πI: a symmetric calculus based on internal mobility

Davide Sangiorgi

INRIA- Sophia Antipolis, France. Email: davide@cma.cma.fr.

1 Motivations

The π-calculus is a development of CCS where names (a synonymous for "channels") can be passed around. This permits the description of mobile systems, i.e., systems whose communication topology can change dynamically.

Name communication gives π-calculus a much greater expressiveness than CCS. For instance, in the π-calculus there are simple and intuitive encodings for: *Data values* [MPW92, Mil91], *agent-passing* process calculi [Tho90, San92] (i.e., calculi where terms of the language can be exchanged), the *λ-calculus* [Mil92], certain *concurrent object-oriented languages* [Jon93, Wal95], the locality and causality relations among the activities of a system, typical of *true-concurrent behavioural equivalences* [BS95]. In CCS, the modelling of such objects is possible, at best, in a clumsy and unnatural way — for instance making heavy use of infinite summations.

But research has also showed that the π-calculus has a much more complex mathematical theory than CCS. This shows up in:

- The *operational semantics.* Certain transition rules of the π-calculus are hard to assimilate.
- The *definition of bisimulation.* Various definitions of bisimilarity have been proposed for the π-calculus, and it remains unclear which form should be preferred. Moreover, most of these bisimilarities are not congruence relations.
- The *axiomatisations.* The axiomatisations of behavioural equivalences for the π-calculus — and in particular the proof of the completeness of the axiomatisations — is at least one order of magnitude more complicated than the corresponding axiomatisations for CCS.
- The *construction of canonical normal forms.* In general we do not know how to transform a π-calculus process P into a normal form which is unique for the equivalence class of P determined by the behavioural equivalence adopted.

In CCS, these problems are well-understood and have simple solutions [Mil89, BK85, DKV91].

There is, therefore, a deep gap between CCS and π-calculus, in terms of expressiveness and mathematical theory. The main goal of the paper is to explain this gap and to examine whether there are interesting intermediate calculi. For instance, are the complications of the theory of the π-calculus w.r.t. that of CCS an inevitable price to pay for the increase in expressiveness?

We shall isolate and analyse one such intermediate calculus, called πI. This calculus appears to have considerable expressiveness: Data values, the lambda calculus, agent-passing calculi, the locality and causality relations of true-concurrent behavioural equivalences can be modelled in πI much in the same way as they are in the π-calculus. But, nevertheless, the theory of πI remains very close to the theory of CCS: Alpha conversion is, essentially, the new ingredient. To obtain πI, we separate the mobility mechanisms of the π-calculus into two, namely *internal* mobility and *external* mobility. The former arises when an input meets a bound output, i.e., the output of a private name; the latter arises when an input meets a free output, i.e., the output of a known name. In πI only internal mobility is retained — the free output construct is disallowed. A pleasant property of πI is the full symmetry between input and output constructs. The operators of matching and mismatching, that in the π-calculus implement a form of case analysis on names and are important in the algebraic reasoning, are not needed in the theory of πI.

Sections 2-4 are devoted to introducing πI and its basic theory. The encoding of the λ-calculus into πI is studied in Section 5: It is challenging because all known encodings of the λ-calculus into π-calculus exploit, in an important way, the free-output construct, disallowed in πI. We sketch the comparison between πI and agent-passing calculi in Section 6. There is an exact correspondence, in terms of expressiveness, between a hierarchy of subcalculi of πI and a hierarchy of agent-passing calculi obtained from the Higher-Order π-calculus [San92]. The definitions of two hierarchies rely on the order of the typing systems of πI and of the Higher-Order π-calculus.

In this short version of the paper, the presentation is kept rather informal; for technical details and proofs, we refer to the full version [San95].

Acknowledgements. I have benefited from discussions with Gerard Boudol, Claudio Calvelli, Robin Milner, David Turner and David Walker, and from the comments of the anonymous referees. This research has been supported by the Esprit BRA project 6454 "CONFER".

2 The calculus πI

In this section we introduce (the finite part of) πI. We examine the move from π-calculus to πI from three different angles: First, our guiding criterion is symmetry; then we take into account the mobility mechanisms; finally, we focus on the algebraic theory. There are not compelling reasons for wanting symmetry: Our major motivation is elegance, which will show up in the presentation of the calculus and of its properties.

Throughout the paper we use a tilde ($\tilde{}$) to denote a finite and possibly empty tuple. All notations are extended to tuples componentwise.

2.1. Looking for symmetry: From π-calculus to πI We shall derive the grammar for πI from the one below, which collects the principal operators of the π-calculus, namely guarded sum, parallel composition and restriction.

Symbols $x, y, z, \ldots$ will range over the infinite set of names; P, Q and R will be metavariables over processes; prefixes, ranged over by α, can be of the form τ (*silent prefix*), $x(y)$ (*input prefix*), or $\overline{x}y$ (*free-output prefix*):

$$P ::= \textstyle\sum_{i\in I} \alpha_i . P_i \mid P \mid P \mid \nu\, x\, P$$
$$\alpha ::= \tau \mid x(y) \mid \overline{x}y$$

I is a finite indexing set; if I is empty, we abbreviate the sum as $\mathbf{0}$. As usual, $+$ is binary sum. Sometimes, we shall write $\alpha_1 . P_1 + \ldots + \alpha_n . P_n$ for $\sum_{1\leq i\leq n} \alpha_i . P_i$.

An input prefix $x(y).P$ and a restriction $\nu\, y\, P$ bind all free occurrences of name y in P. *Free* and *bound* names of processes and of prefixes, and *alpha conversion* are defined as expected. $P\{x/y\}$ denotes the substitution of x for y in P, with renaming possibly involved to avoid capture of free names. In examples, the object part of prefixes will be omitted if not important. A process $\alpha.\mathbf{0}$ will often be abbreviated as α, and $\nu\, x_1 \ldots \nu\, x_n\, P$ as $\nu\, x_1, \ldots, x_n\, P$. Sum and parallel composition will have the lowest syntactic precedence; substitution the highest.

The grammar above does not mention the match and mismatch operators, written $[x = y]P$ and $[x \neq y]P$, respectively. The former means: "if x equal to y, then P"; the latter means "if x different from y, then P". Match and mismatch are often included in the π-calculus, mainly because very useful in the algebraic theory. But they will not be needed in the algebraic theory of $\pi\mathrm{I}$, as shown in Section 3.

We wish to make two remarks about the π-calculus language above presented. The first regards the asymmetry between the input and output constructs, namely $x(y).-$ and $\overline{x}y.-$. The asymmetry is both syntactic — the input is a binder whereas the output is not — and semantic — in an input *any* name can be received, whereas in an output a *fixed* name is emitted. The second remark regards a derived form of prefix, called *bound output*, written $\overline{x}(y)$ as an abbreviation for $\nu\, y\, \overline{x}y$. Bound output plays a central role in π-calculus theory, for instance in the operational semantics and in the axiomatisation. In the operational semantics, bound output is introduced in the OPEN rule, one of the of the two rules for restriction:

$$\textsc{open}: \quad \frac{P \xrightarrow{\overline{x}y} P'}{\nu\, y\, P \xrightarrow{\overline{x}(y)} P'} \quad x \neq y\,.$$

(We can make an analogy between bound output and silent prefix: Both can be viewed as derived operators — $\tau.P$ as abbreviation for $\nu\, x\, (x.P \mid \overline{x})$, for some x not free in P; and both are needed in the operational semantics and axiomatisations.)

Having noticed the importance of bound output, we can reasonably add it to the grammar of prefixes:

$$\alpha ::= \tau \mid x(y) \mid \overline{x}y \mid \overline{x}(y)\,.$$

The new syntax still contains asymmetries: First, the free-output construct has no input counterpart. Second, input and bound output, although syntactically similar — both are binders — are semantically very far apart, as revealed by the interactions they can participate in: *Any* name can be received through an input, whereas only a *fresh* name can be emitted through a bound output.

We move to πI by eliminating the free output construct.

Definition 1 (finite πI). The class of finite πI processes is described by the following grammar:

$$P ::= \textstyle\sum_{i\in I} \alpha_i.P_i \mid P \mid P \mid \nu\, x\, P$$
$$\alpha ::= \tau \mid x(y) \mid \overline{x}(y)\,.$$

In πI, the input and output constructs are truly symmetric: Since only outputs of private names are possible, an input $x(y).P$ means "receive a fresh name at x", which is precisely the dual of the output $\overline{x}(y).P$. Indeed, we can define an operation "`dual`" which transforms every output into an input and vice versa: The symmetry of the calculus is then manifested by the fact that `dual` commutes with the transition relation (Lemma 2).

2.2. Internal and external mobility Above, the motivation to the introduction of πI was symmetry. A more pragmatic motivation is given here.

What distinguishes π-calculus from CCS is *mobility*, that is, the possibility that the communication linkage among processes changes at run-time. In the π-calculus there are two mechanisms to achieve mobility, which are embodied in the two communication rules of the calculus (usually called COM and CLOSE). Accordingly, we can distinguish between two forms of mobility, *internal mobility* and *external mobility*. Internal mobility shows up when a bound output meets an input, for instance thus:

$$\overline{x}(y).P \mid x(y).Q \xrightarrow{\tau} \nu\, y\,(P \mid Q)\,.$$

Two separate local (i.e., internal) names are identified and become a single local name. The two participants in the interaction, $\overline{x}(y).P$ and $x(y).Q$, agree on the bound name; for this, some alpha conversion might have to be used. The interaction consumes the two prefixes but leave unchanged the derivatives underneath. With internal mobility, alpha conversion is the only form of name substitution involved.

External mobility shows up when a free output meets an input, for instance thus:

$$\overline{x}y.P \mid x(z).Q \xrightarrow{\tau} P \mid Q\{y/z\}\,.$$

Here, a local name gets identified with a free (i.e., external) name. In this case, alpha conversion is not enough: Name y is free, and might occur in Q; hence in general z cannot be alpha converted to y. Instead, a substitution must be imposed on the derivatives so to force the equality between y and z.

In πI, only internal mobility is present. Studying πI means examining internal mobility in isolation, and investigating its impact on expressiveness and

mathematical theory. From the experimentation that we have conducted so far, it appears that internal mobility is responsible for much of the expressiveness of the π-calculus, whereas external mobility is responsible for much of the semantic complications. Some evidence to this will be given in the remaining sections.

2.3. Some advantages of the theory of πI Through examples, we show a few weaknesses of the theory of the π-calculus, and we show why they do not arise in πI.

Below, $\sim_\pi$ denotes π-calculus original bisimilarity, as in [MPW92]; it is sometimes called *late bisimilarity*. (The examples we use are rather simple, so we do not need to recall the definition of $\sim_\pi$.) Consider the π-calculus process $x \mid \overline{y}$, where x and y are different names. We can rewrite it as follows, using expansion:

$$x \mid \overline{y} \sim_\pi x.\overline{y} + \overline{y}.x\,. \tag{1}$$

However, this equality can break down underneath an input prefix:

$$z(x).(x \mid \overline{y}) \not\sim_\pi z(x).(x.\overline{y} + \overline{y}.x)\,. \tag{2}$$

The process on the left-hand side can receive y in the input and become $y \mid \overline{y}$, which then can terminate after a silent step. This behaviour is not matched by the process $z(x).(x.\overline{y} + \overline{y}.x)$, which, upon receiving y, can only terminate after two visible actions.

To have a fully-substitutive equality, some case analysis has to be added to the expansion (1), by means of the *match* operator:

$$x \mid \overline{y} \sim_\pi x.\overline{y} + \overline{y}.x + [x = y]\tau\,.$$

The third summand allows a τ if x and y are the same name. This equality can now be used underneath a prefix:

$$z(x).(x \mid \overline{y}) \sim_\pi z(x).(x.\overline{y} + \overline{y}.x + [x = y]\tau)\,.$$

The above discussion outlines two important points: First, π-calculus bisimilarity is not preserved by input prefix; second, to get congruence equalities some case analysis on names might be needed. In the above example, one level of case analysis was enough, but for more complex processes it can be heavier; the mismatch operator might be needed too. In general, if in the π-calculus we wish to manipulate a subcomponent P of a given process algebraically, then we cannot assume that the free names of P will always be different with each other: By the time the computation point has reached P, some of these names might have become equal. Therefore we have to take into account all possible equalities and inequalities among these names; if they are n, then there are 2^n cases to consider.

These inconvenients do not arise in πI. Bisimilarity is naturally a full congruence, and no case analysis on names is required. For instance, consider processes $x \mid \overline{y}$ and $x.\overline{y} + \overline{y}.x$ in (1), and let $\sim$ be πI bisimilarity. As in the π-calculus, so in πI the two processes are bisimilar; but, unlike the π-calculus, their bisimilarity is preserved by input prefix:

$$z(x).(x \mid \overline{y}) \sim z(x).(x.\overline{y} + \overline{y}.x)\,.$$

This because in πI only fresh names are communicated, hence the free name y can never be received in an input at z. The absence of case analysis explains why match has not been included among the πI operators.

Besides late bisimilarity, other formulations of bisimilarity for the π-calculus have appeared in the literature (see [FMQ94]), and it is far from clear which one should be preferred. (Some of these relations are full congruences, but all require the case analysis on names mentioned before.) The differences among these bisimilarities are due to the different interpretation of name substitution in an input action. The choice is about *when* should such a substitution be made: For instance immediately, in the input rule, or later, in the communication rule, or only when the name received is needed. The choice affects the resulting behavioural equivalence, since a substitution can change the relationships of equality or inequality among names. In πI, alpha conversion is the only form of name substitution needed. Alpha conversion is semantically harmless, because it does not change the equalities and inequalities among names; hence in πI the bisimilarity relation is unique.

3 Basic theory of πI

We consider the basic theory of πI: Operational semantics, bisimilarity, axiomatisation, construction of canonical normal forms. In all these cases, a clause for alpha conversion represents the only difference w.r.t. the theory of CCS. An exception to this is the appearance of a restriction in the communication rule for πI.

3.1. Operational semantics and bisimilarity We write $\overline{\alpha}$ for the complementary of α; that is, if $\alpha = x(y)$ then $\overline{\alpha} = \overline{x}(y)$, if $\alpha = \overline{x}(y)$ then $\overline{\alpha} = x(y)$, and if $\overline{\alpha} = \tau$, then $\overline{\alpha} = \alpha$. We write $P \equiv_\alpha Q$ if P and Q are alpha convertible. We write $\mathrm{fn}(P), \mathrm{bn}(P)$ (resp. $\mathrm{fn}(\alpha), \mathrm{bn}(\alpha)$) for the *free names* and the *bound names* of P (resp. α). The *names* of P or α, written $\mathrm{n}(P)$ and $\mathrm{n}(\alpha)$, are the union of their free and bound names. Table 1 contains the set of the transition rules for πI. We have omitted the symmetric of rule PAR. The only formal difference w.r.t. the set of rules for CCS is the presence of the alpha conversion rule and the generation of a restriction in the communication rule. Unlike the π-calculus, there is only one rule for communication and one rule for the restriction operator. Note that the alphabet of actions is the same as the alphabet of prefixes. We call a transition $P \xrightarrow{\tau} P'$ a *reduction.*

We define an operation `dual` which complements all visible prefixes of a πI process: If $P \in \pi\mathrm{I}$, then $\overline{P}$ is obtained from P by transforming every prefix α into the prefix $\overline{\alpha}$. Operation `dual` can be defined on πI because of its *syntactic* symmetry. The following lemma shows that the symmetry is also *semantic.*

Lemma 2. *If* $P \xrightarrow{\alpha} P'$, *then* $\overline{P} \xrightarrow{\overline{\alpha}} \overline{P'}$. □

Note that since $\overline{\overline{P}} = P$, the converse of Lemma 2 holds too.

$$\textsc{alpha:}\ \frac{P \equiv_\alpha P' \qquad P' \xrightarrow{\alpha} P''}{P \xrightarrow{\alpha} P''} \qquad \textsc{pre:}\ \alpha.P \xrightarrow{\alpha} P$$

$$\textsc{par:}\ \frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q}\ \text{if } \mathrm{bn}(\alpha) \cap \mathrm{fn}(Q) = \emptyset \qquad \textsc{res:}\ \frac{P \xrightarrow{\alpha} P'}{\boldsymbol{\nu}\, x\, P \xrightarrow{\alpha} \boldsymbol{\nu}\, x\, P'}\ \text{if } x \notin \mathrm{n}(\alpha)$$

$$\textsc{com:}\ \frac{P \xrightarrow{\alpha} P' \quad Q \xrightarrow{\overline{\alpha}} Q'}{P \mid Q \xrightarrow{\tau} \boldsymbol{\nu}\, x\, (P' \mid Q')}\ \text{for } \alpha \neq \tau,\ x = \mathrm{bn}(\alpha) \qquad \textsc{sum:}\ \frac{P_i \xrightarrow{\alpha} P_i',\ i \in I}{\sum_{i \in I} P_i \xrightarrow{\alpha} P_i'}$$

Table 1. The transition system for πI

Definition 3 (πI strong bisimilarity). *Strong bisimilarity* is the largest symmetric relation $\sim$ on πI processes s.t. $P \sim Q$ and $P \xrightarrow{\alpha} P'$, with $\mathrm{bn}(\alpha) \cap \mathrm{fn}(Q) = \emptyset$, imply that there is Q' s.t. $Q \xrightarrow{\alpha} Q'$ and $P' \sim Q'$.

By contrast with πI bisimilarity, in π-calculus bisimilarity [MPW92] the clauses for input and output must be distinguished, the reason being that input and output are not symmetric.

Lemmas 4 and 5 are technical results useful to deal with the alpha convertibility clause on processes and transitions. Lemma 5 shows that bisimilarity is preserved by injective substitutions on names.

Lemma 4. *If $P \equiv_\alpha Q$, then $P \sim Q$.* □

Lemma 5. *If $y \notin fn(P)$, then for all x, $P \sim Q$ implies $P\{y/x\} \sim Q\{y/x\}$.* □

Proposition 6 (congruence for $\sim$). *Strong bisimilarity is a congruence.*

PROOF: By showing that it is preserved by all operators of the language. Each case is simple. For instance, for prefixes, one shows that $\{(\alpha.P, \alpha.Q)\} \cup \sim$ is a strong bisimulation. The move $\alpha.P \xrightarrow{\alpha} P$ is matched by $\alpha.Q \xrightarrow{\alpha} Q$; this is enough even if α is an input prefix, since no instantiation of the bound name is required. □

Weak transitions and weak bisimilarity, written $\approx$, are defined in the expected way. As strong bisimilarity, so weak bisimilarity is preserved by all operators of the language.[1]

3.2. Axiomatisation We show a sound and complete axiomatisation for strong bisimilarity over finite πI processes.

To have more readable axioms, it is convenient to decompose sums $\sum_{i \in I} \alpha_i.P_i$ into binary sums. Thus we assume that sums are generated by the grammar

$$M := M + N \mid \alpha.P \mid \mathbf{0}.$$

[1] The congruence is not broken by sum because of the guarded form of our sums.

We let M, N, L range over such terms. The axiom system is reported in Table 2; we call it $\mathcal{A}$. We write $\mathcal{A} \vdash P = Q$ if $P = Q$ can be inferred from the axioms in $\mathcal{A}$ using equational reasoning.

Alpha-conversion:	If P and Q alpha-convertible then $P = Q$	
Summation:	$M + \mathbf{0} = M$	$M + N = N + M$
	$M + (N + L) = (M + N) + L$	$M + M = M$
Restriction:	if, for all $i \in I$, $x \notin \mathrm{n}(\alpha_i)$ then $\boldsymbol{\nu}\, x\, (\sum_i \alpha_i . P_i) = \sum_i \alpha_i . \boldsymbol{\nu}\, x\, P_i$	
	if x is the subject of α then $\boldsymbol{\nu}\, x\, (M + \alpha . P) = \boldsymbol{\nu}\, x\, M$	

Expansion:

Assume that $P = \sum_i \alpha_i . P_i$ and $Q = \sum_j \beta_j . Q_j$, and that for all i and j with $\alpha_i, \beta_j \neq \tau$, it holds that $\mathrm{bn}(\alpha_i) = \mathrm{bn}(\beta_j) = x \notin \mathrm{fn}(P, Q)$. Then infer

$$P \mid Q = \sum_i \alpha_i . (P_i \mid Q) + \sum_j \beta_j . (P \mid Q_j) + \sum_{\alpha_i \; opp \; \beta_j} \tau . \boldsymbol{\nu}\, x\, (P_i \mid Q_j)$$

where $\alpha_i \; opp \; \beta_j$ holds if $\alpha_i = \overline{\beta_j}$.

Table 2. The axiom system for finite πI processes

Theorem 7 (soundness and completeness). *$P \sim Q$ iff $\mathcal{A} \vdash P = Q$.* □

Omitting the axiom for alpha conversion and the bound name x in the expansion scheme, the axioms of Table 2 form a standard axiom system for strong bisimilarity of CCS. Also the proofs of soundness and completeness for the πI axiomatisation are very similar to those for CCS [Mil89]. For instance, as in CCS, so in the completeness proof for πI a restriction can be pushed down into the tree structure of a process until either a $\mathbf{0}$ process is reached, or a $\mathbf{0}$ process is introduced by cutting branches of the tree, and then the restriction disappears.

The proof of completeness of the axiomatisation [San95] exploits a transformation of processes to normal forms, that is tree-like structures built from the operators of sum, prefixing and $\mathbf{0}$. Then the axioms for commutativity, associativity and idempotence of sum, and alpha conversion can be used to obtain canonical and minimal representatives for the equivalence classes of $\sim$. Again, this mimics a well-known procedure for CCS.

4 Extending the signature of the finite and monadic πI

4.1. Infinite processes To express processes with an infinite behaviour, we add recursive agent definitions to the language of finite πI processes. We assume

a set of constants, ranged over by D, each of which has a non-negative *arity*, and add the production

$$P ::= D\langle\widetilde{x}\rangle$$

to the grammar of Definition 1. It is assumed that each constant D has a unique defining equation of the form $D \stackrel{\text{def}}{=} (\widetilde{x})\, P$. Both in a constant definition $D \stackrel{\text{def}}{=} (\widetilde{x})\, P$ and in a constant application $D\langle\widetilde{x}\rangle$, the parameter $\widetilde{x}$ is a tuple of all distinct names whose length equals the arity of D.

The constraint that the actual parameters $\widetilde{x}$ in a constant application should be distinct — normally not required in the π-calculus — ensures that alpha conversion remains the only relevant form of name substitution in πI. The transition rule for constants is:

$$\frac{P \xrightarrow{\alpha} P'}{D\langle\widetilde{x}\rangle \xrightarrow{\alpha} P'} \text{if } D \stackrel{\text{def}}{=} (\widetilde{y})\, Q \text{ and } (\widetilde{y})\, Q \equiv_\alpha (\widetilde{x})\, P\,.$$

4.2. Polyadicity The calculi seen so far are *monadic*, in that precisely *one* name is exchanged in any communication. We extend these calculi with polyadic communications following existing polyadic formulations of the π-calculus [Mil91]. The operational semantics and the algebraic theory of the polyadic πI are straightforward generalisations of those of the monadic πI, and will be omitted.

The syntax of the polyadic πI only differs from that of the monadic calculus because the object part of prefixes is a tuple of names:

$$\alpha ::= \tau \;\mid\; x(\widetilde{y}) \;\mid\; \overline{x}(\widetilde{y})\,.$$

Names in $\widetilde{y}$ are all pairwise different. When $\widetilde{y}$ is empty, we omit the surrounding parenthesis.

As in the π-calculus [Mil91, section 3.1], so in πI the move to polyadicity does not increase expressiveness: A polyadic interaction can be simulated using monadic interactions and auxiliary fresh names.

4.3. Typing Having polyadicity, one needs to impose some discipline on names so to avoid run-time arity mismatchings in interactions, as for $x(y).\, P \mid \overline{x}(y, z).\, Q$. In the π-calculus, this discipline is achieved by means of a *typing system* (in the literature it is sometimes called *sorting system*; in this paper we shall prefer the word "type" to "sort"). A typing allows us to specify the arity of a name and, recursively, of the names carried by that name. The same formal systems can be used for the typing of πI. (However, the typed πI enjoys a few properties which are not true in the typed π-calculus; one such property is that the *by-structure* and *by-name* definitons of equality between types [PS93] coincide.) We shall not present the type system here; an extensive treatment is in [San95].

4.4. Recursion versus replication Some presentations of the π-calculus have the replication operator in place of recursion. A replication $!P$ stands for an infinite number of copies of P in parallel, and is easily definable in terms of recursion. The typing system of πI, as well as that of the π-calculus, allows recursive types. However, if in πI recursion is replaced by replication, then all

processes can be typed without the use of recursive types. Starting from this observation, in [San95] we show that recursion cannot be encoded in terms of replication. This contrasts with the π-calculus, where recursion and replication are interdefinable [Mil91].

5 The encoding of the λ-calculus

πI appears to have considerable expressiveness. We have examined various non-trivial applications, which include the encodings of values and data structures, the λ-calculus, agent-passing calculi, and the locality and causality relations among actions of processes. Values and data structures can be modelled in πI in the same way as they are in π-calculus: The π-calculus representations given by Milner [Mil91, Sections 3.3., 6.2 and 6.3] only utilise the πI operators. Also, the encodings of locality and causality into π-calculus in (see [BS95]) can be easily adapted to πI. More interesting is the encoding of the λ-calculus and of agent-passing calculi. We look at the λ-calculus here, and sketch the study of agent-passing calculi in Section 6.

In this section, $M, N, \ldots$ are λ-calculus terms, whose syntax is given by

$$M := x \mid \lambda x.M \mid MM$$

where x and y range over λ-calculus variables. In Abramsky's *lazy lambda calculus* [Abr89], the redex is always at the extreme left of a term. There are two reduction rules:

$$\texttt{beta:}\quad (\lambda x.M)N \Longrightarrow M\{N/x\}\,, \qquad\qquad \texttt{app-L:}\quad \frac{M \Longrightarrow M'}{MN \Longrightarrow M'N}\,.$$

We first encode the *linear* lazy λ-calculus, in which no subterm of a term may contain more than one occurrence of x, for any variable x. We begin by recalling Milner's encoding $\mathcal{C}$ into the π-calculus. Then we describe the changings to be made to obtain an encoding $\mathcal{P}$ into πI. Both encodings are presented in Table 3. The core of any encoding of the λ-calculus into a process calculus is the translation of function application. This normally becomes a particular form of parallel combination of two agents, the function and its argument; beta-reduction is then modeled as process reduction.

Let us examine $\mathcal{C}$. In the pure λ-calculus, every term denotes a function. When supplied with an argument, it yields another function. Analogously, the translation of a λ-term M is a process with a location p. It rests dormant until it receives along p two names: The first is a trigger x for its argument and the second is the location to be used for the next interaction. The location of a term M is the unique port along which M interacts with its environment. Two types of names are used in the encoding: *Location names*, ranged over by p, q and r, and *trigger names*, ranger over by x, y and z. For simplicity, we have assumed that the set of trigger names is the same as the set of λ-variables. More details on this encoding and a study of its correctness can be found in [Mil91, San92].

The encoding into π-calculus; $\overline{r}(x)p.-$ is an output prefix at r in which the private name x and the free name p are emitted.

$$\begin{aligned}
\mathcal{C}[\![\lambda x.M]\!]_p &\stackrel{\text{def}}{=} p(x,q).\mathcal{C}[\![M]\!]_q \\
\mathcal{C}[\![x]\!]_p &\stackrel{\text{def}}{=} \overline{x}p \\
\mathcal{C}[\![MN]\!]_p &\stackrel{\text{def}}{=} \boldsymbol{\nu}\, r\,(\mathcal{C}[\![M]\!]_r \mid \overline{r}(x)p.\, x(q).\mathcal{C}[\![N]\!]_q) \qquad x \text{ fresh}
\end{aligned}$$

The encoding into πI:

$$\begin{aligned}
\mathcal{P}[\![\lambda x.M]\!]_p &\stackrel{\text{def}}{=} \overline{p}(w).\, w(x,q).\mathcal{P}[\![M]\!]_q \\
\mathcal{P}[\![x]\!]_p &\stackrel{\text{def}}{=} \overline{x}(r).\, r \rightarrow p \\
\mathcal{P}[\![MN]\!]_p &\stackrel{\text{def}}{=} \boldsymbol{\nu}\, r\,(\mathcal{P}[\![M]\!]_r \mid r(w).\,\overline{w}(x,q').\,(q' \rightarrow p \mid x(q).\mathcal{P}[\![N]\!]_q) \qquad x \text{ fresh}
\end{aligned}$$

Table 3. The encodings of the linear lazy λ-calculus

Encoding $\mathcal{C}$ is not an encoding into πI because there are outputs of free names, one in the rule for variables, and one in the rule for applications. Indeed, the free output construct plays an important role in $\mathcal{C}$: It is used to redirect location names which, in this way, can bounce an unbounded number of times before arresting as subject of a prefix.

Encoding $\mathcal{P}$ is obtained from $\mathcal{C}$ with two modifications. First, the output of a free name b is replaced by the output of a bound name c plus a *link* from c to b, written $c \rightarrow b$. Names b and c are "connected" by the link, in the sense that a process performing an output at c and a process performing an input at b can interact, asynchronously, through the link. In other words, a link behaves a little like a name buffer: It receives names at one end-point and transmit names at the other end-point. However, the latter names are not the same as the former names — as it would be in a real buffer — but, instead, are *linked* to them: This accounts for the recursion in the definition of links below. For tuples of names $\widetilde{u} = u_1, \ldots, u_n$ and $\widetilde{v} = v_1, \ldots, v_n$ we write $\widetilde{u} \rightarrow \widetilde{v}$ to abbreviate $u_1 \rightarrow v_1 \mid \ldots \mid u_n \rightarrow v_n$.

If a and b have the same type, then we define: $\quad a \rightarrow b \stackrel{\text{def}}{=} a(\widetilde{u}).\,\overline{b}(\widetilde{v}).\,\widetilde{v} \rightarrow \widetilde{u}$

(for convenience, we have left the parameters a and b of the link on the left-hand side of the definition). Note that the link is ephemeral for a and b — they can only be used once — and that it inverts its direction at each cycle — the recursive call creates links from the objects of b to the objects of a. Both these features are tailored to the specific application in exam, namely the encoding of the lazy λ-calculus.

The other difference between encodings $\mathcal{C}$ and $\mathcal{P}$ is that the latter has a level of indirection in the rule for abstraction. A term signals to be an abstraction before receiving the actual arguments. This is implemented using a new type

of names, ranged over by w. This modification could be avoided using more sophisticated links, but they would complicate the proofs in Lemma 8.

When reasoning about encoding $\mathcal{P}$, one does not have to remember the definition of links; the algebraic properties of links in Lemma 8 are enough. Assertion (1) of this lemma shows that two links with a common hidden end-point behave like a single link; assertions (2) and (3) show that a link with a hidden end-point acts as a substitution on the encoding of a λ-term.

Lemma 8. *Let M be a linear λ-term.*

1. *If a, b and c are distinct names of the same type, then $\nu\, b\,(a \to b \mid b \to c) \approx a \to c$.*
2. *If x and y are distinct trigger names and y is not free in M, then $\nu\, x\,(x \to y \mid \mathcal{P}[\![M]\!]_r) \approx \mathcal{P}[\![M\{y/x\}]\!]_p$.*
3. *If p and r are distinct location names, then $\nu\, r\,(r \to p \mid \mathcal{P}[\![M]\!]_r) \approx \mathcal{P}[\![M]\!]_p$.*

The main result needed to convince ourselves of the correctness of $\mathcal{P}$ is the validity of beta-reduction. The proof is conceptually the same as the proof of validity of beta-reduction for Milner's encoding into π-calculus; in addition, one has to use Lemma 8(3).

Theorem 9. *For all M, N, p it holds that $\mathcal{P}[\![(\lambda.xM)N]\!]_p \approx \mathcal{P}[\![M\{N/x\}]\!]_p$.* □

To encode the full lazy λ-calculus, where a variable may occur free in a term more than once, the argument of an application must be made persistent. This is achieved by adding, in both encodings $\mathcal{C}$ and $\mathcal{P}$, a replication in front of the prefix $x(q).-$, in the rule for application (recall that replication is a derived operator in a calculus with recursion). In addition, for $\mathcal{P}$ also the link for trigger names must be made persistent, so that it can serve the possible multiple occurrences of a trigger in a term. Thus

if x and y are trigger names, then we define: $\quad x \to y \stackrel{\text{def}}{=} !\, x(\widetilde{u}).\, \overline{y}(\widetilde{v}).\, \widetilde{v} \to \widetilde{u}\,.$

In this way, Lemma 8 and Theorem 9 remain true for the full lazy λ-calculus.

Links — as defined here, or variants of them — can be used to increase the parallelism of processes. For instance, adding links in the encoding of λ-abstractions, as below, gives an encoding of a *strong lazy* strategy, where reductions can also occur underneath an abstraction (i.e., the `Xi` rule, saying that if $M \longrightarrow M'$ then $\lambda x.\, M \longrightarrow \lambda x.\, M'$, is now allowed):

$$\mathcal{P}[\![\lambda x.M]\!]_p \stackrel{\text{def}}{=} \nu\, q, x\,\big(\overline{p}(w).\, w(y, r).\, (q \to r \mid x \to y) \mid \mathcal{P}[\![M]\!]_q\big)\,.$$

In the lazy λ-calculus encoding, there is a rigid sequentialisation between the behaviour of (the encodings of) the head $\lambda x.-$ and of the body M of the abstraction: The latter cannot do anything until the former has supplied it with its arguments x and q. In the strong-lazy encoding, the *only* dependencies of the body from the head are given by the actions in which these arguments appear; any other activity of the body can proceed independently from the activity of the head.

6 Relationship with agent-passing process calculi

We have used πI and subcalculi of it, to study the expressiveness of *agent-passing process calculi* (they are sometimes called *higher-order* process calculi in the literature). In these calculi, agents, i.e., terms of the language, can be passed as values in communications. The agent-passing paradigm is often presented in opposition to the *name-passing* paradigm, followed by π-calculus and related calculi, where mobility is modelled using communication of names, rather than of agents. An important criterion for assessing the value of the two paradigms is the expressiveness which can be achieved. Below, we briefly summarise work reported in [San95].

6.1. A hierarchy of πI subcalculi Using the typing system of πI and imposing constraints on it, we have defined a hierarchy of calculi $\{\pi\mathrm{I}^n\}_{n\leq\omega}$. A calculus $\pi\mathrm{I}^n$ includes those πI processes which can be typed using types of order n or less than n, and $\pi\mathrm{I}^\omega$ is the union of the $\pi\mathrm{I}^n$'s. Instead of giving the formal definition of the order of a type, we explain — very informally — what syntactic constraints the orders of types impose on processes. Take a process in which the bound names are all distinct from each other and from the free names; we say that a name of this process *depends on* another name if the latter carries the former. For instance, in process $x(y).\overline{y}(z).z.\mathbf{0}$, name y depends on x and z depends on y. A *dependency chain* is a sequence $x_1,\ldots,x_n$ of names s.t. x_{i+1} depends on x_i, for all $1 < i \leq n$. Thus the processes in $\pi\mathrm{I}^n$ are those which have *dependency chains* among names of length at most n. For instance, process $x(y).\overline{y}(z).z.\mathbf{0}$ is in $\pi\mathrm{I}^n$, for all $n \geq 3$, since its maximal dependency chain has length 3, involving names x, y and z. Calculus $\pi\mathrm{I}^1$ includes processes like $a.b|\overline{a}.\overline{b}$ in which names are only used for pure synchronisation; $\pi\mathrm{I}^1$ represents the core of CCS. $\pi\mathrm{I}^2$ includes processes like

$$x(y,z).(\overline{y} \mid z) \qquad \text{and} \qquad y.x(z).\overline{y}.z$$

where if a name carries another name, then the latter can only be used for pure synchronisation. A technical remark: The syntax of the calculi $\{\pi\mathrm{I}^n\}_{n\leq\omega}$, in [San95], uses the replication operator in place of recursion; this makes sense because, as mentioned at the end of Section 4, the typability of the processes which use replication does not require recursive types (i.e, all types have a bounded order).

Intuitively, the calculi $\pi\mathrm{I}^1, \pi\mathrm{I}^2, \ldots, \pi\mathrm{I}^n, \ldots \pi\mathrm{I}^\omega, \pi\mathrm{I}$ are distinguished by the "degree" of mobility allowed; indeed, if mobility is taken into account, then they form a hierarchy of calculi of strictly increasing expressiveness.

6.2. Correspondence with a hierarchy of agent-passing process calculi Agent-passing developments of CCS are the calculi *Plain CHOCS* [Tho90], and *Strictly-Higher-Order π-calculus*; the latter, abbreviated HOπ^ω, is the fragment of the Higher-Order π-calculus [San92] which is purely higher order, i.e., no name-passing feature is present. In Plain CHOCS processes only can be exchanged. In HOπ^ω besides processes also abstractions, i.e., functions from agents

to agents, of arbitrary high order can be exchanged. Roughly, HOπ^ω is as an extension of CCS with the constructs of the simply-typed λ-calculus, namely variable X, abstraction $(X)\,A$ (it would be written $\lambda X.\,A$ in a λ-calculus notation) and application $A\langle B\rangle$ (where B is the argument of the application). An example of abstraction is $F \stackrel{\text{def}}{=} (X)\,(P \mid X)$: It represents a function from processes to processes, where the process-argument is run in parallel with P in the process-result; for instance, $F\langle Q\rangle$ beta-reduces to $P|Q$. An abstraction one order higher than F is $G \stackrel{\text{def}}{=} (Y)\,(P \mid Y\langle Q\rangle)$, which takes abstractions of the same type as F as argument. The application $G\langle F\rangle$ beta-reduces to $P \mid P \mid Q$. The types used in HOπ^ω are those of the simply-typed λ-calculus with the process type $\bullet$ as the only first-order (i.e., basic) type. For instance, the abstractions F and G above have types $\bullet \longrightarrow \bullet$ and $(\bullet \longrightarrow \bullet) \longrightarrow \bullet$, respectively. The order of a type is determined by the level of arrow-nesting in its definition.

As in πI, so in HOπ^ω we can discriminate processes according to order of the types needed in the typing. This yields a hierarchy of agent-passing calculi $\{\text{HO}\pi^n\}_{n<\omega}$, where HO$\pi^1$ coincides with πI^1 — hence with the core of CCS — and HOπ^2 is the core of Plain CHOCS. For each $n \leq \omega$, we have compared the agent-passing calculus HOπ^n with the name-passing calculus πI^{n-}; the latter is a subcalculus of πI^n whose processes respect a discipline on the input and output usage of names similar to those studied in [PS93]. We have showed that HOπ^n and πI^{n-} have the same expressiveness, by exhibiting faithful encodings of HOπ^n into πI^{n-} and of πI^{n-} into HOπ^n. The encodings are fully abstract w.r.t. the reduction relations of the two calculi. (The encoding from HOπ^n to πI^{n-} is a special case of the compilation of the full Higher-Order π-calculus into π-calculus studied in [San92]; the communication of an agent is translated as the communication of a private name with which the recipient can activate a copy of the agent.) Note in particular the correspondence between HOπ^2 and πI^{2-}: Process passing only gives little expressiveness more than CCS.

These results establish an exact connection between agent-passing calculi and name-passing calculi based on internal mobility, and strengthen the relevance of the latter calculi.

7 Future work

We wish to develop the study of the expressiveness of πI, which we expect to be rather close to that of the π-calculus. The translation of the λ-calculus presented is obtained by refining Milner's encoding into the π-calculus, which makes non-trivial use of the free-output construct — disallowed in πI. We hope that the encoding might also give insights into the comparison between πI and π-calculus.

For the translation of the λ-calculus, we first adopted Abramsky's *lazy* reduction strategy. Our encoding of it uses special πI processes called *links*. We believe that understanding the algebraic properties of links can be helpful to justify transformations of processes aimed at augmenting their parallelism. For instance, in Section 5 by manipulating links we have modified the encoding of

the lazy strategy into an encoding of a *strong-lazy* strategy which is more permissive (i.e., more parallel) because it also allows reductions inside abstractions (the `Xi` rule). At present we are studying the properties of this encoding. We are not aware of other encodings, into a process algebra, of λ-calculus strategies encompassing the `Xi` rule.

We have showed that name-passing process calculi based on internal mobility have a simple algebraic theory, in which the main difference from the theory of CCS is the use of alpha conversion. These calculi also possess a pleasant symmetry in their communication constructs. These features might become useful in the development of denotational models.

References

[Abr89] S. Abramsky. The lazy lambda calculus. In D. Turner, editor, *Research Topics in Functional Programming*, pages 65–116. Addison-Wesley, 1989.

[BK85] J.A. Bergstra and J.W. Klop. Algebra for communicating processes with abstraction. *Theoretical Computer Science*, 37(1):77–121, 1985.

[BS95] M. Boreale and D. Sangiorgi. A fully abstract semantics for causality in the π-calculus. *Proc. STACS'95*, To appear.

[DKV91] P. Degano, S. Kasangian, and S. Vigna. Applications of the calculus of trees to process description languages. *Proc. CTCS '91*, LNCS 530, 1991.

[FMQ94] G. Ferrari, U. Montanari, and P. Quaglia. A π-calculus with explicit substitutions: the late semantics. *Proc. MFCS'94*, LNCS, 1994.

[Jon93] C.B. Jones. A π-calculus semantics for an object-based design notation. In E. Best, editor, *Proceedings of CONCUR '93*, LNCS 715, 1993.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[Mil91] R. Milner. The polyadic π-calculus: a tutorial. Technical Report ECS–LFCS–91–180, LFCS, Dept. of Comp. Sci., Edinburgh Univ., October 1991.

[Mil92] R. Milner. Functions as processes. *Journal of Mathematical Structures in Computer Science*, 2(2):119–141, 1992.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, (Parts I and II). *Information and Computation*, 100:1–77, 1992.

[PS93] B. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. In *8th LICS Conf.*, pages 376–385. IEEE Computer Society Press, 1993.

[San92] D. Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms*. PhD thesis CST–99–93, Department of Computer Science, University of Edinburgh, 1992.

[San95] D. Sangiorgi. Full version of this paper. To Appear as Technical Report, INRIA-Sophia Antipolis, 1995.

[Tho90] B. Thomsen. *Calculi for Higher Order Communicating Systems*. PhD thesis, Department of Computing, Imperial College, 1990.

[Wal95] D. Walker. Objects in the π-calculus. *Information and Computation*, 1995. To appear.