An exact correspondence between a typed
π-calculus and polarised proof-nets

Kohei Honda
Department of Computer Science
Queen Mary, University of London

Olivier Laurent*
Preuves Programmes Systèmes
CNRS – Université Paris 7

September 30, 2009

Abstract
This paper presents an exact correspondence in typing and dynamics be-
tween polarised linear logic and a typed π-calculus based on IO-typing.
The respective incremental constraints, one on geometric structures of
proof-nets and one based on types, precisely correspond to each other,
leading to the exact correspondence of the respective formalisms as they
appear in [Lau03] (for proof-nets) and [HYB04] (for the π-calculus).

1 Introduction
This paper presents exact mutual embeddings between the proof-nets and the π-
calculus. More precisely, we show a specific form of polarised linear logic [Lau03]
and a typed version of the asynchronous π-calculus [HYB04] are essentially
different ways of presenting the same structure. On the one hand, the π-calculus
is a formalism which can represent a wide variety of computational phenomena
starting from sequential programming languages to distributed computation
through a simple operation, channel passing, originating in CCS; on the other,
proof-nets represent dynamics of cut elimination in Linear Logic, which is a
proof-theoretic reformulation of intuitionistic logic. These two formalisms come
from quite different backgrounds: this paper pinpoints an exact way in which
we can relate these two different formalisms.

As is well-known, the so-called Curry-Howard correspondence allows us to
relate proofs of intuitionistic logic to the typed λ-calculus, so that a proof cor-
responds to a typed λ-term, the assumptions and conclusion of a proof to the

*Partially funded by the French ANR projet blanc “Curry Howard pour la Concurrence”
CHOCO ANR-07-BLAN-0324.
types of that term, and cut-elimination (a mechanical procedure to simplify proofs preserving the same assumptions and conclusions) to reduction in the $\lambda$-term. Linear logic [Gir87] can fully embed this isomorphism: further the logic enjoys duality of the classical logic, and the notion of cut-elimination in proof-nets is more fine-grained than the cut elimination in the intuitionistic logic. The connectives of Linear Logic arise as decomposition of the semantic universe underlying intuitionistic logic. The study of Linear Logic and proof-nets is usually motivated by logical or logically oriented semantic concerns, though some of the ideas in Linear Logic (such as linearity) are widely used in computational formalisms other than logically motivated ones.

In contrast, the $\pi$-calculus [MPW92] is a relatively recent development in process algebras, syntactic formalisms for understanding concurrent computation which include, among others, CCS [Mil89], CSP [Hoa85] and ACP [BW90]. In particular, all algebraic operators in the $\pi$-calculus are those of CCS: the $\pi$-calculus merely adds a syntactic treatment of name passing to CCS. The motivation to add name passing in the $\pi$-calculus is practical: the authors of [MPW92] wish to model so-called mobile systems, i.e., communicating systems in which connectivity among processes dynamically changes. Such systems are commonplace nowadays, be they mobile phone networks or email communication where email addresses themselves are exchanged. The study of the $\pi$-calculus in particular and process algebras in general is motivated by modelling practical systems rather than logical concern, even though it is also used as a tool to study semantics and logics of computation.

The present work shows that, in spite of these two very different origins and orientations, there is an expressive common part shared by these two formalisms. This “common part” is obtained through a natural restriction of each formalism: proof-nets are restricted so that they are “polarised” [Lau03], i.e. we have distinction between positive and negative formulae. For the $\pi$-calculus we use, other than asynchrony in communication, a restriction by types used for studying control operators [HYB04]. As we shall show, the correspondence between two fragments is as exact as can be: two translations are mutually inverse (up to the structural equality of the $\pi$-calculus); the well-formed proof-nets are precisely well-typed processes via translation and vice versa; and one-step reduction in one formalism is precisely mirrored by one-step reduction in another formalism.

The result poses many questions: why one of the (arguably) most advanced tools to study dynamics of logics and one of the (arguably) most advanced tools to study dynamics of concurrent computation coincide so closely? What merits does this correspondence give us? For the former the authors do not have an answer (another related mathematical tool for computation is game semantics [AJM00, HO00], which again has a different origin and motivation but has a close tie with both formalisms). For the second question, it may be worth recording a couple of observations we have gained through the present inquiry. First, our correspondence results treat the proof-nets which are semantically non-deterministic (in the $\pi$-calculus notation we allow typed processes of the form $!x.0|!x.0$ which introduce racing at $x$). While non-deterministic proof-nets
have been studied before ([Mau03] for example), the presented variant would be
of interest because of its precise correspondence with the $\pi$-calculus: the whole
laws and theories of non-deterministic processes from process algebras can now
be applied (for example we can now discuss the notion of interactive behaviour
of proof-nets directly using transition relations of the $\pi$-calculus). Second, the
syntactic constructs in the $\pi$-calculus are given an exact geometric presentation
through the corresponding proof-nets. While many graphical formalisms for the
$\pi$-calculi have been studied [BM05, JM04, LPV01, Maz05], our work differs in
that the target graphical formalism is directly based on proof-nets. This allows
us to compare incremental geometric constraints in proof-nets on the one hand
and incremental type-based constraints in the $\pi$-calculus. For emphasising this
point, our presentation starts from the correspondence result for the respective
non-deterministic extensions and adds incremental constraints: for each
constraint we again establish one-to-one correspondence between them, finally
reaching the deterministic, terminating formalisms studied in [Lau03, HYB04].
Thus already the both-way interactions enrich these two theories. We hope that
the present inquiry serves as a starting point of further dialogues between these
two different fields of study.

The connection between process algebras and Linear Logic was first observed
by Samson Abramsky in his computational interpretation of linear logic [Abr93,
Abr94]. The process formalism he used is not a standard process algebra but
a syntactic construction which directly represents proofs of Linear Logic as
terms. Bellin and Scott [BS94] have shown an embedding of a version of proof-
nets with a notion of polarities to a “synchronous” $\pi$-calculus, a non-standard
version of the $\pi$-calculus with commutativity of prefixes. Laneve, Parrow and
Victor [LPV01] showed that some parts of dynamics of the original proof-nets
can be captured by the fusion calculus, which is a variant of the $\pi$-calculus with
(what corresponds to) axiom links. Beffara [Bef06] has shown a realizability
semantics of linear logic which uses a synchronous version of the $\pi$-calculus with
a dynamic link. The present work differs from these works (and also from [BM05,
FM05, CF05]) in that we focus on significant, and independently conceived,
fragments of the original formalisms, and that there is an exact correspondence
in term/graph constructions and dynamics.

In the remainder, we first outline a fragment of proof-nets we shall use
(which is a non-deterministic extension of polarised proof-nets in [Lau03]). Then
we outline a fragment of the $\pi$-calculus we shall use (which is a relaxing of
type constraints in the typed $\pi$-calculus in [HYB04]). We then relate these
two formalisms. In the next section we discuss how incremental restrictions in
these two formalisms lead to more restricted classes of processes/proofs. This
is the core of our correspondence where we finally reach a precise coincidence
between polarised proof-nets [Lau03] and the control $\pi$-calculus [HYB04]. The
final section is dedicated to extensions of the correspondence between the two
formalisms.
2 Proof-nets

The notion of proof-nets we consider here comes from polarised proof-nets [Lau03, Lau05] with two important extensions: \(n\)-ary \(!\)-nodes (responsible for non-determinism) and named cuts. In addition, it is presented in the so-called “focalised” form [Gir01] (using \(n\)-ary multiplicative nodes, \(\otimes\) and \(\multimap\)); we omit axiom nodes; and we only consider exponential cuts.

The \(n\)-ary \(!\)-nodes correspond to a particular use of co-contraction nodes as defined in differential interaction nets [ER06]. An \(n\)-ary \(!\)-node can be implemented as \(n\) usual \(!\)-nodes followed by a \(n\)-ary co-contraction node (or \(n-1\) binary co-contraction nodes).

Formulae. We consider positive and negative formulae given by the following grammar. Let \(n \geq 0\) below.

\[
\begin{align*}
P &::= \bigotimes_{1 \leq i \leq n} ! N_i \\
N &::= \bigvee_{1 \leq i \leq n} ? P_i
\end{align*}
\]

The case of a 0-ary \(\otimes\) (resp. \(\multimap\)) is usually denoted by 1 (resp. \(\bot\)).

A formula is **positive** if it is of the form \(P\), **negative** if it is in the form \(N\).

The orthogonal of a formula, written \(P^\perp\) or \(N^\perp\), is obtained by exchanging \(\otimes\) and \(\multimap\), \(!\) and \(?\). Note taking the orthogonal of the orthogonal gives us the original formula.

Proof-nets. The following are proof-nets for polarised linear logic except we add slightly stronger constraints in the shape of formulae based on \(n\)-ary tensors/paras (i.e. we use the focalised form [Gir01]), using, in addition to the formulae given above, those of the forms \(!N\) (which is different from a unary tensor \(\bigotimes !N\)) and \(?P\) (which is different from a unary par). Moreover \(!\)-nodes are extended to the \(n\)-ary \((n > 0)\) case.

We use the following four kinds of nodes: \(\otimes\)-nodes, \(\multimap\)-nodes, \(?\)-nodes, \(!\)-nodes given by:

\[
\begin{align*}
\frac{! N_1 \ldots ! N_n}{\bigotimes_{1 \leq i \leq n} ! N_i} n \geq 0 \\
\frac{? P_1 \ldots ? P_n}{\bigvee_{1 \leq i \leq n} ? P_i} n \geq 0
\end{align*}
\]

\[
\frac{P \ldots P}{? P} n \geq 0 \text{ premises}
\]

\[
\frac{N \ldots N}{! N} n > 0 \text{ premises}
\]

with the corresponding formulae labelling edges.

Each node given above has the shape of a deduction rule: thus we use the standard terminology such as premise and conclusion of a node. Note we demand identical formulae as the premises of the third (resp. fourth) rule.

We construct a graph from these nodes together with labelled directed edges so that: (1) there is at most one outgoing edge from each node (conceived as going out from the conclusion of the source node); (2) if there is an edge from one node \(a\) to another node \(b\), then the conclusion of \(a\) should occur as one of
the premises of \( b \) (an edge is always directed from the conclusion of a node to a premise of another); and (3) if a node \( a \) has \( n \)-premises, then there are always \( n \)-incoming edges to \( a \), which we label \( 1, 2, \ldots, n \), usually left implicit, such that the conclusion of the source node is covered by the \( i \)-th premise of the source node through the \( i \)-th edge, i.e. the premise of the target should be identical with the conclusion of the source as a formula (thus a premise of the target is justified by exactly one conclusion of the source). However we identify proof-nets up to permutation of the incoming edges of each of the \( ? \)-nodes and \( ! \)-nodes.

By (3), each edge is associated with a formula, occurring in its source as the conclusion and in its target as one of the premises (the \( i \)-th premise if the edge is labelled \( i \)). An edge is called positive (resp. negative) if the associated formula is positive or is of the form \( \forall N \) (resp. negative or is of the form \( \exists P \)).

Note there can be no circular sequence of edges (since formulae are inductively given). If there is an edge from a node \( a \) to a node \( b \), we say \( a \) is above \( b \), or symmetrically \( b \) is below \( a \).

A node without outgoing edge is called a conclusion of the whole proof-net.

A proof-net (without cuts) is a directed (and possibly disconnected) graph of this kind satisfying:

- Each \( ? \)-node has zero or more \( \otimes \)-nodes of the same conclusion above it.
  Each \( ! \)-node has one or more \( \exists \)-nodes of the same conclusion above it.

- Each \( \otimes \)-node has zero or more \( ! \)-nodes above it. Dually each \( \exists \)-node has zero or more \( ? \)-nodes above it.

- With each premise \( a \) of each \( ! \)-node, we associate a box, that is a sub-proof-net with a conclusion \( a \) (called the main conclusion of the box), as well as zero or more \( \otimes \)-conclusions (called the auxiliary conclusions of the box), each of which must be above a \( ? \)-node (there are no other conclusions).

- Two boxes never overlap (they can be nested if they are associated with different \( ! \)-nodes).

A node is at depth \( n \) if there are \( n \)-boxes enclosing it. Note a node which does not occur in any box is at depth 0.

**Cuts.** We define cuts in a proof-net in a not completely usual, but equivalent, way following the idea in \cite{Gir01} that a cut corresponds to two dual edges (i.e. with orthogonal types) living at the same address (i.e. having the same name).

A cut is a pair of dual conclusions that occur in the same boxes (so in particular they are at the same depth), with one being the conclusion of a \( ? \)-node which is not a premise of another node and another being the conclusion of a \( ! \)-node (which may or may not be a premise of another node).

A proof-net with cuts is given by a proof-net with zero or more cuts such that these cuts satisfy the disjointness condition, in the sense that two distinct cuts never share a node. This disjointness condition does not preclude non-deterministic dynamics as we shall discuss when we introduce cut eliminations.
Figure 1: Examples of proof-nets
Example 1. Three examples of proof-nets are given in Figure 1, where dashed lines represent cuts. The top-left proof-net has one box with the unique ?-node above the !-node. It also has two ⊗-nodes sharing the ?-conclusion. The top-right proof-net has two (nested) boxes, whose outermost box has one auxiliary conclusion on the right and its main conclusion on the left. The conclusion of each !-node is cut with a ?-node without premises (the conclusion formula is implicit by duality). Finally the proof-net at the bottom has four boxes. The left-most two boxes share the same conclusion. None of these boxes contain auxiliary conclusions. In Section 4 later (Example 4), we shall present how these proof-nets can be translated to the equivalent π-terms.

Identification of nodes into a single node. We use the following operation on proof-nets. Consider two ?-nodes, say $w_1$ and $w_2$, with the same conclusion and without outgoing edges (i.e. their conclusions are not used for justifying the premises). Then the identification of $w_1$ and $w_2$ is done by replacing $w_1$ and $w_2$ by a single ?-node, say $w$, in such a way that each premise of $w_1$ and $w_2$ is now a premise of $w$. $w_1$ and $w_2$ disappear and their conclusions are now replaced by one (we do not have to worry about nodes below the original conclusions since, by our assumption, they are not used as premises of other nodes). The identification of two !-nodes in a proof net is given in the same way.

Cut elimination. We consider a cut between a ?-node say $w$, having a ⊗-node above it, and a !-node say $o$, all of them being at depth 0. Cut elimination eliminates this cut, and possibly generates new nodes. This corresponds to the $\tau$-action (reduction) in the π-calculus. This is done as follows.

First, recall a !-node has one or more ?-nodes above it, hence the corresponding boxes. We choose any one of these ?-nodes, say $p$, and the box $b$ above $p$, and make a single copy of its content. When copying, if an auxiliary conclusion (i.e. a ⊗-conclusion) of the box justifies a premise of a ?-node through an edge, then we add the corresponding premise to the ?-node, so that there is an edge from the conclusion of the copied ⊗-node to this newly added premise (note this increases the number of premises of this ?-node). Dually we choose one of the ⊗-nodes at depth 0 above $w$, say $t$.

Second, we remove the selected ?-node (which was originally $p$) from the copy of $b$. This leaves $k \geq 0$ ?-conclusions in the copy. Dually we erase the ⊗-node $t$, leaving $k$ !-nodes. Note their numbers are equal because a cut can only be placed between dual formulae.

Third and finally, we put a cut between the conclusion of each of these $k$ ?-nodes and the corresponding !-node one by one, following their labels. However if two such associated nodes $o'$ (the !-node) and $w'$ (the ?-node) are such that $o'$ has already been cut with a ?-node $w''$, we perform the identification of $w'$ and $w''$ in the sense defined above, instead of adding a new cut (which would have violated the disjointness condition for cuts, cf. page 5).

Anticipating the correspondence result in Section 4, here is an informal illustration of how these three steps of cut elimination above are related to ideas
in the \( \pi \)-calculus. The first step above corresponds to a non-deterministic selection of one of the redexes in the \( \pi \)-term, by choosing an input (a \( \exists \)-node above the \(!\)-node) and an output (a \( \otimes \)-node above the \( ? \)-node), followed by making a copy of the body of the replicated input in that redex. The second and third steps graphically carry out the standard (internal) name passing interaction (a similar decomposition is discussed in [Mil92b]). In the second step, the initial part of name passing is performed, annihilating a pair of the initial subjects of the prefixed process (a subject is the initial name occurrence in a prefix, e.g. the initial \( x \) in \( x(y).P \)). The third step then carries out the rest of the interaction, i.e. value passing, which is bound (private) name passing in the present context. The names emitted and the names received are simultaneously identified and hidden, forming new pairs of potential redexes, signified by new cuts.

**Example 2.** Using the proof-net at the bottom in Figure 1, the three steps of cut elimination are shown in Figure 2. The first picture is after the first step, copying the content of the right-most box. Choosing a \( \exists \)-node and a \( \otimes \)-node is trivial (since each is unique). Further, because there is no auxiliary conclusion, we need no additional edges to newly made premises. In the second picture, we simply eliminate the pair of \( \otimes \) and \( \otimes \) nodes. The third picture gives the final result, by identifying the two \( ? \)-nodes on the left with their counterparts on the copied graph (since the conclusions of the \( ? \)-nodes have already been cut).

In addition to this standard cut elimination, we also consider an extended version of cut elimination, which adds the following two cases. First, we consider the case when a premise (\( \otimes \)-node) of the \( ? \)-node \( w \) is inside a larger box \( b' \) (so that it is not of depth 0) but not in the selected box \( b \) of the \( ! \)-node \( o \), in which case the copy of the content of \( b \) is moved inside this enclosing box \( b' \) and we proceed as for the standard case. Second, we reduce a cut between a 0-ary \( ? \)-node and a conclusion \( ! \)-node (i.e. not above another node) by erasing the \( ! \)-node and its boxes and the associated premises of \( ? \)-nodes. We call extended cut elimination the result of adding these last two cases.

**Complete proof-nets.** Since proof-nets have a slightly finer presentation of constructions than the \( \pi \)-calculus, we restrict the shape of proof-nets by a completeness constraint, as well as adding names as labels on conclusions of proof-nets (and on no other node) to get an explicit matching with channel names.

The following conditions are required for *complete proof-nets*:

- Each \( \otimes \)-node is above a \( ? \)-node. Each \( \exists \)-node is above a \( ! \)-node.
- The conclusion of each \( ! \)-node must be cut with the conclusion of a \( ? \)-node.
- Some conclusions of the proof net are labelled with names according to the following rules:
  - If its node is a \( ? \)-node then it must have a label if and only if it does not belong to a cut.
Figure 2: Cut elimination procedure
– If its node is a !-node then it must have a label if it does not belong to a cut. If it does, it may or may not have a label.

Two distinct conclusion nodes are never labelled with the same name. We write $x, y, \ldots$ for names used to label conclusions.

If a cut contains the conclusion of a !-node which is either labelled by a name or a premise of another node, the cut is called named (and unnamed otherwise). We restrict the second case of extended reduction, which we may call erasing reduction, to unnamed cuts (because a named cut may add more $\otimes/\forall$ nodes above $?!/!$ nodes, making non-trivial cut elimination possible: the erasing reduction is intended to eliminate the cut which can never induce interaction).

A node is free if its conclusion is named (hence un-connected to any premise). In this case we also say this conclusion is free. In a complete proof-net, we will only consider free conclusions as the conclusions of the proof-net. In particular a !- or ?-node which is not a premise of any node and which has an unnamed conclusion (so that it must belong to a cut) is now allowed to occur inside a box, which we do not consider as an auxiliary conclusion of the box. This extends the previous definition of box without cut (page 5), but we stick to the fact that auxiliary conclusions of a box are conclusions of $\otimes$-nodes.

There is a simple completion procedure turning any proof-net into a complete one. Let $R$ be a proof-net without names labelling the conclusions (otherwise we can just remove them), first we complete $R$ so that it has neither free $\otimes$-nodes nor free $\forall$-nodes, as follows: If there is a free $\otimes$-node with conclusion $P$ then add a new $?-$node just below it which is from $P$ to $?P$. If there is a free $\forall$-node, simply erase this $\forall$-node. For the conclusion of each !-node which does not belong to a cut, introduce a new $?$-node (with 0 premise) and add a cut between the conclusion of the !-node and the conclusion of this new node. Finally, we add pairwise distinct names to all the conclusions of the resulting proof-net which are not connected to premises, except that we do not name the conclusions of ?-nodes belonging to cuts.

Example 3. Each proof-net in Figure 1 is complete.

Computational intuition behind completion. For those familiar with polarised proof-nets, the completion is a simple procedure to turn visible interface of proof-nets to specific shape (! and ?).

For readers from the process algebra background, we supply some computational intuition. First, a $\otimes$ node (resp. a $\forall$ node) indicates abstraction of names for input (resp. $\nu$-abstraction for bound output), but without the initial subject, so that such a node with $n$ premises corresponds to a $n$-adic communication action — receiving or sending $n$-names. Thus one may consider them as indicating the $(y_{1},\ldots,y_{n})$-part of $!x(y_{1},\ldots,y_{n}),P$ and $\overline{\pi}(y_{1},\ldots,y_{n})P$ for $\forall$ and $\otimes$, respectively. A $?$-node (resp. !-node) represents a channel used for the subject of prefixes ($x$ in $!x(y_{1},\ldots,y_{n}),P$ and $\bar{x}(y_{1},\ldots,y_{n})P$) and, simultaneously, the sharing of this channel by the prefixes: that is, if a $?$-node has $m$ premises, this corresponds to $m$ output
processes sharing the same channel (if \( m = 0 \), then this means there is no process sharing that channel). Thus the completion procedure starts from a graph representing, say, the union of \((y_1...y_n).P \) and \((z_1...z_n)Q\) and reaches the process \( P|\nu(z_1...z_n)Q\). A box denotes a synchronisation boundary, i.e. \((y_1...y_n).P\) of \( P|!y_1...y_n).P\) is in the box (note a box exists only for \(!\), so the correspondence is exact). Finally cuts are when two processes can potentially interact by one having an input and another an output on a shared channel. Unnamed cuts in addition add hiding of that shared channel.

3 Typed \( \pi \)-calculus

We consider a polyadic \( \pi \)-calculus with the following restrictions: communication is private (or internal) [MPW92, San95] and asynchronous [HT91, Bou92] and moreover all and only inputs are replicated. The typing system will enforce an additional locality constraint (channels received by an input are only used for output). From a different viewpoint this calculus is a straightforward non-deterministic and non-terminating extension of the calculus in [HYB04] used for embedding \( \lambda \mu \)-calculus fully abstractly.

Terms. Given a set of names denoted \( x, y, \ldots \), terms are given by:

\[
P : = \ !x(y).P \mid \bar{x}(y)P \mid P|Q \mid \nu xP \mid 0.
\]

\( !x(y).P \) is called a replicated receptor, or input; \( \bar{x}(y)P \) is an output; \( P|Q \) is a parallel composition; \( \nu xP \) is hiding of \( x \); \( 0 \) is the inaction.

In input and output constructions, the \( y \) in \( !x(y).P \) and \( \bar{x}(y)P \) are bound as for \( x \) in \( \nu xP \).

We may call the output construction \( \bar{x}(y)P \) we use, which corresponds to \( \nu y(\bar{x}(y)|P) \) in the usual \( \pi \)-calculus, private output.

Types and judgements. Terms are typed with input/output types given as follows.

\[
\tau_I := (\tau_0^I)^I \\
\tau_0 := (\tau_1^I)^I \\
\tau := \tau_I \mid \tau_0
\]

The mode of a type is defined by \( \text{md}(\tau_I) = I \) and \( \text{md}(\tau_0) = 0 \).

The dual \( \overline{\tau} \) of a type \( \tau \) is obtained by exchanging \( \tau_I \) and \( \tau_0 \), \( ! \) and \( ? \).

Intuitively, a name can have both a type and its dual type, since a single name is shared by both inputs and outputs. In typing judgements, we use the output form of the type if we are sure that no input occurs on that name, otherwise using the input form of the type.

A context \( A \) is a finite function from names to types. If \( x \) is not in the domain of \( A \), we write \( A, x : \tau \) for the context which extends \( A \) with \( x \mapsto \tau \).
(otherwise $A, x : \tau$ is not defined). According to the previous remark, when adding a declaration to a context we use the following partial operation $\circ$:

- $A \circ x : \tau = A, x : \tau$ if $x \notin A$
- $A \circ x : \tau$ is not defined if $x : \tau' \in A$ with $\tau' \neq \tau$ and $\tau' \neq \tau$.
- $A \circ x : \tau_0 = A$ if $x : \tau_0 \in A$ or $x : \tau_0 \in A$
- $A \circ x : \tau_1 = A', x : \tau_1$ if $x : \tau_1 \in A$ and $A' = A \setminus x$
- $A \circ x : \tau_1 = A$ if $x : \tau_1 \in A$

We extend $\circ$ to a partial operation on contexts as follows: given $A_1 = x_1 : \tau_1, \ldots, x_n : \tau_n$ and $A_2 = y_1 : \tau'_1, \ldots, y_k : \tau'_k$, $A_1 \circ A_2$ is given as $A_1 \circ A_2 = (((A_1 \circ y_1 : \tau'_1) \cdots) \circ y_k : \tau'_k)$. $A_1 \circ A_2$ is defined iff all of the involved $\circ$-compositions are defined, with the same resulting value: otherwise it is undefined.

**Lemma 1** (operator $\circ$). Define the partial order $\leq$ on contexts generated from the set inclusion and $A, x : \tau_0 \leq A, x : \tau_0$ for each $\tau_0$ (cf. [Hon96]). Then if $A_1 \circ A_2$ is defined then it gives the join of $A_1$ and $A_2$ with respect to $\leq$.

**Proof.** Immediate from the definition.

Intuitively, the ordering $\leq$ represents the degree of composability: the smaller a type is in this ordering, the easier that type is composable with others (i.e. $\circ$ is more defined). By Lemma 1, $\circ$ defines the (partial) join under the ordering $\leq$, hence as an operation $\circ$ is partially commutative and partially associative (i.e. definedness and, if defined, the resulting values coincide between $A \circ B$ and $B \circ A$, similarly $(A \circ B) \circ C$ and $A \circ (B \circ C)$), with the identity the empty context. In particular, assuming $x$ is not in $\text{dom}(A_1) \cup \text{dom}(A_2)$, $A_1 \circ (A_2, x : \tau)$ and $(A_1 \circ A_2), x : \tau$ coincide in their definedness and their value.

A typing judgement is of the shape $\vdash P \triangleright A$. We present the typing rules.

**Typing rules.** The following typing rules relax some of the constraints of those from [BHY01]. Below $\emptyset$ is the empty context.

\[
\begin{align*}
\vdash \emptyset &\vdash 0 \triangleright 0 \\
\vdash P \triangleright A, y : \tau_0 &\vdash P_1 \triangleright A_1 \\
\vdash P \triangleright A, y : \tau_0 &\vdash P_2 \triangleright A_2 \\
\vdash P \triangleright A, x : \tau_1 &\vdash P \triangleright A \\
\vdash P \triangleright A, x : \tau_1 &\vdash \nu x P \triangleright A \\
\vdash P \triangleright A, y : \tau_1 \vdash [\text{in}]_\text{md}(A) = 0 \\
\vdash P \triangleright A, x : \tau_1 \vdash [\text{out}]_\text{md}(A) = 0 \\
\vdash P \triangleright A \vdash [\text{wk}]_x \not\in A \\
\end{align*}
\]

where $\text{md}(A) = 0$ means that all the types appearing in $A$ are of mode 0. In each of (in), (out) and ([]), we stipulate that the deduction is possible only when $\circ$ in the conclusion is defined. In (wk), we assume $x$ does not occur in $A$. Below and henceforth we always assume the standard bound name convention, i.e. binding names are disjoint from free names.
Lemma 2 (permutation of wk). The application of (wk) can always be permuted up (hence with the same height of the derivation tree) except when the previous rule is (0).

Proof. By rule induction and because, by Lemma 1, a disjoint union commutes with other composition by $\odot$.

The typing is closed under injective renaming as is immediate from the shape of each typing rule (formally by a completely trivial rule induction on typing rules). We shall also use later the following result.

Lemma 3. If $\vdash P \bowtie A, x : \tau_0, y : \tau_0$ then $\vdash P[x/y] \bowtie A, x : \tau_1, y : \tau_1$.

Proof. Both statements are proved by induction on typing rules. For the first statement the only non-trivial case is when the newly added subject of the output prefix (the subject is the initial name of a prefix) in (out) is coalesced, which is immediate by Lemma 1 which says that $\tau_0$ is idempotent with respect to $\odot$. For the second statement the only non-trivial case is when we coalesce the newly added replicated name in (in), which is again direct from Lemma 1 noting the input type is bigger than its dual.

The typing rules also satisfy the standard properties of weakening (by a disjoint context only including output types in its range) and thinning (in the sense that the part of a context whose domain is disjoint from free names of a process can be cut off — note this can only concern names with output types).

Structural Congruence. The $\alpha$-equivalence of terms is defined by the usual renaming of bound variables, avoiding unwanted capture of names.

The structural congruence relation on terms is the smallest congruence generated by the $\alpha$-equivalence and the following equations:

\begin{align*}
0|P & \equiv P & (1) \\
P|Q & \equiv Q|P & (2) \\
P|(Q|R) & \equiv (P|Q)|R & (3) \\
\nu x \nu y P & \equiv \nu y \nu x P & (4) \\
\nu x (P|Q) & \equiv (\nu x P)|Q & \text{if } x \notin Q & (5) \\
x(\bar{u})\bar{y}(\bar{v})P & \equiv \bar{y}(\bar{v})\bar{x}(\bar{u})P & \text{if } x \notin \bar{u} \text{ and } y \notin \bar{u} \text{ and } \bar{u} \cap \bar{v} = \emptyset & (6) \\
x(\bar{u})(P|Q) & \equiv (x(\bar{u})P)|Q & \text{if } \bar{u} \notin Q & (7) \\
\nu y x(\bar{u})P & \equiv x(\bar{u})\nu y P & \text{if } y \notin \{x, \bar{u}\} & (8) \\
\end{align*}

Up to the equivalence relation $\equiv$, $\pi$-terms can be rewritten into canonical forms:

\[ P \equiv \nu x \bar{y} \prod_{i \in I} [v_i(\bar{w}_i), P_i] \]
For the proof, first externalise all \( \nu \)-bound channels which are not under input prefixes, then do the same for all output prefixes. Since there are only a finite number of output prefixes the remaining processes are input processes with no \( \nu \)-binding.

The typing system is well behaved with respect to this structural congruence relation:

**Lemma 4.** If \( \vdash P > A \) and \( P \equiv Q \) then \( \vdash Q > A \).

**Proof.** By Lemma 2 we safely assume, except when the process is 0, the last rule follows the uppermost constructor of the process. The initial equation (identity of 0 for \( \vert \) ) is by weakening. The next two (commutativity and associativity of \( \land \) ) is by partial commutativity and associativity of \( \circ \) from Lemma 1. \( \nu y x P \) and \( \nu y x P' \) have the same contexts since \( \nu \) just takes off a singleton from a context. The next rule, scope extrusion, is immediate from idempotence of an output type (again by Lemma 1). The next rule is again by commutativity of \( \circ \) by Lemma 1. The second rule to the last, the second scope extrusion rule, is as the first scope extrusion rule. Finally the last rule is immediate from inspection of the two involved rules. \( \square \)

**Reduction.** The dynamics of the calculus is defined by a reduction relation between terms.

\[
\begin{align*}
P(x(y))P | x(y)Q & \rightarrow x(y)P | \nu \bar{x}(y)Q \\
P | Q & \rightarrow P'|Q' \\
\nu P & \rightarrow \nu P' \\
\bar{x}(y)P & \rightarrow \bar{x}(y)Q
\end{align*}
\]

The extended reduction is obtained by replacing the first reduction rule above with the following two:

\[
\begin{align*}
\lambda x(y)P | C[\bar{x}(y)Q] & \rightarrow \lambda x(y)P | \nu \bar{x}(y)Q \\
\nu x! x(y)P & \rightarrow 0
\end{align*}
\]

where \( C[\ ] \) is an arbitrary context not binding \( x \). Note \( \rightarrow \subset \rightarrow \). We have:

**Lemma 5.** If \( \vdash P > A \) and \( P \not\rightarrow Q \) then \( \vdash Q > A \).

**Proof.** For the first rule we show \( \bar{x}(y)Q' \) and \( \nu \bar{x}(y)P'(Q') \) have the same typing which is immediate by noting the binding by a private output and that by \( \nu \) both demand the name to be typed by an input type and that an output type is idempotent and can be weakened. For the second rule we show \( \nu x! x(y)P' \) and 0 have the same typing. The context for \( P' \) only contains the output types hence \( \nu x! x(y)P' \) can only contain them which can be weakened for 0. \( \square \)
4 Translations

There is an immediate correspondence between negative formulae and output types and between positive formulae and input types. We turn it into a correspondence between our typing derivations for the $\pi$-calculus and the complete proof-nets.

From $\pi$-terms to proof-nets. A typing derivation with conclusion $\vdash P \Rightarrow A$ is translated as a complete proof-net $R$ with labelled conclusions corresponding to $A$ in such a way that: if $x : \tau_0 \in A$ (resp. $x : \tau_1 \in A$), $R$ has a conclusion labelled by $x$ with type the negative (resp. positive) formula corresponding to $\tau_0$ (resp. $\tau_1$). Moreover any positive conclusion belongs to a cut (following the definition of a complete proof-net).

Derivations are translated by:

- For $!x(\vec{y}).P$, we start with, by induction, a proof-net representing $P$, which has only $? \downarrow$ conclusions. We put a $?\downarrow$-node between the conclusions labelled $\vec{y}$, we remove these labels, and we add a unary $!\downarrow$-node under it with conclusion labelled $x$: the box associated with the premise of the $!\downarrow$-node is the whole proof-net except the $!\downarrow$-node and the free $?\downarrow$-nodes. Finally we cut the $!\downarrow$-node with the $?\downarrow$-node (if any) whose conclusion is labelled $x$ (and we remove the label of the $?\downarrow$-node), or with an augmented 0-ary $?\downarrow$-node if it does not exist.

- For $\bar{x}(\vec{y})P$, by induction we already have a net corresponding to $P$. We add a $\otimes\downarrow$-node between the conclusions labelled $\vec{y}$, and we remove these labels. If there is already a negative edge with name $x$, we add the conclusion of the $\otimes\downarrow$-node as a new premise to the corresponding $?\downarrow$-node. If not, we introduce a new unary $?\downarrow$-node with conclusion labelled $x$ below the new $\otimes\downarrow$-node.

- $P_1|P_2$ is translated as the juxtaposition of the two graphs with some identifications of free nodes (identification of nodes is defined in Section 2, page 7):
  - if $x$ has input type in both $P_1$ and $P_2$, we identify the two $?\downarrow$-nodes corresponding to it, and the two $!\downarrow$-nodes (and thus the two named cuts become one by set union);
  - if $x$ has output type in both $P_1$ and $P_2$, we identify the two $?\downarrow$-nodes corresponding to it;
  - if $x$ has output type in $P_1$ and input type in $P_2$ (or the converse), we identify the $?\downarrow$-node corresponding to the output occurrence and the $?\downarrow$-node which is cut with the $!\downarrow$-node corresponding to the input occurrence. The label $x$ on the conclusion of the $?\downarrow$-node is removed.

- To translate $\nu x P$, we first translate $P$. We then erase the name $x$ on the conclusion of the $!\downarrow$-node with conclusion type $\tau_1$ (the named cut containing this edge becomes an unnamed one).
• 0 is translated as the empty graph.

• The weakening rule is translated by the introduction of a 0-ary ?-node with conclusion labelled \( x \).

From proof-nets to \( \pi \)-terms. Given a complete proof-net \( R \), we build a typing derivation for a judgement \( \vdash P \gg A \) where \( A \) contains typing declarations for the labels of the conclusions of \( R \) (with the input type if a positive conclusion has label \( x \) and the output type otherwise).

The following construction of the translation is by induction on the size of \( R \) (the size of a proof-net is \( n + 2k + p \) where \( n \) is the number of nodes, \( k \) is the number of \( \otimes \)-nodes which are premise of a ?-node involved in a cut, named or not, and \( p \) is the number of unnamed cuts).

• If \( R \) is empty, it corresponds to \( \vdash 0 \gg 0 \).

• If \( R \) has a non-0-ary ?-node at depth 0 named \( x \), it has one or more, say \( m \), \( \otimes \)-nodes above it. We take off each of these \( m \otimes \)-nodes, then we get a net with \( m \times n \) free !-nodes above these \( \otimes \)-nodes, for the arity \( n \) of \( \otimes \) (same arity for all these \( \otimes \)-nodes by typing). By induction we can translate this net into \( \vdash P \gg A, \vec{y}_{1} : \vec{\tau}_{1}, \ldots, \vec{y}_{m} : \vec{\tau}_{m} \). Then \( \vdash \bar{x}(\vec{y}_{1}).\ldots.\bar{x}(\vec{y}_{m})P \gg A, x : (\vec{\tau}_{1})^{n} \) (where length of \( \vec{y}_{j} \) is \( n \), \( 1 \leq j \leq m \)) is the required translation.

• If \( R \) has a 0-ary final ?-node, we remove it, we apply the induction hypothesis and we apply a weakening rule.

• If \( R \) contains an unnamed cut at depth 0, we turn it into a named one by adding a new name \( x \) for its positive conclusion. By induction hypothesis, we have a typing derivation ending with \( \vdash P \gg A, x : \tau_{1} \) and we apply the \((\nu)\) rule to it.

• If \( R \) contains a named cut at depth 0 with name \( x \) and involving the conclusion of a non-0-ary ?-node, take off that cut, then name the ?-node \( y \) and !-node \( x \) with \( y \) fresh, add a new named cut between the !-node and an augmented 0-ary ?-node, then translate the result by induction. We get \( \vdash P \gg A, x : \tau_{1}, y : \tau_{1} \), and take \( \vdash P\{y/x\} \gg A, x : \tau_{1} \) by Lemma 3.

• If \( R \) contains a unique outermost box whose main conclusion (which is a premise of a !-node by definition) is named \( x \), and if \( R \) contains no other nodes except the ?-nodes under the \( \otimes \)-conclusions of the box and the 0-ary ?-node cut with the !-node, then we remove the !-node (with the ?-node cut with it) and the \( \forall \)-node above with the introduction of new names \( y_{1}, \ldots, y_{n} \) as labels for its premises which are ?-nodes. By induction hypothesis, we get a derivation with conclusion \( \vdash P \gg A, \vec{y} : \vec{\tau}_{0} \) (notice that in this case \( \text{md}(A) = 0 \)) and we derive \( \vdash !x(\vec{y}).P \gg A \gg x : (\vec{\tau}_{0})^{n} \).

• If none of the previous cases applies, \( R \) contains more than one box at depth 0, i.e. more than one outermost box. If it contains \( n \) outermost
boxes, we split it into \( n \) proof-nets: one for each box (containing the box, an associated \(!\)-node still with the same name but with a single \(?\)-node above, and a 0-ary \(?\)-node cut with it) with a splitting of the \(?\)-nodes. By induction hypothesis, we have \( \vdash P_1 \triangleright A_1 \), \ldots , \( \vdash P_n \triangleright A_n \) and we get \( \vdash ((P_1|P_2|\ldots)|P_n \triangleright A_1 \odot A_2 \odot \cdots \odot A_n \).

The \( \odot \) operation taking exactly into account the identifications of names.

Above we assume we can choose a \(?\)-node, a \(!\)-node etc. used in each case uniquely by e.g. a certain ordering on nodes. The choice of this ordering does not change the resulting processes, as we shall see soon.

**Example 4.** According to the translation given above, the following three \( \pi \)-terms correspond to the proof-nets of Figure 1:

\[ \nu x(\bar{x} \mid \bar{x} \mid !x.0) \]
\[ !u.\nu x(x.0) \]
\[ \bar{u}(x)(\bar{x}(y'z')(!y'.0)!y'.0 | !z'.0 | !x(yz).\bar{y}) \]

where we write, following the standard convention [MPW92], \( !x.P \) for \( !x().P \) and \( \bar{x} \) for \( \bar{x}().0 \). In the first term, we have the shared redex at a channel named \( x \), with two messages at its output end and one input at its input end (the two messages correspond to two \( \odot \)-nodes above the \(?\)-node). For the second term, \( x \) corresponds to the auxiliary \( \odot \)-conclusion of the box, while the free \( v \) corresponds to the similarly named \(?\)-node. For the third term, the cut elimination depicted in Figure 2 leads to, in one step, the following term:

\[ \bar{u}(x)((\nu y'z')(!y'.0)!y'.0 | !z'.0 | \bar{y}') | !x(yz).\bar{y}) \]

This term corresponds to the proof-net at the bottom in Figure 2. The three steps in Figure 2 roughly correspond to the choice of a redex at \( x \) (unique in this case), copying the body \( \bar{y} \) of the left-most replication, and identifying the first and second output parameters \( y'z' \) and the first and second input parameters \( yz \) (in the copy) respectively.

**Structural congruence and \( \sigma \)-equivalence.** We confirm several basic properties of the translations. First, two typable \( \pi \)-terms \( \vdash P \triangleright A \) and \( \vdash Q \triangleright A \) are called \( \sigma \)-equivalent (in relation with the corresponding notion for the \( \lambda \)-calculus [Reg94]), denoted \( \vdash P \simeq_\sigma Q \triangleright A \), if they translate into the same proof-net.

**Proposition 1.** Let \( \vdash P \triangleright A \) and \( \vdash Q \triangleright A \) be two typed \( \pi \)-terms,

\[ P \equiv Q \iff \vdash P \simeq_\sigma Q \triangleright A \]

**Proof.** The implication \( P \equiv Q \Rightarrow \vdash P \simeq_\sigma Q \triangleright A \) is trivial except the following cases:
• $\bar{x}(\vec{u})\bar{y}(\vec{v})P \equiv \bar{y}(\vec{v})\bar{x}(\vec{u})P$ (with $x \notin \vec{v}$ and $y \notin \vec{u}$ and $\vec{u} \cap \vec{v} = \emptyset$): if $x$ and $y$ are different, the result is immediate, otherwise both $\bar{x}(\vec{u})$ and $\bar{x}(\vec{v})$ add a $\otimes$-node above the $?$-node corresponding to $x$ and the order does not matter.

• $\bar{x}(\vec{u})(PQ) \equiv (\bar{x}(\vec{u})P)Q$ (with $\vec{u} \notin Q$): the only interesting case is when $x \in Q$, in which case we just have to remark that in the first term a $\otimes$-node is added to the common $?$-node coming from $P$ and $Q$ and corresponding to $x$, while in the second term the $?$-node is added to a $\otimes$-node coming from $P$ and then it is merged with the $?$-node with name $x$ coming from $Q$.

• $\nu y\bar{x}(\vec{u})P \equiv \bar{x}(\vec{u})\nu yP$ (with $y \notin \{x, \vec{u}\}$): the $\nu y$ operation hides the conclusion of a cut node which is not concerned by the $\bar{x}(\vec{u})$ operation.

In the other direction: $\vdash P \equiv_{\sigma} Q \Rightarrow P \equiv Q$, we can consider the particular case where $P$ and $Q$ are canonical forms (otherwise $P \equiv P_0$ and $Q \equiv Q_0$ with $P_0$ and $Q_0$ canonical forms entails $P_0 \equiv_{\sigma} P \equiv_{\sigma} Q \equiv_{\sigma} Q_0$ and by this particular case $P \equiv P_0 \equiv Q_0 \equiv Q$).

Let $\mathcal{R}$ be the translation of $P$ and $Q$. We work by induction on $\mathcal{R}$. If $\mathcal{R}$ contains unnamed cuts at depth 0, we introduce conclusions with fresh names for these cuts. This exactly corresponds to removing the $\nu$ constructions in head position in $P$ and $Q$, and by typability we cannot have $\nu$s on non-free variables. Using $\equiv$, the order of the $\nu$s does not matter and we can apply the induction hypothesis.

Observe that two output prefixes $\bar{x}_1(\vec{u}_1)$ and $\bar{x}_2(\vec{u}_2)$ commute by $\equiv$ as soon as the conclusion of the corresponding $\otimes$-nodes goes to named $?$-nodes (more generally, if one is not above the other in $\mathcal{R}$). We consider a $\otimes$-node at depth 0 above a named $?$-node. By the remark, $P$ and $Q$ are equivalent to terms starting with the corresponding $\bar{x}(\vec{u})$. We remove it and we apply the induction hypothesis.

We now consider the case where $\mathcal{R}$ does not contain any $\otimes$-node at depth 0. We split $\mathcal{R}$ according to its boxes. This corresponds to splitting both $P$ and $Q$ through parallel composition (which is commutative and associative according to $\equiv$).

Finally, if $\mathcal{R}$ is reduced to one box (with associated $!$, $?$ and cut), we have $P = !x(\vec{y}).P'$ and $Q = !x(\vec{y}).Q'$ and we apply the induction hypothesis to $P'$ and $Q'$.

Second we show the two translations are mutually inverse.

**Proposition 2.** If we translate $\mathcal{R}$ into a typed process, and this typed process into a proof-net $\mathcal{R}'$ back again, then $\mathcal{R}$ and $\mathcal{R}'$ are isomorphic. Symmetrically, if we translate $\vdash P \Rightarrow A$ into a proof-net, and this proof-net into a process say $\vdash Q \Rightarrow A$ back again, then $P \equiv Q$.

**Proof.** In each direction, we use rule induction on the translation rules. All cases are mechanical. For example, if we translate $\mathcal{R}$ by choosing a non-0-ary
?-node, then this is translated into a sequence of outputs at the same subject, say \( x \), in the form:

\[
\bar{x}(\vec{y}_1) . x(\vec{y}_n) P
\]

where \( x \) does not occur in \( P \). Apply the translation of this term and we obtain precisely the same net as before. Similarly the other direction. \( \square \)

**Corollary 1.** The translation of \( \mathcal{R} \) as given before is determined uniquely up to \( \equiv \) regardless of the choices of nodes in translation.

**Proof.** Suppose two different ways to translate \( \mathcal{R} \) result in \( \equiv \)-distinct processes. Then their translations do not result in the same net by Proposition 1, contradicting Proposition 2. \( \square \)

**Simulations.** We turn our attention to dynamics.

**Theorem 1.** \( P \rightarrow Q \) if and only if the translation of \( \vdash P \triangleright A \) reduces to the translation of \( \vdash Q \triangleright A \) by cut elimination at depth 0. \( P \\vdash Q \) if and only if the translation of \( \vdash P \triangleright A \) reduces to the translation of \( \vdash Q \triangleright A \) by extended cut elimination at depth 0.

**Proof.** If \( P \) reduces to \( P' \) (with \( \vdash P \triangleright A \) and thus \( \vdash P' \triangleright A \) by Lemma 5), then there is an active input and an active output sharing the same channel, say \( x \). By structural equality (which is harmless by Proposition 1), we can transform the process so that all possible redexes at \( x \) can be juxtaposed:

\[
\Pi_i ! x(\vec{y}). P_i | \Pi_j \bar{x}(\vec{y}) Q_j
\]

(with an enclosing reduction context which does not contain an input at \( x \)). By translation this becomes a named cut (or an un-named one if the enclosing context contains \( \nu x \)). The rest follows by observing the precise correspondence between the reduction in the \( \pi \)-calculus and the construction of a cut elimination. First, we choose one input and one output at \( x \) from the above, which corresponds to the choice of one \( \exists \)-node and one \( \otimes \)-node in the cut elimination. Then we make the copy and pass names to obtain, in the place of \( \bar{x}(\vec{y}) Q_j \):

\[
(\nu \vec{y})(P_i | Q_j)
\]

which corresponds to making the copy of the content of a box (corresponding to copying \( P_i \) above from under the replication), making new cuts at \( \vec{y} \) (corresponding to \( | \)), and hiding these cuts (corresponding to \( (\nu \vec{y}) \)). Note each step exactly corresponds to each other in both ways.

If \( \mathcal{R} \) reduces to \( \mathcal{R}' \), then there is an either named or unnamed cut. Translate this cut (which we can assume named without loss of generality, by considering a name hiding operation at the level of proof-nets), by which we obtain a term of the form \( \Pi_i ! x(\vec{y}). P_i | \Pi_j \bar{x}(\vec{y}) P_{2j} \) (where \( \Pi \) denotes \( n \)-fold composition). The rest follows again by the correspondence between the construction of cut elimination and the reduction. In each direction, it is crucial that only the redex changes the shape: other parts remain the same, hence we can use induction. The cases for extended reduction are similar. \( \square \)
5 Additional constraints

This section incrementally adds constraints on polarised proof-nets and processes one by one, leading to the coincidence of their fragments which fully abstractly embed the call-by-name and call-by-value \(\lambda\mu\)-calculi (or, equivalently, a simply typed \(\lambda\)-calculus extended with callcc).

5.1 Reception determinism

If we modify the definition of \(\otimes\) by demanding that \((A, x : \tau_1) \otimes x : \tau_1\) is no longer defined (thus constraining the (\()\)-rule), we get proof-nets in which each \(!\)-node has exactly one premise. Both constraints act locally and in precise correspondence with each other, hence the same simulation result as Theorem 1 holds.

Computationally this means that all messages to the same input name (i.e. messages going through the same channel) always reach a unique replicated input process [San97, BHY01].

5.2 Emission determinism

We define, following [BHY01], a partial operation \(\Box\) on modes:

\[
\begin{align*}
I \Box I &= I \\
I \Box 0 &= 0 \\
0 \Box I &= 0
\end{align*}
\]

and stipulate \(0 \Box 0\) is not defined.

We add to judgements an input/output mode \(\phi\): \(\vdash_{\phi} P \triangleright A\).

As a result, a typed process with the \(I\)-mode never has an active output (hence redex) and one with the \(O\)-mode has at most one active output (hence redex). These properties are invariant under reduction.

A type derivation will now correspond to a proof-net with at most one \(\otimes\)-node in each box and at depth 0. If the conclusion of the derivation has mode \(I\), the proof-net does not have any \(\otimes\)-node at depth 0. Again these properties act locally in respective formalisms’ derivations while preserving an exact structural correspondence: we can easily check that the same mutual simulation result as Theorem 1 holds.
5.3 Refined emission determinism

We can further remove the rule $\vdash I \Pi \triangleleft A \vdash O \Pi \triangleleft A$ from the above, by which a typed process with the $\circ$-mode now has exactly one active output, which is again invariant under reduction. This also corresponds to a well-defined subset of the presented proof-nets: in this case the derivation corresponds to a proof-net with exactly one $\otimes$-node in each box. By this constraint a proof-net has one $\otimes$-node at depth 0 if and only if the conclusion of the derivation in the corresponding $\pi$-term has mode $O$. The number of $\otimes$ at each level controls the number of activities (or threads) in computation.

5.4 Acyclicity

A relation $R$ on a finite set $E$ is acyclic\(^1\) if there is no sequence $x_1, \ldots, x_n$ of elements of $E$ with $x_i R x_{i+1}$ and $x_n R x_1$.

We add to judgements an acyclic relation on the names appearing in the context: $\vdash (\phi) P \triangleright A; R$.

If $A; R$ is a context with $x \notin A$, we define $x : \tau R A$ to be the context $A, x : \tau ; R$. If the resulting $R'$ is not acyclic, we stipulate that $x : \tau R A$ is not defined.

\[
\begin{align*}
\vdash (\phi_1) P_1 \triangleright A_1; R_1 & \vdash (\phi_2) P_2 \triangleright A_2; R_2 \\
\vdash (\phi_1, \triangledown) P_1 | P_2 \triangleright A_1 \otimes A_2; R_1 \cup R_2 \\
\vdash (\phi) P \triangleright A, x : \tau \Pi; R \\
\vdash (\phi) \mu x P \triangleright A; R \uparrow A
\end{align*}
\]

A proof-net is acyclic if the directed graph obtained by:

- positive edges are oriented upwardly and negative edges are oriented downwardly;
- conclusions of $\otimes$-nodes are removed;
- each !-node with conclusion $!N$ and with $n$ boxes with auxiliary conclusions $\Gamma_1, \ldots, \Gamma_n$ is replaced by a node with an edge $!N$ oriented upwardly (that is towards the !-node) and edges to the ?-nodes under $\Gamma_1, \ldots, \Gamma_n$ oriented downwardly (that is towards the ?-nodes);

\(^1\)The definition herein gives a different but equivalent presentation of the acyclicity in strongly normalising processes studied in [YBH04].
• cuts are replaced by directed edges from the corresponding $\xi$-node to the corresponding $\nabla$-node

is acyclic.

The typing derivations with acyclic relations correspond to acyclic proof-nets. This is because any sub-net of an acyclic proof-net (in the sense above) is acyclic, similarly for a process. The rest follows by induction. Thus we again obtain, by restricting Theorem 1 to these restricted processes/nets, the exact mutual simulation result.

5.5 Putting everything together

If we put together all the constraints we have stipulated, we get a notion of typed $\pi$-calculus which exactly corresponds to proof-nets satisfying the correctness criterion of polarised proof-nets. As a consequence there exist fully complete embeddings of various systems of classical logic into these objects (typed $\pi$-terms or correct proof-nets).

The calculus thus obtained (without the refinement in Section 5.3) precisely corresponds to the $\pi^C$-calculus (the control $\pi$-calculus) in [HYB04].

Concerning proof-nets, let us look at the syntax presented in [Lau05]. If we consider correct axiom-free proof-nets, we replace $\nabla$-nodes by simple edges, we glue together trees of successive $\otimes$-nodes and 1-nodes (resp. $\exists$-nodes and $\bot$-nodes) and we apply the completion procedure of Section 2, we obtain a proof-net as described here and satisfying all the constraints introduced in this section. Conversely, the proof-nets used here allow for slightly more sharing than in [Lau05] and we have to do a simple unfolding of boxes for $\nabla$-nodes above a $\otimes$-node. We first introduce a $\nabla$-node between each $\otimes$-node and the $\xi$-node below it (inside the box if the $\otimes$-node is inside a box). We replace $n$-ary $\otimes$-nodes (resp. $\exists$-nodes) by a corresponding tree of binary $\otimes$-nodes and 1-nodes (resp. $\exists$-nodes and $\bot$-nodes) with the appropriate number of leaves. Finally, if we have a $\nabla$-node whose conclusion is both in a cut and premise of a $\otimes$-node, we make a copy of the $\nabla$-node together with the (unique) associated box (the auxiliary conclusions of the copy of the box becoming new premises of the corresponding $\xi$-nodes as we did for cut elimination) and we modify the cut so that the conclusion of the new $\nabla$-node belongs to it instead the original $\nabla$-node.

The proof-net thus obtained belongs to the syntax of [Lau05] and satisfies the associated correctness criterion.

6 Discussions

We discuss some of the possible extensions of the correspondence results discussed in the preceding sections.

Axioms. The axiom for a proposition which may include a variable acts as a generic link in proof-nets. In the $\pi$-calculus, such a generic link is realised as a
process which sends a datum it receives at one port to another, whose semantic
significance is studied, for example, [HY95, Mi92a]. This is a basic behaviour
in communicating systems such as network routers and is called forwarder in
the study of actor model, cf. [Agh86]. We can directly represent such a generic
link using free name passing [BHY03]. Here in order to preserve the preceding
translation structure, we add a generic link to the grammar of processes.

\[
P ::= \ldots \mid x \mapsto y
\]

whose typing is given by:

\[
\Gamma \vdash x \mapsto y \mapsto x : \tau_1, y : \tau_2, (x, y)
\]

\(x \mapsto y\) is a non \(\eta\)-expanded and “polymorphic” version of identity (for example,
at type \((\_\)\), \(x \mapsto y\) and \(!x.g\) are semantically equivalent). For the extended
reduction we add two rules. The first rule is:

\[
x \mapsto y|C[\bar{x}(\bar{z})P] \Downarrow x \mapsto y|C[\bar{y}(\bar{z})P]
\]

which is also added to the (non-extended) reduction rule when \(C[\_\_\_]\) is the trivial
identity context \([\_\_]\). The second rule:

\[
\nu x(x \mapsto y) \Downarrow 0
\]

garbage-collects the link when it is no longer necessary.

The corresponding proof-net is built from the standard rule for the axiom
link which connects two ends of mutually dual interfaces:

\[
\begin{array}{c}
P_1 \quad \bar{P}_1 \\
\end{array}
\]

by applying:

\[
\begin{array}{c}
? \quad ! \quad ? \\
x \quad \bar{x} \quad y
\end{array}
\]

Polymorphism. We consider parametric polymorphism for the \(\pi\)-calculus in
which an existentially typed output not only sends channels but also instantiates
types (and, dually, an universally typed input expects both concrete messages
and type instantiation). We consider polymorphism where types are explicitly
passed at the time of communication, with explicit duality between existential
and universal types. The resulting constructions correspond to the polymorphic
\(\pi\)-calculus studied in [Tur95, PS00] at the level of dynamics and [BHY03] at the level of types. We first extend the grammar of types:

\[
\begin{align*}
\tau_I &::= \ldots | X \\
\tau_O &::= \ldots | \bar{X}
\end{align*}
\]

We set above all type variables \(X, Y, \ldots\) are of mode \(I\) and stipulate types of the form \(\bar{X}\) have the mode \(O\). Then \(\bar{X}\) returns \(X\).

The grammar of processes is extended as:

\[
P ::= \ldots | !x\Lambda \bar{X}(\vec{y}).P \mid \pi T \bar{\tau}^I(\vec{y})P
\]

where in the first construct \(\bar{X}\) is a vector of mutually distinct type variables which acts as a binder. The grammar of types now includes quantifications:

\[
\begin{align*}
\tau_I &::= \ldots | (\forall \bar{X}^I) \\
\tau_O &::= \ldots | (\exists \bar{X}^I)
\end{align*}
\]

The duality is extended naturally: for example the dual of \((\forall X \bar{X}(X))^I\) is \((\exists X.X(X))^I\). The typing rules are standard [BHY03] with \(FV(A)\) standing for the set of type variables in \(A\):

\[
\frac{\vdash P \triangleright A, \vec{y} : \tau_0 \quad \text{md}(A) = 0 \quad \{\bar{X}\} \cap FV(A) = \emptyset}{\vdash !x\Lambda \bar{X}(\vec{y}).P \triangleright A \circ x : (\forall \bar{X}\tau_0)^I}\quad \text{in} \forall
\]

\[
\frac{\vdash P \triangleright A, \bar{y} : \bar{\tau}^I \{\bar{\tau}/\bar{X}\} \quad \{\bar{X}\} \cap FV(A) = \emptyset}{\vdash \hat{x} T \bar{\tau}^I(\bar{y})P \triangleright A \circ x : (\exists \bar{X}\bar{\tau}^I)^I}\quad \text{out} \exists
\]

The rules for extended reduction now accompany type instantiation:

\[
!x\Lambda \bar{X}(\vec{y}).P \triangleright C[xT \bar{\tau}^I Q] \downarrow !x\Lambda \bar{X}(\vec{y}).P \triangleright C[v\\vec{y}(P[\bar{\tau}/\bar{X}]|Q)]
\]

\[
\nu x!x\Lambda \bar{X}(\vec{y}).P \downarrow 0
\]

with only the first rule and a trivial \(C[\ ]\) for (non-extended) reduction. This polymorphic extension works with the generic link we introduced in the previous paragraph. The strong normalisation of this fragment is proved in [BHY03].

The corresponding proof-nets add the familiar rules for quantifications for types of the forms \(\forall X \otimes_{1 \leq i \leq n} P_i\) and \(\exists X \otimes_{1 \leq i \leq n} N_i\) through new nodes:

\[
\frac{P\{P'/X\}}{\exists X P} \quad \frac{N}{\forall X N}
\]

The cut elimination for the proof-nets follows [Gir87, Gir91].
Additives. Additives in Linear Logic present an interactional, fully dualised form of sum types. Their underlying dynamics is closely related with an encoding of a local choice in the π-calculus discussed in [Mil92a]. Its typed representation in the π-calculus is discussed in [HVK98] (see also [Vas94, THK94]) following which we extend the grammar of processes as follows.

\[ P ::= \ldots | !x[\&_{i \in I}(y_i)].P_i | \bar{x}_{\text{in}}(y)P \]

We call the first form branching and the second selection. The intuition is that a branching waits with multiple options while a selection selects one of the provided options at the time of interactions. Types are extended as:

\[ \tau_I ::= \ldots | [\&_{i \in I} \tau_0_i] \]

\[ \tau_0 ::= \ldots | [\oplus_{i \in I} \tau_1_i] \]

We type branching and selection as follows.

\[ \begin{array}{c}
\vdash P \triangleright A, y_i : \tau_0_i \quad \vdash !x[\&_{i \in I}(y_i)].P_i \triangleright A \circ x : [\&_{i \in I} \tau_0_i]^i \\
\vdash P \triangleright A, y_i : \tau_1_i \\
\vdash \bar{x}_{\text{in}}(y)P \triangleright A \circ x : [\oplus_{i \in I} \tau_1_i]^i 
\end{array} \]

The extended reduction is given as:

\[ !x[\&_{i \in I}(y_i)].P_i | C[\bar{x}_{\text{in}}(y)Q] \setminus !x[\&_{i \in I}(y_i)].P_i | C[\nu y_i (P_i | Q)] \]

together with the garbage collection rule as before, which we omit. The (non-extended) reduction contains only the particular case where \( C[\ ] \) is trivial.

The corresponding deduction in proof-nets are the standard additive rules with boxes [Lau02], with focalised formulae of the forms \( \&_{1 \leq i \leq n} \tau_0_j \bot_{1 \leq j \leq n} \tau_1_i \) for negative formulae and \( \oplus_{1 \leq i \leq n} \otimes_{1 \leq j \leq n} N_{ij} \) for positive formulae. The cut elimination eliminates branches which are not chosen.

We can further extend this fragment with linearity (which, at the level of processes, means simply adding inputs without replication with the corresponding type discipline as in [YBH04], enabling precise representation of state-changing recursive agents [Mil92b]; and, at the level of polarised proof-nets, adding the linear lifting following [Lau02]). These extensions reveal a close link which can exist between the systematic articulation of well-behaved processes centring on name passing communication, and the logical structures which arise from the study of distilled proof dynamics.

\[ \text{At the level of dynamics, these two constructs can be faithfully captured without syntactic extensions, as indicated in [Mil92a]: } !x[\&_{1 \leq i \leq n}(y_i)].P_i \text{ becomes } !x(b(c_1 \ldots c_n)).\Pi_i c_i(y_i).P_i \text{ and } \bar{x}_{\text{in}}(y)P \text{ becomes } \bar{x}(b(c_1 \ldots c_n)).c_i(y)P. \]
[There are strong relations between this work and the work by T. Ehrhard and O. Laurent published in CONCUR’07. However the present work has been completely developed before the Ehrhard-Laurent’s one. This is why we do not mention it here. A comparison between the two works is sketched in the introduction of their CONCUR paper.]

References


