introduces a plug α if it is an export that "creates" α or a capsule with right part α.

Definition 2. (P *introduces* x) : Either $P = Q\widehat{\beta}\,[x]\,\widehat{y}R$ with $x \notin fs(Q,R)$, or $P = \langle x{\cdot}\alpha\rangle$.
(P *introduces* α) : Either $P = \widehat{x}Q\widehat{\beta}{\cdot}\alpha$ and $\alpha \notin fp(Q)$, or $P = \langle x{\cdot}\alpha\rangle$.

The principal reduction rules are:

Definition 3 (Logical rules). Let α and x be introduced in, respectively, the left- and right-hand side of the main cuts below.

$$\begin{array}{rrcl}
(cap): & \langle y{\cdot}\alpha\rangle\widehat{\alpha} \dagger \widehat{x}\langle x{\cdot}\beta\rangle & \to_{\mathcal{X}} & \langle y{\cdot}\beta\rangle \\
(exp): & (\widehat{y}P\widehat{\beta}{\cdot}\alpha)\widehat{\alpha} \dagger \widehat{x}\langle x{\cdot}\gamma\rangle & \to_{\mathcal{X}} & \widehat{y}P\widehat{\beta}{\cdot}\gamma \\
(imp): & \langle y{\cdot}\alpha\rangle\widehat{\alpha} \dagger \widehat{x}(Q\widehat{\beta}\,[x]\,\widehat{z}R) & \to_{\mathcal{X}} & Q\widehat{\beta}\,[y]\,\widehat{z}R \\
(exp\text{-}imp): & (\widehat{y}P\widehat{\beta}{\cdot}\alpha)\widehat{\alpha} \dagger \widehat{x}(Q\widehat{\gamma}\,[x]\,\widehat{z}R) & \to_{\mathcal{X}} & \begin{cases} Q\widehat{\gamma} \dagger \widehat{y}(P\widehat{\beta} \dagger \widehat{z}R) \\ (Q\widehat{\gamma} \dagger \widehat{y}P)\widehat{\beta} \dagger \widehat{z}R \end{cases}
\end{array}$$

The first three logical rules above specify a renaming procedure, whereas the last rule specifies the basic computational step: it links the export of a function, available on the plug α, to an adjacent import via the socket x. The effect of the reduction will be that the exported function is placed in-between the two sub-terms of the import, acting as interface. Notice that two cuts are created in the result, that can be grouped in two ways; these alternatives do not necessarily share all normal forms (reduction is non-confluent, so normal forms are not unique).

In $\mathcal{X}$ there are in fact two kinds of reduction, the one above, and the one which defines how to reduce a cut when one of its sub-nets does not introduce a connector mentioned in the cut. This will involve moving the cut inwards, towards a position where the connector *is* introduced. In case both connectors are not introduced, this search can start in either direction, indicated by the tilting of the dagger.

Definition 4 (Active cuts). The syntax is extended with two *flagged* or *active* cuts:

$$P ::= \ldots \mid P_1\widehat{\alpha} \not\dagger \widehat{x}P_2 \mid P_1\widehat{\alpha} \not\backslash \widehat{x}P_2$$

We define two cut-activation rules.

$$\begin{array}{rcl}
(a\not\dagger): & P\widehat{\alpha} \dagger \widehat{x}Q & \to_{\mathcal{X}} \quad P\widehat{\alpha} \not\dagger \widehat{x}Q \text{ if } P \text{ does not introduce } \alpha \\
(\not\backslash a): & P\widehat{\alpha} \dagger \widehat{x}Q & \to_{\mathcal{X}} \quad P\widehat{\alpha} \not\backslash \widehat{x}Q \text{ if } Q \text{ does not introduce } x
\end{array}$$

The next rules define how to move an activated dagger inwards.

Definition 5 (Propagation rules). Left propagation:

$$\begin{array}{rrcll}
(d\not\dagger): & \langle y{\cdot}\alpha\rangle\widehat{\alpha} \not\dagger \widehat{x}P & \to_{\mathcal{X}} & \langle y{\cdot}\alpha\rangle\widehat{\alpha} \dagger \widehat{x}P & \\
(cap\not\dagger): & \langle y{\cdot}\beta\rangle\widehat{\alpha} \not\dagger \widehat{x}P & \to_{\mathcal{X}} & \langle y{\cdot}\beta\rangle & \beta \neq \alpha \\
(exp\text{-}outs\not\dagger): & (\widehat{y}Q\widehat{\beta}{\cdot}\alpha)\widehat{\alpha} \not\dagger \widehat{x}P & \to_{\mathcal{X}} & (\widehat{y}(Q\widehat{\alpha} \not\dagger \widehat{x}P)\widehat{\beta}{\cdot}\gamma)\widehat{\gamma} \dagger \widehat{x}P & \gamma \text{ fresh} \\
(exp\text{-}ins\not\dagger): & (\widehat{y}Q\widehat{\beta}{\cdot}\gamma)\widehat{\alpha} \not\dagger \widehat{x}P & \to_{\mathcal{X}} & \widehat{y}(Q\widehat{\alpha} \not\dagger \widehat{x}P)\widehat{\beta}{\cdot}\gamma & \gamma \neq \alpha \\
(imp\not\dagger): & (Q\widehat{\beta}\,[z]\,\widehat{y}R)\widehat{\alpha} \not\dagger \widehat{x}P & \to_{\mathcal{X}} & (Q\widehat{\alpha} \not\dagger \widehat{x}P)\widehat{\beta}\,[z]\,\widehat{y}(R\widehat{\alpha} \not\dagger \widehat{x}P) & \\
(cut\not\dagger): & (Q\widehat{\beta} \dagger \widehat{y}R)\widehat{\alpha} \not\dagger \widehat{x}P & \to_{\mathcal{X}} & (Q\widehat{\alpha} \not\dagger \widehat{x}P)\widehat{\beta} \dagger \widehat{y}(R\widehat{\alpha} \not\dagger \widehat{x}P) &
\end{array}$$

Right propagation:

$$
\begin{array}{rlll}
(\nwarrow d): & P\widehat{\alpha} \nwarrow \widehat{x}\langle x\cdot\beta\rangle & \to_{\mathcal{X}} & P\widehat{\alpha} \dagger \widehat{x}\langle x\cdot\beta\rangle \\
(\nwarrow cap): & P\widehat{\alpha} \nwarrow \widehat{x}\langle y\cdot\beta\rangle & \to_{\mathcal{X}} & \langle y\cdot\beta\rangle \qquad\qquad y \neq x \\
(\nwarrow exp): & P\widehat{\alpha} \nwarrow \widehat{x}(\widehat{y}Q\widehat{\beta}\cdot\gamma) & \to_{\mathcal{X}} & \widehat{y}(P\widehat{\alpha} \nwarrow \widehat{x}Q)\widehat{\beta}\cdot\gamma \\
(\nwarrow imp\text{-}outs): & P\widehat{\alpha} \nwarrow \widehat{x}(Q\widehat{\beta}\,[x]\,\widehat{y}R) & \to_{\mathcal{X}} & \\
 & & & P\widehat{\alpha} \dagger \widehat{z}((P\widehat{\alpha} \nwarrow \widehat{x}Q)\widehat{\beta}\,[z]\,\widehat{y}(P\widehat{\alpha} \nwarrow \widehat{x}R)),\ z \textit{ fresh} \\
(\nwarrow imp\text{-}ins): & P\widehat{\alpha} \nwarrow \widehat{x}(Q\widehat{\beta}\,[z]\,\widehat{y}R) & \to_{\mathcal{X}} & (P\widehat{\alpha} \nwarrow \widehat{x}Q)\widehat{\beta}\,[z]\,\widehat{y}(P\widehat{\alpha} \nwarrow \widehat{x}R)\ \ z \neq x \\
(\nwarrow cut): & P\widehat{\alpha} \nwarrow \widehat{x}(Q\widehat{\beta} \dagger \widehat{y}R) & \to_{\mathcal{X}} & (P\widehat{\alpha} \nwarrow \widehat{x}Q)\widehat{\beta} \dagger \widehat{y}(P\widehat{\alpha} \nwarrow \widehat{x}R)
\end{array}
$$

We write $\to_{\mathcal{X}}$ for the (reflexive, transitive, compatible) reduction relation generated by the logical, propagation and activation rules.

The reduction $\to_{\mathcal{X}}$ is not confluent; confluent sub-systems are defined in [8].

Summarising, reduction brings all cuts down to logical cuts where both connectors single and introduced, or elimination cuts that are cutting towards a capsule that does not contain the relevant connector. Cuts towards connectors occurring in capsules lead to renaming ($P\widehat{\alpha} \nwarrow \widehat{x}\langle x\cdot\beta\rangle \to_{\mathcal{X}} P[\beta/\alpha]$ and $\langle z\cdot\alpha\rangle\widehat{\alpha} \not\frown \widehat{x}P \to_{\mathcal{X}} P[z/x]$), and towards non-occurring connectors leads to elimination ($P\widehat{\alpha} \nwarrow \widehat{x}\langle z\cdot\beta\rangle \to_{\mathcal{X}} \langle z\cdot\beta\rangle$ and $\langle z\cdot\beta\rangle\widehat{\alpha} \not\frown \widehat{x}P \to_{\mathcal{X}} \langle z\cdot\beta\rangle$).

2 Typing for $\mathcal{X}$: from LK to $\mathcal{X}$

$\mathcal{X}$ offers a natural presentation of the classical propositional calculus with implication, and can be seen as a variant of system LK.

We first define types and contexts.

Definition 6 (Types and Contexts). 1. The set of types is defined by the grammar: $A, B ::= \varphi \mid A \to B$, where φ is a basic type of which there are infinitely many.
2. A *context of sockets* Γ is a finite set of *statements* $x{:}A$, such that the *subject* of the statements (x) are distinct. We write Γ_1, Γ_2 to mean the union of Γ_1 and Γ_2, provided Γ_1 and Γ_2 are compatible (if Γ_1 contains $x{:}A_1$ and Γ_2 contains $x{:}A_2$ then $A_1 = A_2$), and write $\Gamma, x{:}A$ for $\Gamma, \{x{:}A\}$.
3. Contexts of *plugs* Δ are defined in a similar way.

The notion of type assignment on $\mathcal{X}$ that we present in this section is the basic implicative system for Classical Logic (Gentzen's system LK) as described above. The Curry-Howard property is easily achieved by erasing all term-information. When building witnesses for proofs, propositions receive names; those that appear in the left part of a sequent receive names like x, y, z, etc, and those that appear in the right part of a sequent receive names like α, β, γ, etc. When in applying a rule a formula disappears from the sequent, the corresponding connector will get bound in the net that is constructed, and when a formula gets created, a new connector will be associated to it.

Definition 7 (Typing for $\mathcal{X}$). 1. *Type judgements* are expressed via a ternary relation $P :\cdot \Gamma \vdash_{\mathcal{X}} \Delta$, where Γ is a context of *sockets* and Δ is a context of *plugs*, and P is a net. We say that P is the *witness* of this judgement.

2. *Type assignment for* $\mathcal{X}$ is defined by the following rules:

$$(cap): \frac{}{\langle y\cdot\alpha\rangle \;:\cdot\; \Gamma, y{:}A \vdash_{\mathcal{X}} \alpha{:}A, \Delta} \qquad (imp): \frac{P \;:\cdot\; \Gamma \vdash_{\mathcal{X}} \alpha{:}A, \Delta \quad Q \;:\cdot\; \Gamma, x{:}B \vdash_{\mathcal{X}} \Delta}{P\widehat{\alpha}\,[y]\,\widehat{x}Q \;:\cdot\; \Gamma, y{:}A{\to}B \vdash_{\mathcal{X}} \Delta}$$

$$(exp): \frac{P \;:\cdot\; \Gamma, x{:}A \vdash_{\mathcal{X}} \alpha{:}B, \Delta}{\widehat{x}P\widehat{\alpha}\cdot\beta \;:\cdot\; \Gamma \vdash_{\mathcal{X}} \beta{:}A{\to}B, \Delta} \qquad (cut): \frac{P \;:\cdot\; \Gamma \vdash_{\mathcal{X}} \alpha{:}A, \Delta \quad Q \;:\cdot\; \Gamma, x{:}A \vdash_{\mathcal{X}} \Delta}{P\widehat{\alpha} \dagger \widehat{x}Q \;:\cdot\; \Gamma \vdash_{\mathcal{X}} \Delta}$$

Notice that Γ and Δ carry the types of the free connectors in P, as unordered sets. There is no notion of type for P itself, instead the derivable statement shows how P is connectable.

Example 8 (A proof of Peirce's Law). The following is a proof for Peirce's Law in LK:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A \vdash A, B}\,(Ax)}{\vdash A \Rightarrow B, A}\,(\Rightarrow R) \quad \dfrac{}{A \vdash A}\,(Ax)}{(A \Rightarrow B) \Rightarrow A \vdash A}\,(\Rightarrow L)}{\vdash ((A \Rightarrow B) \Rightarrow A) \Rightarrow A}\,(\Rightarrow R)$$

Inhabiting this proof in $\mathcal{X}$ gives the derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\langle y\cdot\delta\rangle \;:\cdot\; y{:}A \vdash_{\mathcal{X}} \delta{:}A, \eta{:}B}\,(cap)}{\widehat{y}\langle y\cdot\delta\rangle\widehat{\eta}\cdot\alpha \;:\cdot\; \vdash_{\mathcal{X}} \alpha{:}A{\to}B, \delta{:}A}\,(exp) \quad \dfrac{}{\langle w\cdot\delta\rangle \;:\cdot\; w{:}A \vdash_{\mathcal{X}} \delta{:}A}\,(cap)}{(\widehat{y}\langle y\cdot\delta\rangle\widehat{\eta}\cdot\alpha)\widehat{\alpha}\;[z]\;\widehat{w}\langle w\cdot\delta\rangle \;:\cdot\; z{:}(A{\to}B){\to}A \vdash_{\mathcal{X}} \delta{:}A}\,(imp)}{\widehat{z}((\widehat{y}\langle y\cdot\delta\rangle\widehat{\eta}\cdot\alpha)\widehat{\alpha}\;[z]\;\widehat{w}\langle w\cdot\delta\rangle)\widehat{\delta}\cdot\gamma \;:\cdot\; \vdash_{\mathcal{X}} \gamma{:}((A{\to}B){\to}A){\to}A}\,(exp)$$

The following soundness result is proven in [8]:

Theorem 9 (Witness reduction). If $P \;:\cdot\; \Gamma \vdash_{\mathcal{X}} \Delta$, and $P \to_{\mathcal{X}} Q$, then $Q \;:\cdot\; \Gamma \vdash_{\mathcal{X}} \Delta$.

3 The asynchronous π-calculus with pairing and nesting

The notion of asynchronous π-calculus that we consider in this paper is different from other systems studied in the literature [22]. One reason for this change lies directly in the calculus that is going to be interpreted, $\mathcal{X}$: since we are going to model sending and receiving pairs of names as interfaces for functions, we add pairing, inspired by [2]. The other reason is that we want to achieve a preservation of *full* cut-elimination; to this aim, we need to use *non-blocking* inputs, by adding the reduction rule (*nesting*) (see Definition 12). Without this last addition, we cannot model full cut-elimination; this was, for example, also the case with the interpretations defined by Milner [26], Sangiorgi [29], Honda *et al* [24], and Thielecke [31], where reduction in the original calculus had to be restricted in order to get a completeness result. Notice that this last extension of π *only* relates to cut-elimination: that all proofs in LK are representable in π is not affected by this, nor is the preservation of types.

To ease the definition of the interpretation function of circuits in $\mathcal{X}$ to processes in the π-calculus, we deviate slightly from the normal practice, and write either Greek characters $\alpha, \beta, \upsilon, \ldots$ or Roman characters $x, y, z, \ldots$ for channel names; we use n for either a Greek or a Roman name, and '$\circ$' for the generic variable. We also introduce a structure over names, such that not only names but also pairs of names can be sent (but not a pair of pairs). In this way a channel may pass along either a name or a pair of names. We also introduce the let-construct to deal with inputs of pairs of names that get distributed over the continuation.

Definition 10. Channel names and data are defined by:

$$a, b, c, d ::= x \mid \alpha \quad \textit{names} \qquad\qquad p ::= a \mid \langle a, b \rangle \quad \textit{data}$$

Notice that pairing is *not* recursive. Processes are defined by:

$$\begin{array}{lll} P, Q ::= 0 & \textit{Nil} & \\ \quad \mid P \mid Q & \textit{Composition} & \mid a(x).P \quad \textit{Input} \\ \quad \mid !\,P & \textit{Replication} & \mid \overline{a}\langle p \rangle \quad \textit{(Asynchronous) Output} \\ \quad \mid (\nu a)\,P & \textit{Restriction} & \mid \textit{let}\,\langle x, y \rangle = z\ \textit{in}\ P \quad \textit{Let construct} \end{array}$$

We abbreviate $a(x).\,\textit{let}\,\langle y, z \rangle = x\ \textit{in}\ P$ by $a(\langle y, z \rangle).P$, and $(\nu m)\,(\nu n)\,P$ by $(\nu m, n)\,P$.

A (process) context is simply a term with a hole $[\cdot]$.

Definition 11 (Congruence). The structural congruence is the smallest equivalence relation closed under contexts defined by the following rules:

$$\begin{array}{rlrl} P \mid \mathbf{0} & \equiv P & (\nu m)\,(\nu n)\,P & \equiv (\nu n)\,(\nu m)\,P \\ P \mid Q & \equiv Q \mid P & (\nu n)\,(P \mid Q) & \equiv P \mid (\nu n)\,Q \quad \textit{if}\ n \notin \mathit{fn}(P) \\ (P \mid Q) \mid R & \equiv P \mid (Q \mid R) & !\,P & \equiv P \mid !\,P \\ (\nu n)\,\mathbf{0} & \equiv \mathbf{0} & \textit{let}\,\langle x, y \rangle = \langle a, b \rangle\ \textit{in}\ R & \equiv R[a/x, b/y] \end{array}$$

Definition 12. 1. The *reduction relation* over the processes of the π-calculus is defined by following (elementary) rules:

$$\begin{array}{rl} (\textit{synchronisation}): & \overline{a}\langle b \rangle.P \mid a(x).Q \ \rightarrow_\pi \ P \mid Q[b/x] \\ (\textit{binding}): & P \rightarrow_\pi P' \ \Rightarrow \ (\nu n)\,P \rightarrow_\pi (\nu n)\,P' \\ (\textit{composition}): & P \rightarrow_\pi P' \ \Rightarrow \ P \mid Q \rightarrow_\pi P' \mid Q \\ (\textit{nesting}): & P \rightarrow_\pi Q \ \Rightarrow \ n(x).P \rightarrow_\pi n(x).Q \\ (\textit{congruence}): & P \equiv Q \ \&\ Q \rightarrow_\pi Q' \ \&\ Q' \equiv P' \ \Rightarrow \ P \rightarrow_\pi P' \end{array}$$

2. We write $\rightarrow_\pi^*$ for the reflexive and transitive closure of $\rightarrow_\pi$.
3. We write $P \downarrow n$ if $P \equiv (\nu b_1 \ldots b_m)\,(\overline{n}\langle p \rangle \mid Q)$ for some Q, where $n \neq b_1 \ldots b_m$.
4. We write $Q \Downarrow n$ if there exists P such that $Q \rightarrow_\pi^* P$ and $P \downarrow n$.

Notice that we no longer consider input in π to be *blocking*; we are aware that this is a considerable breach with normal practice, but this is strongly needed in our completeness result (Theorem 20); without it, we can at most show a partial result.

Moreover, notice that

$$\overline{a}\langle\langle b, c \rangle\rangle \mid a(\langle x, y \rangle).Q \ \rightarrow_\pi^* \ Q[b/x, c/y]$$

Definition 13 ([23]). *Barbed contextual simulation* is the largest relation $\preceq_\pi$ such that $P \preceq_\pi Q$ implies:

- for each name n, if $P \downarrow n$ then $Q \Downarrow n$;
- for any context C, if $C[P] \rightarrow_\pi P'$, then for some Q', $C[Q] \rightarrow_\pi^* Q'$ and $P' \preceq_\pi Q'$.

4 Type assignment

In this section, we introduce a notion of type assignment for processes in π that describes the '*input-output interface*' of a process. This notion is novel in that it assigns to channels the type of the input or output that is sent over the channel; in that it differs from normal notions, that would state:

$$\frac{}{\overline{a}\langle b\rangle \;\vdots\cdot\; \Gamma, b{:}A \vdash a{:}\mathsf{ch}(A), \Delta}$$

In order to be able to encode LK, types in our system will not be decorated with channel information.

As for the notion of type assignment on $\mathcal{X}$ terms, in the typing judgements we always write channels used for input on the left and channels used for output on the right; this implies that, if a channel is both used to send and to receive, it will appear on both sides.

Definition 14 (Type assignment). The types and contexts we consider for the π-calculus are defined like those of Definition 6, generalised to names. Type assignment for π-calculus is defined by the following sequent system:

$$(0): \frac{}{0 \;\vdots\cdot\; \Gamma \vdash_\pi \Delta} \qquad (\mathit{in}): \frac{P \;\vdots\cdot\; \Gamma, x{:}A \vdash_\pi x{:}A.\Delta}{a(x).P \;\vdots\cdot\; \Gamma, a{:}A \vdash_\pi \Delta}$$

$$(!): \frac{P \;\vdots\cdot\; \Gamma \vdash_\pi \Delta}{!\,P \;\vdots\cdot\; \Gamma \vdash_\pi \Delta} \qquad (\mathit{out}): \frac{}{\overline{a}\langle b\rangle \;\vdots\cdot\; \Gamma, b{:}A \vdash_\pi a{:}A, b{:}A, \Delta}$$

$$(\nu): \frac{P \;\vdots\cdot\; \Gamma, a{:}A \vdash_\pi a{:}A, \Delta}{(\nu a)\,P \;\vdots\cdot\; \Gamma \vdash_\pi \Delta} \qquad (\mathit{pair\text{-}out}): \frac{}{\overline{a}\langle\langle b, c\rangle\rangle \;\vdots\cdot\; \Gamma, b{:}A \vdash_\pi a{:}A{\rightarrow}B, c{:}B, \Delta}$$

$$(|): \frac{P \;\vdots\cdot\; \Gamma \vdash_\pi \Delta \quad Q \;\vdots\cdot\; \Gamma \vdash_\pi \Delta}{P \mid Q \;\vdots\cdot\; \Gamma \vdash_\pi \Delta} \qquad (\mathit{let}): \frac{P \;\vdots\cdot\; \Gamma, y{:}B \vdash_\pi x{:}A, \Delta}{\mathit{let}\,\langle x, y\rangle = z \;\mathit{in}\; P \;\vdots\cdot\; \Gamma, z{:}A{\rightarrow}B \vdash_\pi \Delta}$$

Notice that it is possible to derive $\overline{a}\langle a\rangle \;\vdots\cdot\; \vdash_\pi a{:}A$, although sending a channel name over that channel itself is never produced by our encoding, nor by the reduction of processes created by the encoding.

Example 15. We can derive

$$\frac{\dfrac{P \;\vdots\cdot\; \Gamma, y{:}B \vdash_\pi x{:}A, \Delta}{\mathit{let}\,\langle x, y\rangle = z \;\mathit{in}\; P \;\vdots\cdot\; \Gamma, z{:}A{\rightarrow}B \vdash_\pi \Delta}}{a(z).\,\mathit{let}\,\langle x, y\rangle = z \;\mathit{in}\; P \;\vdots\cdot\; \Gamma, a{:}A{\rightarrow}B \vdash_\pi \Delta}$$

so the following rule is derivable:

$$(\textit{pair-in}) : \frac{P \vdash \Gamma, y{:}B \vdash_\pi x{:}A, \Delta}{a(\langle x, y\rangle).P \vdash \Gamma, a{:}A{\to}B \vdash_\pi \Delta}$$

Notice that the rule $(\textit{pair-out})$ does not directly correspond to the logical rule $(\Rightarrow R)$, as that $(\textit{pair-in})$ does not directly correspond to $(\Rightarrow L)$; this is natural, however, seen that the encoding does not map rules to rules, but proofs to type derivations. This apparent discrepancy is solved by Theorem 21.

In fact, this notion of type assignment does not (directly) relate back to LK. For example, rules $(|)$ and $(!)$ do not change the contexts, so do not correspond to any rule in the logic, not even to a $\lambda\mu$-style activation step.

Notice that the cases $P \vdash \Gamma \vdash_\pi x{:}A, \Delta$ and $P \vdash \Gamma, x{:}A \vdash_\pi \Delta$ can be generalised by weakening to fit the lemma.

We now come to the main soundness result for our notion of type assignment for π.

Theorem 16 (Witness reduction). If $P \vdash \Gamma \vdash_\pi \Delta$ and $P \to_\pi Q$, then $Q \vdash \Gamma \vdash_\pi \Delta$.

5 Interpreting $\mathcal{X}$ into π

In this section, we define an encoding from nets in $\mathcal{X}$ onto processes in π.

The encoding defined below is based on the intuition as formulated in [8]: the cut $P\widehat{\alpha} \dagger \widehat{x}Q$ expresses the intention to connect all αs in P and xs in Q, and reduction will realise this by either connecting all αs to all xs, or all xs to all αs. Translated into π, this results in seeing P as trying to send at least as many times over α as Q is willing to receive over x, and Q trying to receive at least as many times over x as P is ready to send over α.

As discussed above, when creating a witness for $(\Rightarrow R)$ (the net $\widehat{x}P\widehat{\alpha}\cdot\beta$, called an *export*), the exported interface of P is the functionality of 'receiving on x, sending on α', which is made available on β. When encoding this behaviour in π, we are faced with a problem. It is clearly not sufficient to limit communication to the exchange of single names, since then we would have to separately send x and α, breaking perhaps the exported functionality, and certainly disabling the possibility of assigning arrow types. We overcome this problem by sending out a pair of names, as in $\overline{a}\langle\langle v, \delta\rangle\rangle$. Similarly, when interpreting a witness for $(\Rightarrow L)$ (the net $P\widehat{\alpha}\,[x]\,\widehat{y}Q$, called an *import*), the circuit that is to be connected to x is ideally a function whose input will be connected to α, and its output to y. This means that we need to receive a pair of names over x, as in $x(\langle v, \delta\rangle).P$.

A cut $P\widehat{\alpha} \dagger \widehat{x}Q$ in $\mathcal{X}$ expresses two nets that need to be connected via α and x. If we model P and Q in π, then we obtain one process sending on α, and one receiving on x, and we need to link these via $\alpha(\cdot).\overline{x}\langle\cdot\rangle$. Since each output on α in P takes place only once, and Q might want to receive in more than one x, we need to replicate the sending; likewise, since each input x in Q takes place only once, and P might have more than one send operation on α, Q needs to be replicated.

We added pairing to the π-calculus in order to be able to deal with arrow types. Notice that using the polyadic π-calculus would not be sufficient: since we would like the interpretation to respect reduction, in particular we need to be able to reduce the interpretation of $(\widehat{x}P\widehat{\alpha}\cdot\beta)\widehat{\beta}\dagger\widehat{z}\langle z\cdot\gamma\rangle$ to that of $\widehat{x}P\widehat{\alpha}\cdot\gamma$ (when β not free in P). So, choosing to encode the export of x and α over β as $\overline{\beta}\langle x,\alpha\rangle$ would force the interpretation of $\langle z\cdot\gamma\rangle$ to receive a pair of names. But requiring for a capsule to always deal with pairs of names is too restrictive, it is desirable to allow capsules to deal with single names as well. So, rather than moving towards the polyadic π-calculus, we opt for letting communication send a single item, which is either a name or a pair of names. This implies that a process sending a pair can also successfully communicate with a process not explicitly demanding to receive a pair.

Definition 17 (Notation). In the definition below, we use '$\circ$' for the generic variable, to separate plugs and sockets (and their interpretation) from the 'internal' variables of π. Also, although the departure point is to view Greek names for outputs and Roman names forinputs, by the very nature of the π-calculus (it is only possible to communicate using the *same* channel for in and output), in the implementation we are forced to use Greek names also for inputs, and Roman names for outputs; in fact, we need to explicitly convert '*an output sent on α is to be received as input on x*' via '$\alpha(\circ)\overline{x}\langle\circ\rangle$' (so α is now also an input, and x also an output channel), which for convenience is abbreviated into $\alpha{=}x$.

Definition 18. The interpretation of circuits is defined by:

$$\begin{array}{rcl}
\ulcorner\langle x\cdot\alpha\rangle\urcorner_\pi & = & x(\circ).\,\overline{\alpha}\langle\circ\rangle \\
\ulcorner\widehat{y}Q\widehat{\beta}\cdot\alpha\urcorner_\pi & = & (\nu y,\beta)\,(\ulcorner Q\urcorner_\pi \mid \overline{\alpha}\langle\langle y,\beta\rangle\rangle) \\
\ulcorner P\widehat{\alpha}\,[x]\,\widehat{y}Q\urcorner_\pi & = & x(\langle v,d\rangle).\,(\nu\alpha)\,(!\ulcorner P\urcorner_\pi \mid !\,\alpha{=}v) \mid (\nu y)\,(!\,d{=}y \mid !\ulcorner Q\urcorner_\pi) \\
\ulcorner P\widehat{\alpha}\dagger\widehat{x}Q\urcorner_\pi & = & \ulcorner P\widehat{\alpha}\not\leftarrow\widehat{x}Q\urcorner_\pi = \ulcorner P\widehat{\alpha}\not\rightarrow\widehat{x}Q\urcorner_\pi = (\nu\alpha,x)\,(!\ulcorner P\urcorner_\pi \mid !\,\alpha{=}x \mid !\ulcorner Q\urcorner_\pi)
\end{array}$$

Notice that the interpretation of the inactive cut is the same as that of activated cuts. This implies that we are, in fact, also interpreting a variant of $\mathcal{X}$ *without* activated cuts, allowing arbitrary movement of cuts over cuts, but with the same set of rewrite rules. This is very different from Gentzen's original definition – he in fact does not define a cut-over-cut step, and uses innermost reduction for his *Hauptsatz* result – and different from Urban's definition – allowing only *activated* cuts to propagate is crucial for his Strong Normalisation result. Also, one could argue that then the reduction rules no longer present a system of *cut-elimination*, since now rule ($\not\leftarrow cut$) reads:

$$P\widehat{\alpha}\dagger\widehat{x}(Q\widehat{\beta}\dagger\widehat{y}R)\ \rightarrow_{\mathcal{X}}\ (P\widehat{\alpha}\dagger\widehat{x}Q)\widehat{\beta}\dagger\widehat{y}(P\widehat{\alpha}\dagger\widehat{x}R)$$

in which it is doubtful that a cut has been eliminated; it is also easy to show that this creates loops in the reduction system. However, this rewriting is still sound with respect to typeability. Here we can abstract from these aspects, since we only aim to prove a *simulation* result, for which the encoding above will be shown adequate.

Example 19. The encoding of the witness of Peirce's law becomes:

$$\begin{array}{l}
\ulcorner\widehat{z}((\widehat{y}\langle y\cdot\delta\rangle\widehat{\eta}\cdot\alpha)\widehat{\alpha}\,[z]\,\widehat{w}\langle w\cdot\delta\rangle)\widehat{\delta}\cdot\gamma\urcorner_\pi = \\
(\nu z,\delta)\,(z(\langle v,d\rangle).\,(\nu\alpha)\,!\,((\nu y,\eta)\,(y(\circ).\,\overline{\delta}\langle\circ\rangle \mid \overline{\alpha}\langle\langle y,\eta\rangle\rangle) \mid \alpha{=}v) \mid \\
\qquad\qquad (\nu w)\,!\,(d{=}w \mid w(\circ).\,\overline{\delta}\langle\circ\rangle) \mid \overline{\gamma}\langle\langle z,\delta\rangle\rangle)
\end{array}$$

That this process is a witness of $((A\to B)\to A)\to A$ is a straightforward application of Theorem 21 below.

The correctness result for the encoding essentially states that the image of the encoding in π contains some extra behaviour that can be disregarded.

Theorem 20. *If $P \to_{\mathcal{X}} P'$, then for some Q, $\ulcorner P\urcorner_\pi \to_\pi^* Q$ and $\ulcorner P'\urcorner_\pi \preceq_\pi Q$.*

This result might appear weak at first glance, but it would be a mistake to dismiss the encoding on such an observation.

Our result states that the encoding of $\mathcal{X}$ into π contains more behaviour than the original term. In part, the extra behaviour is due to replicated processes, which can be easily discharged; but, more importantly, π has no notion of *erasure* of processes: the cut $P\widehat{\alpha} \dagger \widehat{x}Q$, with α not in P and x not in Q, in $\mathcal{X}$ erases either P or Q, but $\ulcorner P\widehat{\alpha} \dagger \widehat{x}Q\urcorner_\pi$ then runs to $\ulcorner P\urcorner_\pi \mid \ulcorner Q\urcorner_\pi$. The result presented in [24] is stronger, but only achieved for Call-by-Value $\lambda\mu$, and at the price of a very intricate translation that depends on types. Also $\ulcorner P\urcorner_\pi$ essentially contains all normal forms of P in parallel; since $\lambda\mu$ is confluent, there is only one normal form, so the problem disappears. Moreover, restricting to either (confluent) call-by-name or call-by-value restrictions, also then the problem disappears.

The following theorem states one of the main results of this paper: it shows that the encoding preserves types.

Theorem 21. *If $P \vdots \Gamma \vdash_{\mathcal{X}} \Delta$, then $\ulcorner P\urcorner_\pi \vdots \Gamma \vdash_\pi \Delta$.*

Notice that this theorem links proofs in LK to type derivations in $\vdash_\pi$

6 The Lambda Calculus

We assume the reader to be familiar with the λ-calculus; we just repeat the definition of (simple) type assignment.

Definition 22 (Type assignment for the λ-calculus).

$$(Ax): \frac{}{\Gamma, x{:}A \vdash_\lambda x : A} \qquad (\to I): \frac{\Gamma, x{:}A \vdash_\lambda M : B}{\Gamma \vdash_\lambda \lambda x.M : A\to B}$$

$$(\to E): \frac{\Gamma \vdash_\lambda M : A\to B \qquad \Gamma \vdash_\lambda N : A}{\Gamma \vdash_\lambda MN : B}$$

The following was already defined in [8]:

Definition 23 (Interpretation of the λ-calculus in $\mathcal{X}$).

$$\begin{aligned}
[\![x]\!]^\lambda_\alpha &\triangleq \langle x\cdot\alpha\rangle \\
[\![\lambda x.M]\!]^\lambda_\alpha &\triangleq \widehat{x}[\![M]\!]^\lambda_\beta\widehat{\beta}\cdot\alpha && \beta \text{ fresh} \\
[\![MN]\!]^\lambda_\alpha &\triangleq [\![M]\!]^\lambda_\gamma\widehat{\gamma} \dagger \widehat{x}([\![N]\!]^\lambda_\beta\widehat{\beta}\ [x]\ \widehat{y}\langle y\cdot\alpha\rangle) && \gamma,\beta,x,y \text{ fresh}
\end{aligned}$$

Observe that every sub-net of $[\![M]\!]^{\lambda}_{\alpha}$ has exactly one free plug, and that this is precisely α. Moreover, notice that, in the λ-calculus, the output (i.e. result) is anonymous; where an operand 'moves' to carries a name via a variable, but where it comes from is not mentioned, since it is implicit. Since in $\mathcal{X}$, a net is allowed to return a result in more than one way, in order to be able to connect outputs to inputs we have to name the outputs; this forces a name on the output of an interpreted λ-term M as well, carried in the sub-script of $[\![M]\!]^{\lambda}_{\alpha}$; this name α is also the name of the current continuation, i.e. the name of the hole in the context in which M occurs.

Combining the interpretation of λ into $\mathcal{X}$ and $\mathcal{X}$ into π, we get yet another encoding of the λ-calculus into π [27, 26], one that preserves assignable simple types; as usual, the interpretation is parametric over a name.

Definition 24 (Interpretation of the λ-calculus in π via $\mathcal{X}$). The mapping $[\![\cdot]\!]^{\pi}_{\cdot} : \Lambda \rightarrow \pi$ is defined by: $[\![M]\!]^{\pi}_{\alpha} = \lceil [\![M]\!]^{\lambda}_{\alpha} \rceil_{\pi}$

Since in [8] it is shown that the interpretation $[\![\cdot]\!]^{\lambda}_{\cdot}$ preserves both reduction and types, the following result is immediate:

Corollary 25 (Simulation of the Lambda Calculus).

1. If $M \rightarrow_{\beta} N$ then $[\![M]\!]^{\pi}_{\gamma} \pi\succeq [\![N]\!]^{\pi}_{\gamma}$.
2. If $\Gamma \vdash_{\lambda} M : A$, then $[\![M]\!]^{\pi}_{\alpha} :\cdot \Gamma \vdash_{\overline{\pi}} \alpha{:}A$.

Conclusion

We studied how to give the computational meaning to classical proofs via the π-calculus. Our results have been achieved in two steps: (1) we have encoded $\mathcal{X}$ into π enriched with pairing and non-blocking input, and showed that the encoding preserves interesting semantic properties; (2) we have defined a novel and 'unusual' type system for π and proved that types are preserved by the encoding.

The caveat of the paper was to find the right intuition to reflect the computational meaning of *cut*-elimination in π. Essentially we have interpreted the input in π as 'witness' for the formulae on the left-hand side of the turnstyle in LK, and outputs as 'witnesses' for the right-hand side. Arrow-right in LK corresponds to an output channel that sends a pair of names, while arrow-left corresponds to a channel that inputs a pair of names (via the let constructor). The *cut*-elimination procedure is then interpreted as a forwarder that connects an input and an output via private channels that have the same type. Essentially, if we take the view that input are witnesses for fomulae on the left-hand side of the turnstyle in LK and output are witnesses for fomulae on the right-hand side of the turnstye in LK then the cut eliminates the same formulue on the right and on the left of the turnstyle. Thus the representation of a cut in π has to guarantee that the input's and the output's witness of formulae on the right and left-hand side of the turnstyle can communicate. This is achieved by using the concept of forwarder, that connects two processes with different inputs and outputs.

The work that naturally compares with ours is [24], where the encoding of CBV-$\lambda\mu$ is presented. In that paper, full abstraction is proved, but for natural deduction rather

than for the sequent calculus as treated in this paper. In order to achieve the full abstraction result, the authors have to introduce a notion of typed equivalence of Call-by-Value $\lambda\mu$. By contrast, we have tried to give a simple, intuitive compositional encoding of LK in π and we leave for future work to consider a restriction of π in order to make our result stronger. $\mathcal{X}$ is a calculus without application and substitution that is much easier to interpret in π; notice that we needed no continuation-style encoding to achieve our results.

In [10] an intuive relation between fragments of linear logic and π-calculus was studied; the results there do not compare with ours. The notion of correctness presented in that paper is not between the logical rules and π, but between π and the '*cut algebra*' which is essentially a dialect of π. Note also that they encode the linear logic as opposed to the implicative fragment of Classical Logic. In other work [3], the relationship with linear logic and game semantics is studied. Both linear logic and game semantics are outside the scope of this paper, yet we leave for future work the study of the relation of linear $\mathcal{X}$ (with explicit weakening and contraction) [36], and relate that with both game semantics and π without replication.

One of the main goals we aimed for with our interpretation was: if α does not occur free in P, and x does not occur free in Q, then both $\ulcorner P\widehat{\alpha} \dagger \widehat{x}Q\urcorner_\pi \rightarrow_\pi \ulcorner P\urcorner_\pi$ and $\ulcorner P\widehat{\alpha} \dagger \widehat{x}Q\urcorner_\pi \rightarrow_\pi \ulcorner Q\urcorner_\pi$. However, we have not achieved this; we can at most show that $\ulcorner P\widehat{\alpha} \dagger \widehat{x}Q\urcorner_\pi$ reduces to a process that contains $\ulcorner P\urcorner_\pi \mid \ulcorner Q\urcorner_\pi$. It is as yet not clear what this say about either $\mathcal{X}$, or LK, or π, or simply about the encoding. The problem is linked to the fact that π does not have an automatic *cancellation*: since communication is based on the exchange of channel names, processes that do not communicate with each other just 'sit next to each other'. In $\mathcal{X}$, a process that wants to be 'heard', but is not 'listened' to, disappears; this corresponds to a proof contracting to a proof, not to two non-connected proofs for the same sequent. But, when moving to *linear* $\mathcal{X}$, or $*\mathcal{X}$, studied in [36], this all changes. Since there reduction can generate non-connected nets, it seems promising to explore an encoding of $*\mathcal{X}$ in π.

References

1. M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. *JFP*, 1(4), 1991.
2. M. Abadi and A. Gordon. A Calculus for Cryptographic Protocols: The Spi Calculus. In *4th CCCS*, ACM Press, 1997.
3. S. Abramsky. Computational interpretations of linear logic. *TCS*, 111(1&2), 1993.
4. S. Abramsky and R. Jagadeesan. Games and full completeness for multiplicative linear logic. *JSL*, 59(2), 1994.
5. S. Abramsky. Proofs as processes. *TCS*, 135(1), 1994.
6. Z. M. Ariola and H. Herbelin. Minimal classical logic and control operators. In *ICALP'03*, *LNCS* 2719, 2003.
7. S. van Bakel, S. Lengrand, and P. Lescanne. The language $\mathcal{X}$: circuits, computations and classical logic. In *ICTCS'05*, *LNCS* 3701, 2005.
8. S. van Bakel and P. Lescanne. Computation with classical sequents. *MSCS*, 2008.
9. H. Barendregt. *The Lambda Calculus: its Syntax and Semantics*. North-Holland, Amsterdam, revised edition, 1984.
10. G. Bellin and P. J. Scott. *On the pi-Calculus and Linear Logic*. *TCS*, 135(1), 11–65, 1994.
11. T. Coquand and G. Huet. The Calculus of Constructions. *IAC*, 76(2,3), 1988.

12. P.-L. Curien and H. Herbelin. The Duality of Computation. In *ICFP'00*, ACM, 2000.
13. H.B. Curry and R. Feys. *Combinatory Logic*, volume 1. North-Holland, Amsterdam, 1958.
14. N. G. de Bruijn. A namefree lambda calculus with facilities for internal definition of expressions and segments. TH-Report 78-WSK-03, University of Eindhoven, 1978.
15. G. Gentzen. Untersuchungen über das Logische Schliessen. *Math. Zeitschrift*, 39, 1935.
16. J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.
17. J.Y. Girard. The System F of Variable Types, Fifteen years later. *TCS*, 45, 1986.
18. J.-Y. Girard. A new constrcutive logic: classical logic. *Mathematical Structures in Computer Science*, 1(3):255–296, 1991.
19. T. Griffin. A formulae-as-types notion of control. In *POPL'90*, ACM, 1990.
20. H. Herbelin. Séquents qu'on calcule : de l'interprétation du calcul des séquents comme calcul de λ-termes et comme calcul de stratégies gagnantes. Thèse d'université, Paris 7, 1995.
21. H. Herbelin. C'est maintenant qu'on calcule: au cœur de la dualité. Mémoire de habilitation, Université Paris 11, Décembre 2005.
22. K. Honda and M. Tokoro. An object calculus for asynchronous communication. In *ECOOP'91*, LNCS 512, 133–147, 1991.
23. K. Honda and N. Yoshida. On the Reduction-based Process Semantics. *TCS*, 151:437–486, 1995.
24. K. Honda, N. Yoshida, and M. Berger. Control in the π-calculus. In *CW'04*, 2004.
25. J.W. Klop. Term Rewriting Systems. In *Handbook of Logic in Computer Science*, volume 2, chapter 1, pages 1–116. Clarendon Press, 1992.
26. R. Milner. Function as processes. In *MSCS*, 2(2), 1992.
27. R. Milner. *Communicating and Mobile Systems: the π-calculus*. Cambridge University Press, 1999.
28. M. Parigot. An algorithmic interpretation of classical natural deduction. In *LPAR'92*, *LNCS* 624, 1992.
29. D. Sangiorgi and D. Walker. *The Pi-Calculus*. Cambridge University Press, 2003.
30. A.J. Summers. Extending lambda-mu with first class continuations. Manuscript, 2007.
31. H. Thielecke. *Categorical Structure of Continuation Passing Style*. PhD thesis, University of Edinburgh, 1997.
32. C. Urban. *Classical Logic and Computation*. PhD thesis, University of Cambridge, 2000.
33. C Urban. Strong Normalisation for a Gentzen-like Cut-Elimination Procedure'. In *TLCA'01*, *LNCS* 2044, 2001.
34. C. Urban and G. M. Bierman. Strong normalisation of cut-elimination in classical logic. *FI*, 45(1,2), 2001.
35. P. Wadler. Call-by-Value is Dual to Call-by-Name. In *ICFP'03*, ACM, 2003.
36. D. Žunić. Computing with Sequents and Diagrams in Classical Logic - Calculi ${}^*\mathcal{X}$, ${}^d\mathcal{X}$, and ${}^{\copyright}\mathcal{X}$. PhD thesis, ENS Lyon, 2007.