Abstract

We study the \( \pi \)-calculus, where the family of names is enriched with pairing, 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 \( \pi \), and show that reduction and type assignment are preserved. Since \( \mathcal{X} \) enjoys the Curry-Howard isomorphism for Gentzen’s calculus \( \text{LK} \), this implies that all proofs in \( \text{LK} \) have an encoding in \( \pi \). We then enrich the logic with the connectors \&, \lor and \neg, and show that also these can be represented in \( \pi \).

Introduction

In this paper we present an encoding of proofs of Gentzen’s \( \text{LK} \) [15] into the \( \pi \)-calculus, that respects cut-elimination as well as assignable types. The encoding of classical logic into \( \pi \)-calculus is attained by using the intuition of the calculus \( \mathcal{X} \), which gives a computational meaning to \( \pi \)-calculus as an alternative computational model for classical logic. A cut has only introduction rules, the only way to eliminate a cut.

1

\[ \Gamma \vdash A, \Delta \]

The relation between process calculi and classical logic is an interesting and very promising area of research (a very similar attempt, in a natural deduction context, can be found [20]), which should be developed further, widening further the road to practical application of classical logic in computation. The aim of this paper is to focus on linking \( \text{LK} \) and \( \pi \) via \( \mathcal{X} \); the main achievements of this paper are:

- an encoding of \( \mathcal{X} \) into \( \pi \) is defined, that preserves the operational semantics; the non-confluent nature of reduction in \( \mathcal{X}/\text{LK} \) is neatly reflected by the non-determinism of \( \pi \);
- the encoding preserves assignable types, effectively showing that all proofs in \( \text{LK} \) have an encoding in \( \pi \), thereby representing classical logic directly in a process calculus;
- to achieve the necessary non-standard notion of types (types do not contain channel information), the target calculus is enriched with pairing [2], and a notion of type assignment is defined which encompasses implication;
- in addition to [8], we treat the full classical logic, including the connectives \( \rightarrow, \neg, \& \), and \( \lor \), not only for \( \mathcal{X} \), but also for \( \pi \).

Classical sequents

The sequent calculus, \( \text{LK} \) 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 derives statements with a single conclusion, whereas \( \text{LK} \) allows for multiple conclusions, deriving sequents of the form \( A_1, \ldots, A_n \Rightarrow B_1, \ldots, B_m \). Implicational \( \text{LK} \) has four rules: axiom, left introduction of the arrow, right introduction, and cut.

\[ (\text{Ax}) : \Gamma \vdash A, \Delta \]

\[ (\Rightarrow L) : \Gamma \vdash A, \Delta \quad \Gamma, B \vdash \Delta \]

\[ (\Rightarrow R) : \Gamma, A \Rightarrow B, \Delta \]

\[ (\text{cut}) : \Gamma \vdash A, \Delta \quad \Gamma, A \vdash \Delta \]

Since \( \text{LK} \) 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 \( (\text{cut}) \)-rule.

Gentzen defined a cut-elimination procedure that eliminates all applications of the \( (\text{cut}) \)-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 rewriting steps, i.e., local reductions of the proof-tree, which has –with some discrepancies– the flavour of the evaluation of explicit substitutions [14, 1].

The principle of \( \mathcal{X} \)

The calculus \( \mathcal{X} \) achieves a Curry-Howard isomorphism for the proofs in \( \text{LK} \) by constructing witnesses (called nets) for derivable sequents, without any notion of substitution or application. In establishing the isomorphism for \( \mathcal{X} \), similar to calculi like \( \lambda \mu \) [24] and \( \lambda \mu \) [12], Roman names are attached to formulae in the left context, and Greek names for those on the right. Names on the left can be seen as inputs to the net, and names to the right as outputs; since multiple formulae can appear on both sides. This implies that

[copyright notice will appear here]
a net can not only have more than one input, but also more than one output. There are two kinds of names in \( \mathcal{X} \): sockets (inputs, with Roman names, that are reminiscent of values) and plugs (outputs, with Greek names, that are reminiscent of continuations), that correspond to variables and co-variables, respectively, in [31], or, alternatively, to Parigot’s \( \lambda \)- and \( \mu \)-variables [24] (see also [12]). 

In the construction of the witness, when in applying a rule a premise or conclusion disappears from the sequent, the corresponding name gets bound in the net that is constructed, and when a premise or conclusion gets created, a different free (often new) name is associated to it. For example, in the creation of the net for right introduction of the arrow

\[
P : \Gamma, x : A \vdash \alpha : B, \Delta \\
\xi P \alpha \cdot \beta : \Gamma \vdash : \beta : \alpha \to B \Delta
\]

the input \( x \) and the output \( \alpha \) are bound, and \( \beta \) is free. This case is interesting in that it highlights a special feature of \( \mathcal{X} \), not found in other calculi. In (applicative) calculi related to natural deduction, the \( \lambda \)-calculus, only inputs are named, and the linking to a term that will be inserted is done via \( \lambda \)-abstraction and application. The output (i.e. result) on the other hand is anonymous; where a term ‘moves to’ carries a name via a variable, but where it comes from is not mentioned, since it is implicit. Since a term \( P \) can have many inputs and outputs, it is unsound to consider \( P \) as a function; however, fixing one input \( x \) and one output \( \alpha \), we can see \( P \) as a function from \( x \) to \( \alpha \). We make this limited view of \( P \) available via the output \( \beta \), thereby exporting \( P \) as a function from \( x \) to \( \alpha \): notice that the types given to the connectors confirm this view.

Gentzen’s proof reductions by cut-elimination become the fundamental principle of computation in \( \mathcal{X} \). Cuts in proofs are witnessed by \( \xi P \alpha \cdot \beta \) (called the cut of \( P \) and \( Q \) via \( \alpha \) and \( x \)), and the reduction rules specify how to remove them: a term is in normal form if and only if it has no sub-term of this shape. The intuition behind reduction is: the cut \( \xi P \alpha \cdot \beta \) expresses the intention to connect all \( \alpha \) in \( P \) and all \( \alpha \) in \( Q \), and reduction will realise this by either connecting all \( \alpha \) to all \( \alpha \) (if \( x \) does not exist, \( P \) will disappear), or all \( \alpha \) to all \( \alpha \) (if \( \alpha \) does not exist, \( Q \) will disappear). Since cut-elimination in \( \mathcal{LX} \) is not confluent, neither is reduction in \( \mathcal{X} \); for example, as suggested above, when \( P \) does not contain \( \alpha \) and \( Q \) does not contain \( x \), reducing \( \xi P \alpha \cdot \beta \) leads to both \( P \) and \( Q \), two different nets.

\[
P \dashv P \xi \alpha \cdot \beta Q \to \gamma Q
\]

Reduction in \( \mathcal{X} \) boils down to renaming: since the calculus is substitution-free, during reduction nets are re-organised, creating nets that are similar, but with different connector names inside. This view is central in our encoding of cut-elimination in \( \mathcal{LX} \).

As an illustration of reduction in \( \mathcal{X} \), consider the following reductions (the net \( \{x\alpha\} \) is a witness for the axiom in \( \mathcal{LK} \)):

\[
\begin{align*}
(x \alpha \alpha \alpha \alpha) \xi \gamma (y \beta) & \to (x \beta) \\
(x \alpha \alpha \alpha \alpha) \xi \gamma (y \beta) & \to (x \alpha) \\
(x \alpha \alpha) \xi \gamma (y \beta) & \to (y \beta) \\
(x \alpha \alpha) \xi \gamma (y \beta) & \to (x \alpha) \\
\end{align*}
\]

Notice the change, between the various lines, in connector names involved in the cut, and that the last variant has two outcomes, underlining the non-confluence of \( \mathcal{X} \).

**Capturing \( \mathcal{X} \) in \( \pi \)**

\( \mathcal{X} \)’s notion of multiple inputs and outputs is also found in \( \pi \), and was the original inspiration for our research. The aim of this work is to find an accurate encoding of \( \mathcal{LX} \)-proofs in \( \pi \), and to devise a notion of type assignment for \( \pi \) so that the encoding would preserve the types. To achieve this, we made full use of the view of \( \mathcal{X} \)-nets sketched above. Clearly this implies that we had to define a notion of type assignment that uses the type constructor \( \to \) for the full \( \pi \)-calculus, without having to linearise the calculus as done in [20], and this is one of the contributions of this paper. Although the calculi \( \mathcal{X} \) and \( \pi \) are, of course, essentially different, the similarities go beyond the correspondence of inputs and output between nets in \( \mathcal{X} \) and processes in \( \pi \). Like \( \mathcal{X} \), \( \pi \) 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.

Since in \( \mathcal{X} \) a net can return a result in more than one way, in order to be able to connect outputs to inputs we also have to name outputs. So when creating a witness for \( (=R) \) (the net \( \xi P \alpha \cdot \beta \), called an export), the exported interface of \( P \) is the functionality of ‘receiving on \( x \), sending on \( \alpha \)’, which is made available on \( \beta \). When encoding this behaviour in \( \pi \), 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 \( \alpha \), breaking perhaps the exported functionality, and certainly disabling the possibility of assigning arrow types. We can overcome this problem by sending out a pair of names. Similarly, when interpreting a witness for \( (=L) \) (the net \( \xi P \alpha \cdot \beta \), called an import), the circuit that is to be connected to \( x \) is ideally a function whose input will be connected to \( \alpha \), and its output to \( y \). This means that we need to receive a pair of names over \( x \).

Perhaps surprisingly, adding pairing does not just give sufficient expressivity to encode implicational \( \mathcal{X} \) (and thus implicative \( \mathcal{LX} \)), but also to encode the other first-order logical connectives. To the best of our knowledge, this is the first time this result is presented, and represents a real achievement since we are able to show that the \( \pi \)-calculus is an adequate computational model for \( \mathcal{LX} \).

A cut \( \xi P \alpha \cdot \beta \) in \( \mathcal{X} \) expresses two nets that need to be connected via \( \alpha \) and \( x \). If we model \( P \) and \( Q \) in \( \pi \), then we obtain one process sending on \( \alpha \), and one receiving on \( x \), and we need to link these via \( \{x\alpha\} \). Since each output on \( \alpha \) in \( P \) takes place only once, and \( Q \) might want to receive in more than one, 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 \( \alpha \), \( Q \) needs to be replicated. Considering the reduction in \( \mathcal{X} \) shown above, we are able to show that (where \( \triangleq \) is a simulation relation, defined in Definition 3.4):

\[
\begin{align*}
\{x\alpha\} \xi \gamma (y \beta) & \triangleq \{x\alpha\} \xi \gamma (y \beta) \\
\{x\alpha\} \xi \gamma (y \beta) & \triangleq \{x\alpha\} \xi \gamma (y \beta) \\
\{x\alpha\} \xi \gamma (y \beta) & \triangleq \{x\alpha\} \xi \gamma (y \beta) \\
\{x\alpha\} \xi \gamma (y \beta) & \triangleq \{x\alpha\} \xi \gamma (y \beta)
\end{align*}
\]

The last alternative represents the fact that the two possible reductions in \( \mathcal{X} \) are represented by the composition of their translation.

**Related work**

The relation between logic and computation hinges around the Curry-Howard isomorphism (sometimes also attributed to De Bruijn), which expresses the fact that, for certain calculi with a notion of types, one can find a corresponding logic such that it is possible to associate terms with proofs in such a way that types become propositions, and proof contractions become term reductions (or computations). This phenomenon was first discovered for Combinatory Logic [13].

In the past, say before Herbelin’s PhD [18] and Urban’s PhD [30], the study of the relation between computation, programming languages and logic has concentrated mainly on natural deduction systems. This holds most strongly in the context of non-classical logics; for example, the Curry-Howard relation between Intuition-
istic 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 [16] and the Calculus of Constructions [11]. Abramski [3, 5] has studied correspondence between multiplicative linear logic and processes, and later moved to the context of game semantics [4].

The link between Classical Logic and continuations and control was first established for the λc-Calculus [17] (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 ⊥; for a detailed discussion, see [26]). This imbalance can be observed in Parigot’s λµ-calculus [24], 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 ⊥ as pseudo-type (only negation, or A→⊥, is expressed; ⊥→A is not a type), the λµ-calculus corresponds to minimal classical logic [6].

Herbelin [18, 12, 19] has studied the calculus λµυ as a non-applicative extension of λµ, which gives a fine-grained account of manipulation of sequents. The relation between call-by-name and call-by-value in the fragment of I.K with negation and conjunction is studied in [31]; as in calculi like λµυ and λµυυ, the Dual Calculus considers a logic with active formulae, so these calculi do not achieve the Curry-Howard isomorphism with I.K.

The π-calculus is equipped with a rich type theory [25]: from the basic type system for counting the arity of channels to sophisticated linear types [20], which studies a relation between Call-by-value and Applicative extension of the basic type system for counting the arity of channels to sophisticated linear types [20], which studies a relation between Call-by-name and Applicative extension of the basic type system for counting the arity of channels to sophisticated linear types [20], which studies a relation between Call-by-name and Applicative extension of the basic type system for counting the arity of channels to sophisticated linear types [20], which studies a relation between Call-by-name and Applicative extension of the basic type system for counting the arity of channels to sophisticated linear types [20], which studies a relation between Call-by-name and Applicative extension of the basic type system for counting the arity of channels to sophisticated linear types [20], which studies a relation between Call-by-name and Applicative extension of the basic type system for counting the arity of channels to sophisticated linear types [20], which studies a relation between Call-by-name and Applicative extension of the basic type system for counting the arity of channels to sophisticated linear types [20], which studies a relation between Call-by-name and Applicative extension of the basic type system for counting the arity of channels to sophisticated linear types [20], which studies a relation between Call-by-name and Applicative extension of the basic type system for counting the arity of channels to sophisticated linear types [20], which studies a relation between Call-by-name and Applicative extension of the basic type system for counting the arity of channels to sophisticated linear types [20], which studies a relation between Call-by-name and Applicative extension of the basic type system for counting the arity of channels to sophisticated linear types [20], which studies a relation between Call-by-name and Applicative extension of the basic type system for counting the arity of channels to sophisticated linear types [20], which studies a relation between Call-by-name and Applicative extension of the basic type system for counting the arity of channels to sophisticated linear types [20], which studies a relation between Call-by-name and Applicative extension of the basic type system for counting the arity of channels to sophisticated linear types [20], which studies a relation between Call-by-name and Applicative extension of the basic type system for counting the arity of channels to sophisticated linear types [20], which studies a relation between Call-by-name and Applicative extension of the basic type system for counting the arity of channels to sophisticated linear types [20], which studies a relation between Call-by-name and Applicative extension of the basic type system for counting the arity of channels to sophisticated linear types [20], which studies a relation between Call-by-name and Applicative extension of the basic type system for counting the arity of channels to sophisticated linear types [20], which studies a relation between Call-by-name and Applicative extension of the basic type system for counting the arity of channels to sophisticated linear types [20], which studies a relation between Call-by-name and Applicative extension of the basic type system for counting the arity of channels to sophisticated linear types [20], which studies a relation between Call-by-name and Applicative extension of the basic type system for counting the arity of channels to sophisticated linear types [20], which studies a relation between Call-by-name and Applicative extension of the basic type system for counting the arity of channels to sophisticated linear types [20], which studies a relation between Call-by-name and Applicative extension of the basic type system for counting the arity of channels to sophisticated linear types [20], which studies a relation between Call-by-name and Applicative extension of the basic type system for counting the arity of channels to sophisticated linear types [20], which studies a relation between Call-by-name and Applicative extension of the basic type system for counting the arity of channels to sophisticated linear types [20], which studies a relation between Call-by-name and Applicative extension of the basic type sys...
The next rules define how to move an activated dagger inwards.

**Definition 1.5 (Propagation rules). Left propagation:**

\[
\begin{align*}
&\text{(d) : } (y\cdot)\alpha \frac{\not\exists P}{\not\exists P} \quad \Gamma, y : A = \Gamma \cup \{x:A\}, \text{ if } \Gamma \text{ is not defined on } x \\
&\text{otherwise}.
\end{align*}
\]

(Notice that the second case implies that \(x:A \notin \Gamma\). So, when writing a context as \(\Gamma, y : A\), this implies that \(x:A \in \Gamma\), or \(\Gamma\) is not defined on \(x\). When we write \(\Gamma_1, \Gamma_2\) we mean the union of \(\Gamma_1\) and \(\Gamma_2\), provided \(\Gamma_1\) and \(\Gamma_2\) are compatible (if \(\Gamma_1\) contains \(:A_1\) and \(\Gamma_2\) contains \(:A_2\) then \(A_1 = A_2\)).

3. Contexts of plugs \(\Delta\) are defined in a similar way.

The notion of type assignment on \(\mathcal{X}\) that we present in this section is the basic implicational system for Classical Logic (Gentzen 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 \(\alpha, \beta, \gamma, \) 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 different connector will be associated to it.

**Definition 2.2 (Typing for \(\mathcal{X}\)).** 1. Type judgements are expressed via a ternary relation \(P : \Gamma, \Delta\), where \(\Gamma\) is a context of sockets and \(\Delta\) 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:

\[
\begin{align*}
&\text{(cap) : } (y\cdot) : \Gamma, y : A, \Delta & : A, \Delta \\
&\text{(imp) : } P : \Gamma, \Delta & : A, \Delta \quad Q : \Gamma, x : B, \Delta \\
&\text{(exp) : } P : \Gamma, x : A, \Delta & : B, \Delta \\
&\text{(cut) : } P \vdash \Gamma, x : A, \Delta & : Q \\
&\text{We write } \not\exists P \text{ for the (reflexive, transitive, compatible) reduction relation generated by the logical propagation and activation rules.}
\end{align*}
\]

The reduction relation \(\not\exists P\) is not confluent; this comes in fact from the critical pair that activates a cut \(P\not\exists P\) in two ways.

**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, as in \(P\not\exists Q\) or \((z\cdot)\not\exists P\); performing the elimination cuts, via \((\chi\text{cap})\) or \((\chi\text{f} P)\), will remove the term \(P\).

In [8, 9] some basic properties are shown, which essentially show that the calculus is well-behaved, as well as the relation between \(\mathcal{X}\) and a number of other calculi. These results motivate the formulation of admissible rules:

**Lemma 1.6 (Garbage Collection and Renaming [9]).**

\[
\begin{align*}
&\text{(gc) : } P\not\exists Q \not\exists P & \quad \text{if } \alpha \notin \emptyset(P) \\
&\text{(gc) : } P\not\exists Q \not\exists P & \quad \text{if } \alpha \notin \emptyset(P) \\
&\text{(ren-L) : } P\not\exists Q \not\exists P & \quad \text{if } \alpha \notin \emptyset(P) \\
&\text{(ren-R) : } P\not\exists Q \not\exists P & \quad \text{if } \alpha \notin \emptyset(P)
\end{align*}
\]

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 2.1 (Types and Contexts).** 1. The set of types is defined by the grammar:

\[
A, B ::= \varphi \mid A \rightarrow B.
\]

where \(\varphi\) is a basic type of which there are infinitely many.

2. A context of sockets \(\Gamma\) is a finite set of statements \(x:A\), such that the subject of the statements (x) are distinct. We write \(\Gamma, x : A\) for the context defined by:

\[
\Gamma, x : A = \Gamma \cup \{x:A\}, \text{ if } \Gamma \text{ is not defined on } x \\
= \Gamma, \text{ otherwise}.
\]

The following is proven in [8]:

\[
\begin{align*}
&\text{(Ax) } A \rightarrow A, B \\
&\vdash A \rightarrow B, A \quad (\Rightarrow R) \\
&\vdash A \rightarrow A \quad (\Rightarrow L) \\
&\vdash (A \rightarrow B) \rightarrow A \quad (\Rightarrow R)
\end{align*}
\]

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

\[
\begin{align*}
&\text{(cap) : } (y\cdot) : y : A \rightarrow \Delta \\
&\vdash y : A \rightarrow \Delta \quad (\Rightarrow R) \\
&\vdash (y : A \rightarrow \Delta) \rightarrow A \quad (\Rightarrow L) \\
&\vdash y : A \rightarrow \Delta \quad (\Rightarrow R) \\
&\vdash (y : A \rightarrow \Delta) \rightarrow A \quad (\Rightarrow L)
\end{align*}
\]

The following is proven in [8]:

\[
\begin{align*}
&\text{(Ax) } A \rightarrow A, B \\
&\vdash A \rightarrow B, A \quad (\Rightarrow R) \\
&\vdash (A \rightarrow B) \rightarrow A \quad (\Rightarrow L)
\end{align*}
\]
3. The π-calculus with pairing

The notion of π-calculus that we consider in this paper is slightly different from other systems studied in the literature. The reason for this change lies directly in the calculus that is going to be interpreted, X: since we are going to model sending and receiving pairs of names as interfaces for functions, we consider the π-calculus with pairing, inspired by [2]. We take the view that processes communicate by sending data over channels, not just names.

To ease the definition of the interpretation function of circuits in X to processes in the π-calculus, we deviate slightly from the normal practice, and write either Greek characters α, β, υ, . . . or Roman characters x, y, z, . . . for channel names; we use n for either a Greek or a Roman name, and ‘·’ 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 3.1. Channel names and data are defined by:**

\[
\begin{align*}
\alpha, \beta, \upsilon, \ldots & ::= x \mid \alpha \quad \text{names} \\
p & ::= a \mid (a, b) \quad \text{data}
\end{align*}
\]

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

\[
\begin{align*}
P, Q & ::= 0 \quad \text{Nil} \\
| P & | Q \quad \text{Composition} \\
| & P \quad \text{Replication} \\
| (\upsilon a) & P \quad \text{Restriction} \\
| P + Q & \quad \text{Choice} \\
| a(x) & . P \quad \text{Input} \\
| \upsilon(p). & P \quad \text{Output} \\
| \text{let} & (x, y) = z \text{ in } P \quad \text{Let construct}
\end{align*}
\]

We abbreviate \(a(x) . \text{let} (y, z) = x \text{ in } P\) by \(a(y, z) \cdot P\), and \((\upsilon n)(\upsilon n). P\) by \((\upsilon n, n) \cdot P\).

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

**Definition 3.2 (Congruence).** The structural congruence is the smallest equivalence relation closed under contexts defined by the following rules:

\[
\begin{align*}
P & \equiv 0 \equiv P \\
| P & | Q \equiv | P | Q \\
(P | Q) & | R \equiv P | (Q | R) \\
| P & | P \equiv | P | P \\
(\upsilon n) & 0 \equiv 0 \\
(\upsilon n) & (\upsilon m) \equiv (\upsilon m) (\upsilon n) P \\
(\upsilon n) & (P | Q) \equiv (\upsilon n) (P | Q) \quad \text{if } n \notin \text{fn}(P) \\
P + Q & \equiv Q + P \\
(P + Q) & | R \equiv P + (Q | R) \\
\text{let} (x, y) & = (a, b) \text{ in } R \equiv R(a/x, b/y)
\end{align*}
\]

**Definition 3.3. 1. The reduction relation over the processes of the π-calculus is defined by following (elementary) rules:**

- (synchronisation): \(\sigma(b) . P + G \Downarrow |a(x) . Q + S \Rightarrow P | Q[b/x]\)
- (choice): \(P \rightarrow_n P' \Rightarrow P + Q \rightarrow_n P'\)
- (binding): \(P \rightarrow_n P' \Rightarrow (\upsilon n) P \rightarrow_n (\upsilon n) P'\)

\[\begin{align*}
\text{(composition)} : & \quad P \rightarrow_n P' \Rightarrow P \mid Q \rightarrow_n P' \mid Q \\
\text{(congruence)} : & \quad P \equiv Q \& Q \rightarrow_n Q' \& Q' \equiv P' \Rightarrow P \rightarrow_n P'
\end{align*}\]

2. We write \(\rightarrow^*_n\) for the reflexive and transitive closure of \(\rightarrow_n\).

3. We write \(P \rightarrow \) if \(P \equiv (\upsilon p_1 \ldots p_m) (a. R + G | Q)\), where \(a = \upsilon n(b) \text{ or } a = n(x) \text{ and } n \notin p_1 \ldots p_m\) for some \(R, G, Q\).

4. We write \(Q \rightarrow \) if there exists \(P\) such that \(Q \rightarrow^*_n P \text{ and } P \rightarrow \).

Notice that

\[\pi(b, c). P \mid a(x, y). Q \rightarrow^*_n P \mid Q[b/x, c/y]\]

**Definition 3.4. Barbed contextual simulation is the largest relation \(\gtrsim_n\) such that \(P \gtrsim_n Q\) implies:

- for each name \(n\), if \(P \rightarrow_n \) then \(Q \gtrsim_n\);
- for any context \(C\), if \(C[P] \rightarrow_n P'\), then for some \(Q', C[Q] \rightarrow_n^* Q'\) and \(P' \gtrsim_n Q'\).

4. Type assignment

In this section, we introduce a notion of type assignment for processes in \(\pi\), 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:

\[P \vdash \Gamma, b : A \rightarrow \Delta\]

\[\pi(b). P \vdash \Gamma, b : A \rightarrow \alpha : \text{ch}(A), \Delta\]

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

As for the notion of type assignment on \(X\) terms, in the typing judgement 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 4.1 (Type assignment).** The types and contexts we consider for the π-calculus are defined like those of Definition 2.1, generalised to names. Type assignment for π-calculus is defined by the following sequent system:

\[
\begin{align*}
(0) & : 0 \vdash \Gamma \vdash \Delta \\
(\nu) & : \nu a : A \vdash \Delta \vdash \Delta \\
(\upsilon n) & : \upsilon n : P \vdash \Gamma[\nu n] \vdash \Delta \\
(\upsilon n) & : (\upsilon m) (\upsilon n) P \vdash \Gamma[\nu m, \nu n] \vdash \Delta \\
(\upsilon n) & : (\upsilon n) (P | Q) \equiv \Gamma[\nu n] \vdash \Delta \\
(\upsilon n) & : P + Q \equiv \Gamma[\nu n] \vdash \Delta \\
\text{let} (x, y) & = (a, b) \text{ in } R \equiv \Gamma[\alpha/x, \beta/y] \\
\text{(in)} & : a(x) . P \vdash \Gamma, a : A \rightarrow \Delta \\
\text{(out)} & : \alpha(b). P \vdash \Gamma, b : A \rightarrow \Delta \\
\text{(pair-out)} & : \pi(b, c). P \vdash \Gamma, b, c : A \rightarrow \Delta
\end{align*}
\]
As usual, we write $P : \Gamma \vdash x : A$ if there exists a derivation using these rules that has this expression in the conclusion, and write $D : P : \Gamma \vdash \Delta$ if we want to name that derivation.

Notice that $t$ is possible to derive $\pi(a) : \vdash x : A$.

The 'input-output interface of a $\pi$-process' property is nicely preserved by all the rules; it also explains how the handling of pairs is restricted by the type system in to the rules (let) and (pair-out).

It should be remarked that this notion of type assignment does not (directly) relate back to $\lambda K$. For example, rules (|) and (+) do not change the contexts, so do not correspond to any rule in the logic, not even to a $\lambda K$-style activation step. Moreover, rule ($\nu$) just removes a formula; it would perhaps be better to have syntax that expresses sending a private name, as used in [20]. E.g., we could write $\pi(b).P$ for the process ($\nu b$)($\pi b$).P; notice the use of '$\cdot$' rather than '$\cdot$'. This would justify the derivation rules

\[
\text{(derivation rules):} \\
\begin{align*}
\text{(out') :} & \quad P : \Gamma, b:A \vdash b:A, \Delta \\
\text{(pair-out') :} & \quad P : \Gamma, b:A \vdash c:B, \Delta \\
\end{align*}
\]

which have a clearer logical context. However, this would not suffice in our case since we cannot encode, for example, $\tilde{x}(x\alpha)\tilde{\gamma}b$ by $x()$, $\pi x\alpha(\pi x\alpha)\tilde{x}(x\alpha)$, since that would imply that $x$ and $\alpha$ occur both free and bound.

**Example 4.2.** We can derive

\[
\begin{align*}
&\vdash P : \Gamma, y:B \vdash x:A, \Delta \\
&\vdash a(z), \text{let } (x,y) = z \text{ in } P : \Gamma, a:A \vdash B \vdash \Delta \\
&\text{so the following rule is derivable:} \\
&\text{(pair-in') :} \quad P : \Gamma, y:B \vdash x:A, \Delta \\
&\quad P : \Gamma, a:A \vdash B \vdash \Delta
\end{align*}
\]

Notice that the rules (pair-out) correspond to the logical rules ($\Rightarrow R$) and ($\Rightarrow L$).

**Lemma 4.3 (Weakening).** The following rule is admissible:

\[
(W) : \quad P : \Gamma \vdash \Delta \quad \vdash \Gamma' \vdash \Delta' \quad \text{(} \Gamma' \supseteq \Gamma, \Delta' \supseteq \Delta \text{)}
\]

This result allows us to be a little less precise when we construct derivations, and allow for rules to join contexts, by using, for example, the rule

\[
(\|) : \quad P : \Gamma_1 \vdash \Delta_2 \quad Q : \Gamma_1 \vdash \Delta_2 \\
\quad P \| Q : \vdash \Gamma_1, \Delta_1, \Delta_2
\]

In what follows, we will use the following abbreviation:

\[
D_{\alpha}(\pi) : A : \quad \begin{align*}
&\vdash 0 : \Gamma, \vdash A: A, \Delta \\
&\vdash a(\cdot), b(\cdot) : \Gamma, a:A \vdash b:A, \Delta
\end{align*}
\]

We have a soundness (witness reduction) result, for which we first need to prove a substitution lemma.

**Lemma 4.4 (Substitution).** If $P : \Gamma, x:A \vdash x:A, \Delta$ then also $P[b/x] : \Gamma, b:A \vdash b:A, \Delta$.

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

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

**Theorem 4.5 (Witness reduction).** If $P : \Gamma \vdash \Delta$ and $P \rightarrow \Delta Q$, then $Q : \Gamma \vdash \Delta$.

5. **Interpreting $\lambda \pi$ into $\pi$**

In this section, we define an encoding from nets in $\lambda \pi$ onto processes in $\pi$. Since in $\pi$, it is impossible to reduce under an input, we cannot fully encode reduction in $\lambda \pi$, but have to limit the notion of reduction, in that reduction is not possible under an import. We show that this limited reduction in $\lambda \pi$ is preserved by the encoding, as well as type assignment.

The encoding defined below is based on the intuition as formulated in [8]; the cut $P\tilde{\gamma}Q\tilde{\gamma} \tilde{\gamma}Q$ expresses the intention to connect all $\alpha$s in $P$ and $\alpha$s in $Q$, and reduction will realise this by either connecting all $\alpha$s to all $x$s, or all $x$s to all $\alpha$s. Translated into $\pi$, this results in seeing the right-activated cut as $P$ trying to send at least as many times over $\alpha$ as $Q$ is willing to receive over $x$, and seeing the left-activated cut as $Q$ trying to receive at least as many times over $x$ as $P$ is ready to send over $\alpha$. The inactive cut is interpreted as the choice between these alternatives; this of course implies that a logical cut will be ‘propagated’, but the result will be the one desired.

As mentioned in the introduction, we add pairing to the $\pi$-calculus in order to be able to deal with arrow types. Notice that using the polyadic $\pi$-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 $\tilde{x}(P\tilde{\gamma}\tilde{\gamma}X\tilde{\gamma})$ to that of $\tilde{x}P\tilde{\gamma}X\tilde{\gamma}$ (with $\beta$ not free in $P$). So, choosing to encode the export of $x$ and $\alpha$ over $\beta$ as $\tilde{\gamma}X\tilde{\gamma}$ would force the interpretation of $\tilde{x}(\tilde{\gamma}X\tilde{\gamma})$ 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 $\pi$-calculus, we opt for the following: communication will take place sending 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.

In the definition below, we use $\tilde{x}$ for the generic variable, to separate plugs and sockets (and their interpretation) from the ‘internal’ variables of $\pi$. Also, although the departure point is to view Greek names for outputs and Roman names for inputs, by the very nature of the $\pi$-calculus, 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 $\alpha$ is to be received as input on $x$’ via $\alpha(\pi)\pi(\cdot)$, which for convenience is abbreviated as $\alpha=x$.

**Definition 5.1.** The interpretation of circuits is defined by:

\[
\begin{align*}
\tau(x,\alpha) &\equiv (x,\pi(\cdot)) \\
\tau(y,\beta) &\equiv \langle y, \beta \rangle (\langle y, \beta \rangle) \\
\tau P\tilde{\alpha} \tilde{\gamma}Q &\equiv (x,(\alpha)\langle \langle y, \beta \rangle \rangle)) + \\
\tau P\tilde{\alpha} \tilde{\gamma}Q &\equiv (\langle (x, \alpha) \rangle \langle (x, \alpha) \rangle) + \\
\tau P\tilde{\alpha} \tilde{\gamma}Q &\equiv (\langle (x, \alpha) \rangle \langle (x, \alpha) \rangle) +
\end{align*}
\]

Notice that the interpretation of the inactive cut is the choice between the interpretations of the activated cuts.
The need to restrict reduction in $\pi$, as mentioned in the beginning of this section, is clear after the definition of encoding. The alternative for the import $P\!\vdash [x] \overrightarrow{y} Q$ creates a process that inputs a pair, over a combination of processes, including the interpretation of $P$ and $Q$; therefore, all cuts that appear in either $P$ or $Q$ are inactive after the interpretation. Since $P \vdash_{\pi} P'$ and $Q \vdash_{\pi} Q'$, then $P\!\vdash [x] \overrightarrow{y} Q \vdash \overrightarrow{P} \overrightarrow{\alpha} \overrightarrow{\beta} [x] \overrightarrow{y} Q'$, this reduction cannot be mimicked by the encoding, and therefore has to be blocked. This does in no way impede the full interpretation of proofs in $\pi_k$ in $\pi$, it just implies that we cannot simulate full cut-elimination.

**Example 5.2.** The encoding of the witness of Peirce’s law becomes:

\[
\pi' (\langle x \overrightarrow{y} \overrightarrow{\delta} \overrightarrow{\alpha} \overrightarrow{\beta} \overrightarrow{\pi} \overrightarrow{\tau} \overrightarrow{\delta} \rangle) \overrightarrow{\tau} \overrightarrow{\pi} = \\
(\nu \alpha) (\nu \beta) (\nu \gamma) (\nu \delta) (\nu x) (\nu y) (\nu z) (\nu w) (\nu \phi)
\]

This that this is a witness of $(\{A - B\} \rightarrow A) \rightarrow A$ is a straightforward application of Theorem 5.5 below.

One of the main goals we aimed for with our interpretation was: if $\alpha$ does not occur free in $P$, and $x$ does not occur free in $Q$, then both $P\!\vdash \overrightarrow{\pi} [x] \overrightarrow{y} Q \vdash \overrightarrow{\pi} P \overrightarrow{\alpha} \overrightarrow{\beta} Q$. However, we have not achieved this; we can at most show that $P\!\vdash \overrightarrow{\pi} [x] \overrightarrow{y} Q$ reduces to a process that contains $\overrightarrow{\pi} P \overrightarrow{\alpha} \overrightarrow{\beta} Q$. The correctness result for the encoding essentially states that the image of the encoding in $\pi$ contains some extra behaviour that can be disregarded. As the examples below show, this is mainly due to the presence of replicated processes in the translation of the cut. The precise formulation of the correctness lemma is stated below.

**Lemma 5.3 (Correctness).** If $P \vdash_{\pi} P'$, then for some $Q$, $\overrightarrow{\pi} P \overrightarrow{\alpha} \overrightarrow{\beta} Q$ and $\overrightarrow{\pi} P' \overrightarrow{\alpha} \overrightarrow{\beta} Q$.

**Example 5.4.** We check two examples from the introduction.

- $(x \overrightarrow{\alpha} \overrightarrow{\beta} \overrightarrow{\alpha} \overrightarrow{\beta})$:
  
  \[
  (x \overrightarrow{\alpha}) \overrightarrow{\tau} \overrightarrow{\pi} [x \overrightarrow{\beta}] \overrightarrow{\tau} \overrightarrow{\pi} = \\
  (\nu x) (\nu y) (\nu z) \overrightarrow{\pi} [x \overrightarrow{\beta}] \overrightarrow{\tau} \overrightarrow{\pi}
  \]

- $(x \overrightarrow{\alpha} \overrightarrow{\alpha} \overrightarrow{\beta} \overrightarrow{\alpha} \overrightarrow{\beta})$:

\[
\begin{align*}
(\nu x) (\nu y) (\nu z) \overrightarrow{\pi} [x \overrightarrow{\beta}] & \overrightarrow{\tau} \overrightarrow{\pi} \\
& = (\nu x) (\nu y) \overrightarrow{\pi} [x \overrightarrow{\beta}] \overrightarrow{\tau} \overrightarrow{\pi} \\
& = (\nu x) (\nu y) \overrightarrow{\pi} [x \overrightarrow{\beta}] \overrightarrow{\tau} \overrightarrow{\pi} \\
& = (\nu x) (\nu y) \overrightarrow{\pi} [x \overrightarrow{\beta}] \overrightarrow{\tau} \overrightarrow{\pi} \\
& = (\nu x) (\nu y) \overrightarrow{\pi} [x \overrightarrow{\beta}] \overrightarrow{\tau} \overrightarrow{\pi}
\end{align*}
\]

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

**Theorem 5.5.** If $P \vdash_{\pi} \overrightarrow{\pi} P \overrightarrow{\alpha} \overrightarrow{\beta} \Delta$, then $\overrightarrow{\pi} P \overrightarrow{\alpha} \overrightarrow{\beta} \Delta$.

**Proof.** By induction on the structure of circuits in $\pi$.

$(x \overrightarrow{\alpha})$ : Then $t(x \overrightarrow{\alpha})_{\pi} = x(x). \overrightarrow{\pi} (y)$, and the $\overrightarrow{\alpha}$-derivation:

\[
(x \overrightarrow{\alpha}) : \Gamma, x : A \vdash x A : \alpha : A, \Delta
\]

Notice that

\[
0 : \Gamma, x : A \vdash x A : \alpha : A, \Delta
\]

$\overrightarrow{\alpha}$ : Then the $\overrightarrow{\alpha}$-derivation is shaped like:

\[
\begin{align*}
P & \vdash \Gamma, x : A \vdash x A : B, \Delta \\
\overrightarrow{\alpha} & \vdash \Gamma, x : A \vdash x A : B, \Delta
\end{align*}
\]

Then, by induction, $\overrightarrow{\pi} P : \Gamma, x : A \vdash x A : B, \Delta$, and we can construct:

\[
\begin{align*}
P & \vdash \Gamma, x : A \vdash x A : B, \Delta \\
\overrightarrow{\pi} P & \vdash \Gamma, x : A \vdash x A : B, \Delta
\end{align*}
\]

$\overrightarrow{\pi} P \overrightarrow{\alpha} \overrightarrow{\beta} Q$ : Then the $\overrightarrow{\alpha \beta}$-derivation is shaped like:

\[
\begin{align*}
P & \vdash \Gamma, x : A \vdash x A : B, \Delta \\
\overrightarrow{\pi} P & \vdash \Gamma, x : A \vdash x A : B, \Delta
\end{align*}
\]

Then, by induction, we have derivations for $\overrightarrow{\pi} P : \Gamma, x : A \vdash x A$ and $\overrightarrow{\pi} Q : \Gamma, x : A \vdash x A$, and we can construct:

\[
\begin{align*}
P & \vdash \Gamma, x : A \vdash x A : B, \Delta \\
\overrightarrow{\pi} P & \vdash \Gamma, x : A \vdash x A : B, \Delta
\end{align*}
\]

$\overrightarrow{\pi} P \overrightarrow{\alpha} \overrightarrow{\beta} Q$ : Then the $\overrightarrow{\alpha \beta}$-derivation is shaped like:

\[
\begin{align*}
P & \vdash \Gamma, x : A \vdash x A : B, \Delta \\
\overrightarrow{\pi} P & \vdash \Gamma, x : A \vdash x A : B, \Delta
\end{align*}
\]

Then, by induction, we have derivations for $\overrightarrow{\pi} P : \Gamma, x : A \vdash x A$ and $\overrightarrow{\pi} Q : \Gamma, x : A \vdash x A$, and we can construct:

\[
\begin{align*}
P & \vdash \Gamma, x : A \vdash x A : B, \Delta \\
\overrightarrow{\pi} P & \vdash \Gamma, x : A \vdash x A : B, \Delta
\end{align*}
\]
by the previous two parts, we can construct:

\[
\begin{array}{c}
\Gamma \vdash \alpha \vdash A, \Delta \\
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash \alpha \vdash A, \Delta \\
\end{array}
\]

Essentially in this section we have shown how we can achieve a compositional encoding of \( \land \) into \( \pi \) that preserves the types.

### 6. The Lambda Calculus

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

**Definition 6.1** (Type assignment for the \( \lambda \)-calculus).

\[
\begin{align*}
(\text{Ax}) : & \quad \Gamma, x : A \vdash \lambda x : B \vdash M : B \\
(\neg \text{I}) : & \quad \Gamma, x : A \vdash \lambda x : A \rightarrow B \\
(\rightarrow \text{E}) : & \quad \Gamma \vdash \lambda x : A \rightarrow B, \Delta \vdash N : A \\
\end{align*}
\]

The following was already defined in [8]:

**Definition 6.2** (Interpretation of the \( \lambda \)-calculus in \( \pi \)).

\[
\begin{align*}
\llbracket x \rrbracket_\alpha^{\pi} & \triangleq \langle x - \alpha \rangle \\
\llbracket \lambda x. M \rrbracket_\alpha^{\pi} & \triangleq \check{\llbracket \lambda x. M \rrbracket_\alpha^{\beta}\beta - \alpha} \\
\llbracket M N \rrbracket_\alpha^{\pi} & \triangleq \check{\llbracket M \rrbracket_\alpha^{\gamma}\gamma} \check{\llbracket N \rrbracket_\alpha^{\beta}\beta [x \gamma]} \check{\gamma(y - \alpha)} \check{\gamma, \beta, x, y, fresh}
\end{align*}
\]

Observe that every sub-net of \( \llbracket M \rrbracket_\alpha^{\pi} \) has exactly one free plug, and that this is precisely \( \alpha \). Moreover, notice that, in the \( \lambda \)-calculus, the output (i.e. result) is anonymous; where an operand ‘moves’ to carry a name via a variable, but where it comes from is not mentioned, since it is implicit. Since in \( \pi \), 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 \( \lambda \)-term \( M \) as well, carried in the sub-script of \( \llbracket M \rrbracket_\alpha^{\pi} \); this name \( \alpha \) 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 \( \lambda \) into \( \pi \) and \( \land \) into \( \pi \), we get yet another encoding of the \( \lambda \)-calculus into \( \pi \) [23, 22], one that preserves assignable simple types; as usual, the interpretation is parametric over a name.

**Definition 6.3** (Interpretation of the \( \lambda \)-calculus in \( \pi \) via \( \pi \)).

The mapping \( \llbracket \rrbracket^{\pi} : \Delta \rightarrow \pi \) is defined by:

\[
\llbracket \lambda M \rrbracket^{\pi} = \llbracket \lambda M \rrbracket^{\pi}_{\alpha^{\pi}}
\]

Since in [8] it is shown that the interpretation \( \llbracket \rrbracket^{\pi} \) preserves both reduction and types, the following result is immediate:

**Corollary 6.4** (Simulation of the Lambda Calculus).

1. If \( \Delta \vdash_{\pi} N \) then \( \llbracket \Delta \rrbracket^{\phi}_{\pi} \models \llbracket N \rrbracket^{\phi}_{\pi} \).
2. If \( \Gamma \vdash_{\alpha} M : A \), then \( \llbracket \Gamma \rrbracket^{\phi}_{\pi} \models \llbracket M \rrbracket^{\phi}_{\pi} \).

### 7. Expressing \( \neg, \& \), and \( \lor \) in \( \pi \)

In this section we will look at the other logical connectives: \( \neg, \& \), and \( \lor \). The sequent rules that correspond to these connectives are as follows:
DEFINITION 7.1.

\[\Gamma \vdash A, \Delta \]  
\[\Gamma, \neg A \vdash \Delta \]  
\[\forall \Gamma, A \vdash \Delta, \Gamma, B \vdash \Delta \implies \Gamma, A \lor B \vdash \Delta \]  
\[\forall \Gamma, A \vdash \Delta \implies \Gamma, B \vdash \Delta, \Gamma \vdash \Delta \]  
\[\forall \Gamma, \alpha \vdash \Delta \implies \Gamma, \beta \vdash \Delta \]  
\[\forall \Gamma, A \vdash \Delta \implies \Gamma, AB \vdash \Delta \]  
\[\forall \Gamma, \alpha \vdash \Delta \]  
\[\forall \Gamma, A \vdash \Delta \]  
\[\forall \Gamma, \alpha \vdash \Delta \]  

To extend the Curry-Howard isomorphism of $\mathcal{X}$ also to these connectors, we follow the same approach as used for the arrow: a disappearing formula in a context corresponds to a connector that gets bound, and a formula that appears in a context corresponds to a connector that is introduced.

DEFINITION 7.2. We extend $\mathcal{X}$'s syntax:

\[x \cdot P \tilde{\alpha} \quad \text{left inversion} \]
\[\tilde{\alpha} P \quad \text{right inversion} \]
\[x \cdot (\tilde{g}, o) P \quad \text{left projection} \]
\[x \cdot (o, \tilde{z}) P \quad \text{right projection} \]
\[(P \tilde{\alpha}, Q \tilde{\beta}) \cdot \gamma \quad \text{pair} \]
\[x \cdot (\tilde{g} P \tilde{\alpha} \mid Q \tilde{\beta}) \quad \text{choice} \]
\[P \tilde{\alpha} \cdot \gamma \quad \text{in-left} \]
\[P \tilde{\beta} \cdot \gamma \quad \text{in-right} \]

and add the type assignment rules:

\[\text{inv-l} : \quad P : \Gamma, x : A, \Delta \implies \Gamma, x : A \times \Delta \]
\[\text{inv-r} : \quad \tilde{P} \alpha : \Gamma \implies \Gamma, x : A, \Delta \]
\[\text{proj-l} : \quad x \cdot (\tilde{g}, o) P : \Gamma, x : A \times \Delta \]
\[\text{proj-r} : \quad x \cdot (o, \tilde{z}) P : \Gamma, x : A \times \Delta \]
\[\text{pair} : \quad (P \tilde{\alpha}, Q \tilde{\beta}) \cdot \gamma : \Gamma \times \Delta \]
\[\text{choice} : \quad x \cdot (\tilde{g} P \tilde{\alpha} \mid Q \tilde{\beta}) : \Gamma, x \times \Delta \]
\[\text{in-l} : \quad P \tilde{\alpha} \cdot \gamma : \Gamma \times \Delta \]
\[\text{in-r} : \quad P \tilde{\beta} \cdot \gamma : \Gamma \times \Delta \]

We can also introduce $z \cdot (\tilde{z}, \tilde{y}) P$ as abbreviation for the double projection $z \cdot (\tilde{z}, o) (z \cdot (\tilde{y}, \tilde{g}) P)$, and, similarly, $P \tilde{\alpha} (\tilde{z}, \tilde{y}) \cdot \gamma$ for $(P \tilde{\alpha} (\tilde{z}, \tilde{y}) \cdot \gamma) (\tilde{z}, \tilde{y})$, together with the type assignment rules

\[\text{proj} : \quad P : \Gamma, y : A, \Delta \implies \Gamma, z : B, \Delta \]
\[\text{in} : \quad P \tilde{\alpha} (\tilde{z}, \tilde{y}) \cdot \gamma : \Gamma \times \Delta \]

We note that reduction is extended naturally by adding the following reduction rules, which are the logical rules for the new logical connectors.

DEFINITION 7.3 (Additional logical rules).

We would also need to add the corresponding propagation rules (including, of course, propagation over import and export).

We will now study the logical equivalence $\neg A \to (A \land \neg B)$. As can be expected, we can mimic the export (which corresponds to right-introduction of the arrow) with

\[
\frac{\exists \tilde{P} \alpha : \Gamma, x : A \times \Delta}{\exists \tilde{P} \alpha : \Gamma, x : A \times \Delta} \quad (W, \alpha \text{ fresh})
\]

and, similarly,

\[
\frac{\exists \tilde{R} \beta : \Gamma \times \Delta}{\exists \tilde{R} \beta : \Gamma \times \Delta} \quad (W, \gamma \text{ fresh})
\]

Notice that, since $\alpha$ is fresh, we can reduce:

\[
\frac{(\exists \tilde{P} \alpha \cdot \beta) \tilde{\gamma} \cdot \gamma}{\exists \tilde{P} \alpha \cdot \beta} \quad (\exists \text{ imp})
\]

7.1 Encoding $\neg$ with $\&$, $\lor$.

We will now study the double equivalence $A \to B \iff (A \land \neg B)$. As can be expected, we can mimic the export (which corresponds to right-introduction of the arrow) with

\[
\frac{\forall \tilde{P} \alpha \cdot \beta : \Gamma, x : A \times \Delta}{\forall \tilde{P} \alpha \cdot \beta : \Gamma, x : A \times \Delta} \quad (W, \alpha \text{ fresh})
\]

and the import (the left-introduction of the arrow) with

\[
\frac{\neg \exists \tilde{R} \beta : \Gamma \times \Delta}{\neg \exists \tilde{R} \beta : \Gamma \times \Delta} \quad (W, \gamma \text{ fresh})
\]

We can also introduce $z \cdot (\tilde{z}, \tilde{y}) P$ as abbreviation for the double projection $z \cdot (\tilde{z}, o) (z \cdot (\tilde{y}, \tilde{g}) P)$, and, similarly, $P \tilde{\alpha} (\tilde{z}, \tilde{y}) \cdot \gamma$ for $(P \tilde{\alpha} (\tilde{z}, \tilde{y}) \cdot \gamma) (\tilde{z}, \tilde{y})$, together with the type assignment rules

\[\text{proj} : \quad P : \Gamma, y : A, \Delta \implies \Gamma, z : B, \Delta \]
\[\text{in} : \quad P \tilde{\alpha} (\tilde{z}, \tilde{y}) \cdot \gamma : \Gamma \times \Delta \]

This suggests an encoding of ‘implicational’ $\mathcal{X}$ into the ‘$\&$-$\lor$-variant’:

\[
\frac{\exists \tilde{P} \alpha \cdot \beta \cdot \gamma}{\exists \tilde{P} \alpha \cdot \beta \cdot \gamma} \quad (\text{proj})
\]
\[
\frac{\forall \tilde{Q} \beta \cdot \gamma \cdot \delta : \Gamma \times \Delta}{\forall \tilde{Q} \beta \cdot \gamma \cdot \delta : \Gamma \times \Delta} \quad (\text{in})
\]
As far as simulation is concerned, we can show:
\[
\begin{align*}
\llbracket (P; \bar{G}; \beta); \gamma \rrbracket \beta & \mapsto \bar{g}(Q; \gamma[y] \vDash B) = \\
\llbracket (z \cdot (\bar{x}, \bar{v})(v \cdot [P]\bar{a}) \cdot \beta); \gamma \rrbracket \beta & \mapsto \\
\bar{g}(y \cdot ([Q]\gamma, ([R]; \gamma) \cdot \gamma) \cdot \beta) & \mapsto \\
((([Q]\gamma, ([R]; \gamma) \cdot \gamma) \cdot \gamma); \beta) & \mapsto \\
\llbracket Q \rrbracket \gamma \mapsto \bar{z}(\llbracket R \rrbracket; \gamma) \gamma \mapsto \\
\llbracket P \rrbracket \alpha \mapsto \bar{g}(y \cdot [Q]\beta) & \mapsto \\
\llbracket P \rrbracket \alpha \mapsto \bar{g}\llbracket R \rrbracket & \mapsto \\
\llbracket P \rrbracket \alpha \mapsto \bar{g}(y \cdot [Q]\beta) & \mapsto \\
\llbracket Q \rrbracket \beta \mapsto \bar{g}(y \cdot [Q]\beta) \mapsto \\
\llbracket R \rrbracket & \mapsto \\
\llbracket P \rrbracket \alpha \mapsto \bar{g}(y \cdot [Q]\beta) \mapsto \\
\llbracket Q \rrbracket \beta \mapsto \bar{g}(y \cdot [Q]\beta) \mapsto \\
\llbracket R \rrbracket & \mapsto \\
\llbracket P \rrbracket \alpha \mapsto \bar{g}(y \cdot [Q]\beta) \mapsto \\
\llbracket Q \rrbracket \beta \mapsto \bar{g}(y \cdot [Q]\beta) \mapsto \\
\llbracket R \rrbracket & \mapsto \\
\llbracket P \rrbracket \alpha \mapsto \bar{g}(y \cdot [Q]\beta) \mapsto \\
\llbracket Q \rrbracket \beta \mapsto \bar{g}(y \cdot [Q]\beta) \mapsto \\
\llbracket R \rrbracket & \mapsto \\
\llbracket P \rrbracket \alpha \mapsto \bar{g}(y \cdot [Q]\beta) \mapsto \\
\llbracket Q \rrbracket \beta \mapsto \bar{g}(y \cdot [Q]\beta) \mapsto \\
\llbracket R \rrbracket & \mapsto \\
\llbracket P \rrbracket \alpha \mapsto \bar{g}(y \cdot [Q]\beta) \mapsto \\
\llbracket Q \rrbracket \beta \mapsto \bar{g}(y \cdot [Q]\beta) \mapsto \\
\llbracket R \rrbracket & \mapsto \\
\llbracket P \rrbracket \alpha \mapsto \bar{g}(y \cdot [Q]\beta) \mapsto \\
\llbracket Q \rrbracket \beta \mapsto \bar{g}(y \cdot [Q]\beta) \mapsto \\
\llbracket R \rrbracket & \mapsto \\
\llbracket P \rrbracket \alpha \mapsto \bar{g}(y \cdot [Q]\beta) \mapsto \\
\llbracket Q \rrbracket \beta \mapsto \bar{g}(y \cdot [Q]\beta) \mapsto \\
\llbracket R \rrbracket & \mapsto \\
\llbracket P \rrbracket \alpha \mapsto \bar{g}(y \cdot [Q]\beta) \mapsto \\
\llbracket Q \rrbracket \beta \mapsto \bar{g}(y \cdot [Q]\beta) \mapsto \\
\llbracket R \rrbracket & \mapsto \\
\llbracket P \rrbracket \alpha \mapsto \bar{g}(y \cdot [Q]\beta) \mapsto \\
\llbracket Q \rrbracket \beta \mapsto \bar{g}(y \cdot [Q]\beta) \mapsto \\
\llbracket R \rrbracket & \mapsto \\
\llbracket P \rrbracket \alpha \mapsto \bar{g}(y \cdot [Q]\beta) \mapsto \\
\llbracket Q \rrbracket \beta \mapsto \bar{g}(y \cdot [Q]\beta) \mapsto \\
\llbracket R \rrbracket & \mapsto \\
\llbracket P \rrbracket \alpha \mapsto \bar{g}(y \cdot [Q]\beta) \mapsto \\
\llbracket Q \rrbracket \beta \mapsto \bar{g}(y \cdot [Q]\beta) \mapsto \\
\llbracket R \rrbracket & \mapsto \\
\llbracket P \rrbracket \alpha \mapsto \bar{g}(y \cdot [Q]\beta) \mapsto \\
\llbracket Q \rrbracket \beta \mapsto \bar{g}(y \cdot [Q]\beta) \mapsto \\
\llbracket R \rrbracket & \mapsto \\
\llbracket P \rrbracket \alpha \mapsto \bar{g}(y \cdot [Q]\beta) \mapsto \\
\llbracket Q \rrbracket \beta \mapsto \bar{g}(y \cdot [Q]\beta) \mapsto \\
\llbracket R \rrbracket & \mapsto \\
\llbracket P \rrbracket \alpha \mapsto \bar{g}(y \cdot [Q]\beta) \mapsto \\
\llbracket Q \rrbracket \beta \mapsto \bar{g}(y \cdot [Q]\beta) \mapsto \\
\llbracket R \rrbracket & \mapsto \\
\llbracket P \rrbracket \alpha \mapsto \bar{g}(y \cdot [Q]\beta) \mapsto \\
\llbracket Q \rrbracket \beta \mapsto \bar{g}(y \cdot [Q]\beta) \mapsto \\
\llbracket R \rrbracket & \mapsto \\
\llbracket P \rrbracket \alpha \mapsto \bar{g}(y \cdot [Q]\beta) \mapsto \\
\llbracket Q \rrbracket \beta \mapsto \bar{g}(y \cdot [Q]\beta) \mapsto \\
\llbracket R \rrbracket & \mapsto \\
\llbracket P \rrbracket \alpha \mapsto \bar{g}(y \cdot [Q]\beta) \mapsto \\
\llbracket Q \rrbracket \beta \mapsto \bar{g}(y \cdot [Q]\beta) \mapsto \\
\llbracket R \rrbracket & \mapsto \\
\llbracket P \rrbracket \alpha \mapsto \bar{g}(y \cdot [Q]\beta) \mapsto \\
\llbracket Q \rrbracket \beta \mapsto \bar{g}(y \cdot [Q]\beta) \mapsto \\
\llbracket R \rrbracket & \mapsto \\
\llbracket P \rrbracket \alpha \mapsto \bar{g}(y \cdot [Q]\beta) \mapsto \\
\llbracket Q \rrbracket \beta \mapsto \bar{g}(y \cdot [Q]\beta) \mapsto \\
\llbracket R \rrbracket & \mapsto \\
\llbracket P \rrbracket \alpha \mapsto \bar{g}(y \cdot [Q]\beta) \mapsto \\
\llbracket Q \rrbracket \beta \mapsto \bar{g}(y \cdot [Q]\beta) \mapsto \\
\llbrack
linking the input $x$ of $P$ to the input $y$ of $Q$, an obviously undesirable reduction. However, the consistence check performed below states that well-typed processes will not cause this behaviour; the types prevent things going wrong.

We add the following type assignment rules for negation, conjunction and disjunction:

**Definition 8.4.**

\[
\begin{align*}
\text{(im-l)} : & \quad P : \Gamma, z : A \vdash \Delta \\
& x(z, s) : \\
\text{(im-r)} : & \quad \pi(x, s) : \\
& P : \Gamma, x : A \vdash \Delta \\
\text{(proj-l)} : & \quad x(y, n) : \\
& P : \Gamma, y : A \vdash \Delta \\
\text{(proj-r)} : & \quad x(n, z) : \\
& P : \Gamma, z : B \vdash \Delta \\
\text{(pair)} : & \quad \pi(a, b) : P \vdash \Delta \\
& \pi(a, b) : \Gamma \vdash c : A, B \vdash \Delta \\
\text{(choice)} : & \quad x(a, b) : \\
& P \vdash \Delta \\
\text{(in-l)} : & \quad \pi(a, n) : \\
& P \vdash \Delta \\
\text{(in-r)} : & \quad \pi(n, b) : \\
& P \vdash \Delta \\
\end{align*}
\]

We can now check that the encoding of the new constructs in $\mathcal{X}$ preserves assignable types as well.

**Theorem 8.5.** If $P : \Gamma \vdash \Delta$, then $\Pi P : \vdash \Gamma \vdash \Delta$.

**Proof.**

\[
\begin{align*}
(P \cdot \alpha) : & \quad \exists \xi \vdash \Delta \\
& \not\exists \alpha \vdash ; \Gamma, \alpha : A \vdash \Delta \\
& \exists \alpha \vdash ; \Gamma, z \vdash A, \alpha : A \vdash \Delta \\
& (\nu \alpha) (x(z, s), (P \mid \alpha = z)) : \Gamma, x : A \vdash \Delta \\
(x \cdot (\gamma P)) : & \quad \exists \xi \vdash ; \Gamma, x : A \vdash \Delta \\
& \not\exists \gamma \vdash ; \Gamma, x : A \vdash \Delta \\
& (\nu \xi) (x(z, s), (P \mid \xi = z)) : \Gamma, x : A \vdash \Delta \\
(x \cdot (\gamma P)) : & \quad \exists \xi \vdash ; \Gamma, x : A \vdash \Delta \\
& \not\exists \gamma \vdash ; \Gamma, x : A \vdash \Delta \\
& (\nu \xi) (x(z, s), (P \mid \xi = z)) : \Gamma, x : A \vdash \Delta \\
\end{align*}
\]

so our extended encoding respects the classical logic rules.

**9. Expressing all connectors in $\pi$**

We have seen that the pairing facility is sufficient to encode $\mathcal{X}$ into $\pi$, but the disadvantage to the approach above is that in the target language, pairing is used to model not just implication, but also negation, conjunction and disjunction. It would be more elegant, perhaps, to have separate syntactic representations for all logical connectors not just in $\mathcal{X}$, but in $\pi$ as well. This is easy to achieve by extending the syntax of names, as well as enriching the matching feature.

We will use $* \xi$ for a bound name that does not occur in the relevant process. Apart from pairing, we add inversion, sum and implication to the construction of names in order to be able to deal with arrow types, negation, conjunction and disjunction, effectively
using the grammar:

\[
\begin{align*}
& a, b, c, d ::= \bullet \mid x \mid \alpha \\
& p ::= a \mid \{a, b\} \mid a + b \mid a > b
\end{align*}
\]

and add four variants of let-construct to deal with matching:

\[
\begin{align*}
& \text{let } (x, y) = (m, n) \text{ in } R \equiv R[n/x, m/y] \\
& \text{let } x > y = n > m \text{ in } R \equiv R[n/x, m/y] \\
& \text{let } x + y = n + m \text{ in } R \equiv R[n/x, m/y] \\
& \text{let } x \cdot \hat{y} = n \text{ in } R \equiv R[n/x]
\end{align*}
\]

With this extension, our encoding now becomes:

**Definition 9.1.**

\[
\begin{align*}
& \Gamma[x; \alpha] = x(:, \alpha) \\
& \Gamma\{g; \beta\} = (\alpha, \beta) (b; \Gamma\{y; \beta\}) \\
& \Gamma[b; \alpha, \beta] = z(d; c, \alpha) (b; \Gamma\{y, \beta\}) \\
& \Gamma(P; \alpha) = (\alpha) (P; \Gamma\{x, \alpha\}) \\
& \Gamma(P; \alpha) \not\equiv \Gamma(Q; \beta) = (\alpha) (P; \Gamma\{x, \alpha\}) (\beta) (P; \Gamma\{x, \alpha\}) (Q) \\
& \Gamma[P; \alpha, \beta] = (\alpha) (P; \Gamma\{x, \alpha\}) (\beta) (P; \Gamma\{x, \alpha\}) (Q) \\
& \Gamma[x; \alpha, \beta] = x(a + b); (\alpha) (a = y; P) + (\beta) (b = z; Q)
\end{align*}
\]

The type assignment rules in Definition 8.4 would have to change appropriately.

**Conclusion**

We studied how to give the computational meaning to classical proofs via the \(\pi\)-calculus. Our results have been achieved in two steps: (1) we have encoded \(\lambda\) into \(\pi\) enriched with pairing and showed that the encoding preserves interesting semantics properties; (2) we have defined a novel and ‘unusual’ type system for \(\pi\) 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 \(\pi\). Essentially we have interpreted the input in \(\pi\) as ‘witnesses’ for the left-hand side of the turnstule in \(\lambda\), and outputs as ‘witnesses’ for the right-hand side. Arrow-right in \(\lambda\) 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 interpreted as a forwarder that connects via private channels different inputs and outputs, that have the same type.

The work that naturally compares with ours is [20], where the encoding of \(\lambda\)-terms is presented. In that paper, full abstraction is proved, but only for natural deduction rather than for the sequent calculus as treated in this paper. Moreover, only the implicative fragment has been encoded. \(\lambda\) is a calculus without application and substitution that is much easier to interpret in \(\pi\). 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 \(\lambda\) (with explicit weakening and contraction) [21], and relate that with both game semantics and \(\pi\) without replication.

**References**