π-calculus, internal mobility, and agent-passing calculi[*]

Davide Sangiorgi
INRIA- Sophia Antipolis,
2004 Rue des Lucioles, B.P. 93, 06902 Sophia Antipolis, France.

Abstract

The π-calculus is a process algebra which originates from CCS and permits a natural modelling of mobility (i.e., dynamic reconfigurations of the process linkage) using communication of names. Previous research has shown that the π-calculus has much greater expressiveness than CCS, but also a much more complex mathematical theory. The primary goal of this work is to understand the reasons of this gap. Another goal is to compare the expressiveness of *name-passing* calculi, i.e., calculi like π-calculus where mobility is achieved via exchange of names, and that of *agent-passing calculi*, i.e., calculi where mobility is achieved via exchange of agents.

We separate the mobility mechanisms of the π-calculus into two, respectively called *internal mobility* and *external mobility*. The study of the subcalculus which only uses internal mobility, called πI, suggests that internal mobility is responsible for much of the expressiveness of the π-calculus, whereas external mobility is responsible for many of the semantic complications. A pleasant property of πI is the full symmetry between input and output constructs.

Internal mobility is strongly related to agent-passing mobility. By imposing bounds on the order of the types of πI and of the Higher-Order π-calculus [San92] we define a hierarchy of name-passing calculi based on internal mobility and one of agent-passing calculi. We show that there is an exact correspondence, in terms of expressiveness, between the two hierarchies.

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. Although there is no theorem to formally support this statement, the evidence is compelling. For instance, in the π-calculus we can encode:

- *Data values* [MPW92, Mil91];
- *agent-passing* process calculi [Tho90, San92, Ama93];

---

[*] Extracts of parts of the material contained in this paper can be found in the Proceedings of TAPSOFT'95 and ICALP'95. This research has been supported by the Esprit BRA project 6454 "CONFER".

- 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* [San95b, BS94].

The encodings are simple and intuitive and, notably, their correctness is supported by full abstraction results. 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, can we describe the dramatic jump from CCS to π-calculus as a sequence of smaller jumps? 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.

Using the typing system of πI, as inherited from the π-calculus, and imposing some constraint on it, we define the 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. Informally speaking, 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 the mobility exhibited is taken into account, then they can be proved to form a hierarchy of calculi of strictly increasing expressiveness. $\pi\mathrm{I}^1$ does not allow mobility at all and is the core of CCS. This hierarchy gives us a classification of mobility and an incremental view of the transition from CCS to π-calculus. (A more comprehensive discussion on external, internal and bounded mobility is deferred to Sections 2 and 6.)

We shall use the above hierarchy also to understand 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 the π-calculus and related calculi, where mobility is modelled using communication of names rather than communication of agents. An important criterion for assessing the value of the two paradigms is the expressiveness which can be achieved. Agent-passing developments of CCS are the calculi *Plain CHOCS* [Tho90], and *Strictly-Higher-Order π-calculus*; the latter, abbreviated $\mathrm{HO}\pi^\omega$, 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 $\mathrm{HO}\pi^\omega$ besides processes also abstractions (i.e., functions from agents to agents) of arbitrary high order can be exchanged. Roughly, $\mathrm{HO}\pi^\omega$ is as an extension of CCS with the constructs of the simply-typed λ-calculus. As in πI, so in $\mathrm{HO}\pi^\omega$ we can discriminate processes according to the order of the types needed in the typing. This yields a hierarchy of agent-passing calculi $\{\mathrm{HO}\pi^n\}_{n<\omega}$, where $\mathrm{HO}\pi^1$ coincides with $\pi\mathrm{I}^1$ — hence with the core of CCS — and $\mathrm{HO}\pi^2$ is the core of Plain CHOCS. For each $n \leq \omega$, we compare the agent-passing calculus $\mathrm{HO}\pi^n$ with the name-passing calculus $\pi\mathrm{I}^{n-}$; the latter is a subcalculus of $\pi\mathrm{I}^n$ whose processes respect a discipline on the input and output usage of names similar to those studied in [PS93]. We show that $\mathrm{HO}\pi^n$ and $\pi\mathrm{I}^{n-}$ have the same expressiveness, by exhibiting faithful encodings of $\mathrm{HO}\pi^n$ into $\pi\mathrm{I}^{n-}$ and of $\pi\mathrm{I}^{n-}$ into $\mathrm{HO}\pi^n$. The encodings are fully abstract w.r.t. the reduction relations of the two calculi.

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.

We introduce the finite part of πI in Section 2 and we study its theory in Section 3. In Section 4 we consider extensions of the signature of the finite πI, intended to capture infinite behaviours and polyadicity. As in the π-calculus, so in πI the extension to polyadicity is smooth. However, the typing system of πI enjoys a few properties not true in the π-calculus; for instance, in πI the by-name and by-structure definitions of equality between types coincide. To have infinite behaviours in πI, we use recursive agent definitions. We also consider the replication operator; we define $\pi\mathrm{I}^\omega$ as the calculus with replication in place of recursion. The typability of processes in πI requires recursive types; that of processes in $\pi\mathrm{I}^\omega$ does not. In Section 5 we examine the encoding of the lambda calculus into πI. 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; hence the encoding gives us some indications about how the effect of free outputs might be achieved in πI. As reduction strategy for lambda terms we chose the *lazy* one [Abr89]. We obtain an encoding into πI as a refinement of Milner's encoding into π-calculus; we show that a further similar refinement leads to an encoding of a more parallel reduction strategy, which allows reductions inside abstractions. We argue that the λ-calculus cannot be encoded into $\pi\mathrm{I}^{\omega}$. In Section 6 we define the calculi $\{\pi\mathrm{I}^{n}\}_{n<\omega}$; we then study the expressiveness of the name-passing calculi introduced. In Section 7 we present the agent-passing calculus $\mathrm{HO}\pi^{\omega}$ and its type system, and we define the calculi $\{\mathrm{HO}\pi^{n}\}_{n<\omega}$. In Section 8 we compare the expressiveness of the agent-passing calculi with that of the name-passing calculi. In Section 9 we report some conclusions and possible directions for future work.

Related work. We are not aware of other work on isolating or classifying different forms of mobility for name-passing calculi.

Encodings of agent-passing calculi into a name-passing calculus have been studied by Thomsen [Tho90], Sangiorgi [San92] and Amadio [Ama93]. Thomsen and Amadio deal with Plain CHOCS and π-calculus; Sangiorgi with Higher-Order π-calculus and π-calculus. The encoding from $\mathrm{HO}\pi^{n}$ to $\pi\mathrm{I}^{n}$ used in this paper is a special case of the encoding in [San92] and, when restricted to $\mathrm{HO}\pi^{2}$, it is the same as the encodings in [Tho90] and [Ama93]. The works [Tho90, San92, Ama93] show that agent-passing can be mimicked using name-passing. In this paper, we push the analysis further, in that: (1) we isolate the specific features of name-passing calculi which make the encodings possible, and (2) we investigate the opposite direction, namely the modelling of name-passing using agent-passing.

The only attempt that we know at encoding a name-passing calculus into an agent-passing calculus is by Thomsen [Tho90], who gives a translation of the π-calculus into Plain CHOCS. However, the translation makes heavy use of a relabelling operator of Plain CHOCS which behaves as a *dynamic binder* — occurrences of names not bound can later become bound. Since we only accept *static binders*, our translation of $\pi\mathrm{I}^{n-}$ into $\mathrm{HO}\pi^{n}$ is quite different from Thomsen's. The absence of relabeling is indeed what distinguishes $\mathrm{HO}\pi^{2}$ from Plain CHOCS.

Important studies of higher-order calculi have been conducted by Astesiano et al. [AGR92], in the framework of generalised algebraic specifications, and by Hennessy [Hen93], who has considered the model theory. However, in their languages the restriction operator, when present, is not a static binder — a significant difference w.r.t. the languages treated in this paper.

2 πI: A Symmetric calculus based on internal mobility

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 no compelling reasons for wanting symmetry: Our major motivation is elegance, which will show up in the presentation of the calculus and of its properties.

Notation: If $\mathcal{R}$ and $\mathcal{S}$ are relation, then $\mathcal{R}\ \mathcal{S}$ is their composition (i.e., $(a,c) \in\mathcal{R}\ \mathcal{S}$ if there is b s.t. $(a,b) \in \mathcal{R}$ and $(b,c) \in \mathcal{S}$). 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. Symbols $x, y, z, \ldots$ will range over the infinite set of names; P, Q and R will be metavariables over processes:

$$\begin{array}{rcl} P & ::= & \sum_{i \in I} \alpha_i .\, P_i \;\Big|\; P \mid P \;\Big|\; \boldsymbol{\nu}\, x\, P \\ \alpha & ::= & \tau \;\Big|\; x(y) \;\Big|\; \overline{x}y \end{array}$$

The guarded-sum construct $\sum_{i \in I} \alpha_i .\, P_i$ is used to make a choice among the summands $\alpha_i .\, P_i$: The first process $\alpha_i .\, P_i$ that succeeds at performing the action α_i continues, whereas the other summands are discarded. 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 $\sum_{1 \leq i \leq n} \alpha_i .\, P_i$. Parallel composition is to run two processes in parallel; restriction $\boldsymbol{\nu}$ makes name x in $\boldsymbol{\nu}\, x\, P$ local, i.e., private, to P. Prefixes, ranged over by α, can be of the form τ (*silent prefix*), $x(y)$ (*input prefix*), or $\overline{x}y$ (*output prefix*). Symbol τ represents internal activity: $\tau .\, P$ can evolve to P without interacting with other processes. A process $x(y) .\, P$ can perform an input at x, and y is the placeholder for the name so received. Process $\overline{x}y .\, P$ can perform the output at x of y, and then continues as P.

An input prefix $x(y) .\, P$ and a restriction $\boldsymbol{\nu}\, y\, P$ 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 a visible prefix, the first name is the *subject*, and the second name is the *object*. 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 $\boldsymbol{\nu}\, x_1 \ldots \boldsymbol{\nu}\, x_n\, P$ as $\boldsymbol{\nu}\, x_1, \ldots, x_n\, P$. 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 they are very useful in the algebraic theory. But they will not be needed in the algebraic theory of πI, 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 $\boldsymbol{\nu}\, y\, \overline{x}y$. 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:

$$\texttt{OPEN}: \; \frac{P \xrightarrow{\overline{x}y} P'}{\boldsymbol{\nu}\, y\, P \xrightarrow{\overline{x}(y)} P'} \quad x \neq y\,.$$

This rule says that if P can perform the output of the free name y at x, then $\boldsymbol{\nu}\, y\, P$ can perform the output of the private name y at x. (We can make an analogy between bound output and silent prefix: Both can be viewed as derived operators — $\tau . P$ as abbreviation for $\boldsymbol{\nu}\, x\, (x .\, P \mid \overline{x})$, 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:

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

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 2.1 (finite πI) *The class of finite πI processes is described by the following grammar:*

$$\begin{array}{rcl} P & ::= & \sum_{i\in I} \alpha_i.\,P_i \;\big|\; P \mid P \;\big|\; \boldsymbol{\nu}\, x\, P \\ \alpha & ::= & \tau \;\big|\; x(y) \;\big|\; \overline{x}(y)\,. \end{array}$$

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 3.1).

2.2 Internal and external mobility

In the previous section, the motivation to the introduction of πI was symmetry. A more pragmatic motivation is given in this section.

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:

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

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 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 many 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 the original form of bisimilarity in the π-calculus, 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 distinct: 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 the free names 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. (Note that, moreover, some form of matching among names is already given by parallel composition: The term $x.\,P \mid \overline{y}.\,Q$ can evolve to $P \mid Q$ only if x and y are equal.)

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* such a substitution should be made: 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 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 main 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 duality

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 $P \in \pi$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*.

ALPHA:	$\dfrac{P \equiv_\alpha P' \qquad P' \xrightarrow{\alpha} P''}{P \xrightarrow{\alpha} P''}$	PRE:	$\alpha.\, P \xrightarrow{\alpha} P$
PAR:	$\dfrac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q}\ \mathrm{bn}(\alpha) \cap \mathrm{fn}(Q) = \emptyset$	RES:	$\dfrac{P \xrightarrow{\alpha} P'}{\boldsymbol{\nu}\, x\, P \xrightarrow{\alpha} \boldsymbol{\nu}\, x\, P'}\ x \notin \mathrm{n}(\alpha)$
COM:	$\dfrac{P \xrightarrow{\alpha} P' \quad Q \xrightarrow{\overline{\alpha}} Q'}{P \mid Q \xrightarrow{\tau} \boldsymbol{\nu}\, x\, (P' \mid Q')}\ \alpha \neq \tau,\ x = \mathrm{bn}(\alpha)$	SUM:	$\dfrac{P_i \xrightarrow{\alpha} P'_i,\ i \in I}{\sum_{i \in I} P_i \xrightarrow{\alpha} P'_i}$

Table 1: The transition system for πI.

Lemma 3.1 *If $P \xrightarrow{\alpha} P'$, then $\overline{P} \xrightarrow{\overline{\alpha}} \overline{P'}$.* ☐

Note that since $\overline{\overline{P}} = P$, the converse of Lemma 3.1 holds too.

3.2 Strong and weak bisimilarity

Definition 3.2 (πI strong bisimilarity) *A symmetric relation $\mathcal{R}$ on πI processes is a* strong bisimulation *if $P \mathrel{\mathcal{R}} Q$ implies:*

- *whenever $P \xrightarrow{\alpha} P'$, with $bn(\alpha) \cap fn(Q) = \emptyset$, there is Q' s.t. $Q \xrightarrow{\alpha} Q'$ and $P' \mathrel{\mathcal{R}} Q'$.*

We say that two πI processes P and Q are strongly bisimilar*, written $P \sim Q$ if $P \mathrel{\mathcal{R}} Q$, for some strong bisimulation $\mathcal{R}$.*

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.

Remark 3.3 The side conditions on the freshness of names, in the transition system and the bisimilarity of πI, can be avoided by adopting a convention for bound names based on de-Bruijn indices. ☐

Lemmas 3.4–3.6 are technical results useful to deal with the alpha convertibility clause on processes and transitions. Lemma 3.5 shows that bisimilarity is preserved by injective substitutions on names.

Lemma 3.4 *If $P \equiv_\alpha Q$, then $P \sim Q$.* ☐

Lemma 3.5 *Suppose $y \notin fn(P)$. Then for all x, $P \sim Q$ implies $P\{y/x\} \sim Q\{y/x\}$.*

Proof: Similar to the analogous result for the π-calculus: One can show that if y is not free in a process R, then R and $R\{y/x\}$ can perform the same actions, up to substitution $\{y/x\}$ and alpha conversion. ☐

Lemma 3.6 *If $P \xrightarrow{x(y)} P'$ and $z \notin fn(P)$, then also $P \xrightarrow{x(z)} P''$, for some P'' with $P''\{y/z\} \equiv_\alpha P'$.* ☐

Remark 3.7 Lemmas 3.4-3.6 show that alpha conversion and injective substitutions are harmless. Hence when examining the derivatives of a process P (for instance, if we compare the behaviour of P with that of another process Q), it is safe to pick some fresh name x and force x to be the bound name of any action which appears in the derivation proof of a transition of P: Any other choice of bound names leads to the same derivative, up to alpha conversion and an injective substitution on names. $\square$

Definition 3.8 (bisimulation up to alpha conversion) *A symmetric relation* $\mathcal{R}$ *is a* bisimulation up to alpha conversion *if* $P \mathcal{R} Q$ *implies:*

- *whenever* $P \xrightarrow{\alpha} P'$, *with* $bn(\alpha) \cap fn(Q) = \emptyset$, *there is* Q' *s.t.* $Q \xrightarrow{\alpha} Q'$ *and* $P' \equiv_\alpha \mathcal{R} \equiv_\alpha Q'$.

Lemma 3.9 *If* $\mathcal{R}$ *is a bisimulation up to alpha conversion, then* $\mathcal{R} \subseteq \sim$. $\square$

Bisimilarity is preserved by all πI operators:

Proposition 3.10 (congruence for $\sim$) *If* $P \sim Q$, *then*

1. $R + \alpha. P \sim R + \alpha. Q$;
2. $\boldsymbol{\nu}\, x\, P \sim \boldsymbol{\nu}\, x\, Q$;
3. $P \mid R \sim Q \mid R$.

Proof: Each case is simple. For instance, for (1) one can show that

$$\{(R + \alpha. P, R + \alpha. Q)\} \cup \sim$$

is a strong bisimulation. The move $R + \alpha. P \xrightarrow{\alpha} P$ is matched by $R + \alpha. Q \xrightarrow{\alpha} Q$; this is enough even if α is an input prefix, since no instantiation of the bound name is required.
For (3), one can prove that the set of all pairs of the form $(\boldsymbol{\nu}\, \widetilde{x}\, (P \mid R), \boldsymbol{\nu}\, \widetilde{x}\, (Q \mid R))$, with $P \sim Q$, is a strong bisimulation up to alpha conversion. $\square$

Weak transitions and weak bisimilarity are defined in the expected way. Relation $\Longrightarrow$ is the reflexive and transitive closure of $\xrightarrow{\tau}$, and relation $\overset{\alpha}{\Longrightarrow}$ is $\Longrightarrow \xrightarrow{\alpha} \Longrightarrow$.

Definition 3.11 (πI weak bisimilarity) *A symmetric relation* $\mathcal{R}$ *on* π*I processes is a* weak bisimulation *if* $P \mathcal{R} Q$ *implies:*

- *whenever* $P \Longrightarrow P'$, *there is* Q' *s.t.* $Q \Longrightarrow Q'$ *and* $P' \mathcal{R} Q'$;
- *whenever* $P \overset{\alpha}{\Longrightarrow} P'$, *with* $\alpha \neq \tau$ *and* $bn(\alpha) \cap fn(Q) = \emptyset$, *there is* Q' *s.t.* $Q \overset{\alpha}{\Longrightarrow} Q'$ *and* $P' \mathcal{R} Q'$.

We say that two π*I processes* P *and* Q *are* weakly bisimilar, *written* $P \approx Q$, *if* $P \mathcal{R} Q$, *for some weak bisimulation* $\mathcal{R}$.

As strong bisimilarity, so weak bisimilarity is preserved by all operators of the language.[1]

Remark 3.12 πI is a subcalculus of the π-calculus: A πI process can also be interpreted as a π-calculus process and then the transition for a πI processes can be viewed as the transition of a π-calculus process. It follows that any bisimilarity result between πI processes which is valid in the π-calculus is also valid in πI. $\square$

---

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

Alpha-conv.	**A**	If P and Q alpha-convertible then	$P = Q$
Summation	**S1**		$M + \mathbf{0} = M$
	S2		$M + N = N + M$
	S3		$M + (N + L) = (M + N) + L$
	S4		$M + M = M$
Restriction	**R1**	If, $\forall i \in I$, $x \notin \mathrm{n}(\alpha_i)$ then	$\boldsymbol{\nu}\, x\, (\sum_{i \in I} \alpha_i .\, P_i) = \sum_{i \in I} \alpha_i .\, \boldsymbol{\nu}\, x\, P_i$
	R2	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 α_i *opp* β_j holds if $\alpha_i = \overline{\beta_j} \neq \tau$.

Table 2: The axiom system for finite πI processes

3.3 Axiomatisation

This section shows 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

$$M := M + N \;\big|\; \alpha .\, P \;\big|\; \mathbf{0}\,.$$

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. Note that a special case of **R1** (for $I = \emptyset$) is

$$\mathbf{R3} \quad \boldsymbol{\nu}\, x\, \mathbf{0} = \mathbf{0}\,.$$

Definition 3.13

- *A process P is in* head normal form, *briefly hnf, if P is of the form*

$$\sum_i \alpha_i .\, P_i\,.$$

- *The* depth *of P, written $d(P)$, is inductively defined as follows:*

$$\begin{array}{rcl} d(\mathbf{0}) & = & 0 \\ d(\alpha.\, P) & = & 1 + d(P) \\ d(\boldsymbol{\nu}\, x\, P) & = & d(P) \\ d(P_1 \mid P_2) & = & d(P_1) + d(P_2) \\ d(P_1 + P_2) & = & max\{d(P_1), d(P_2)\} \end{array}$$

Proposition 3.14 (soundness of $\mathcal{A}$) *If $\mathcal{A} \vdash P = Q$, then $P \sim Q$.* ☐

Lemma 3.15 *For any process P there is a hnf H with $d(H) \leq d(P)$ s.t. $\mathcal{A} \vdash P = H$.*

Proof: By induction on the structure of P. The transformations we consider do not increase the depth of a process. If $P = \mathbf{0}$ or $P = \alpha.\, P_1$, then P is already in hnf. If $P = \boldsymbol{\nu}\, x\, P_1$, then by induction, $\mathcal{A} \vdash P_1 = H_1$, for some hnf H_1; hence $\mathcal{A} \vdash \boldsymbol{\nu}\, x\, P_1 = \boldsymbol{\nu}\, x\, H_1$. Now, the summands of H_1 whose initial action is at x can be removed using **S1-S3** and **R2**; then the remaining term can be rewritten into a hnf H using **R3** or **R1**. If $P = P_1 \mid P_2$, then by induction $\mathcal{A} \vdash P_1 = H_1$, $\mathcal{A} \vdash P_2 = H_2$, for hnf's H_1 and H_2; hence $\mathcal{A} \vdash P = H_1 \mid H_2$. Now $H_1 \mid H_2$ can be put into hnf by means of alpha conversion and the expansion law. Finally, the case $P = P_1 + P_2$ can be accommodated using induction and **S1**. ☐

Theorem 3.16 (completeness of $\mathcal{A}$) *If $P \sim Q$ then $\mathcal{A} \vdash P = Q$.*

Proof: Induction on $d = max\{d(P), d(Q)\}$. By Lemma 3.15 and Proposition 3.14 we can assume that P and Q are in hnf. Moreover, by alpha conversion we can assume that the bound names of all outermost prefixes in P and Q are the same. If $d = 0$, then $P = Q = \mathbf{0}$, hence $\mathcal{A} \vdash P = Q$. Suppose $d > 0$. We show that each summand of P is provable equal to a summand of Q. Then the result follows using the axioms for commutativity, associativity and absorption of sum. If $\alpha.\, P'$ is a summand of P, then $P \xrightarrow{\alpha} P'$. Since $P \sim Q$, there is a summand $\alpha.\, Q'$ of Q s.t. $Q \xrightarrow{\alpha} Q' \sim P'$. But $d(Q') < d$ and $d(P') < d$: Hence, by induction, $\mathcal{A} \vdash P' = Q'$, from which we get $\mathcal{A} \vdash \alpha.\, P' = \alpha.\, Q'$. ☐

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 is 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 transformation to head normal form (Lemma 3.15) can be completed to a transformation to normal forms if process underneath prefixes are manipulated too. 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 constant has a non-negative *arity.*

Definition 4.1 (full πI) *The class of πI processes is defined by adding the production*

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

to the grammar of Definition 2.1. It is assumed that each constant D has a unique defining equation of the form $D \overset{\text{def}}{=} (\widetilde{x})\, P$. Both in a constant definition $D \overset{\text{def}}{=} (\widetilde{x})\, P$ 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. In a constant definition $D \overset{\text{def}}{=} (\widetilde{x})\, P$, all free occurrences of names $\widetilde{x}$ in P are bound; moreover, we require that $\mathrm{fn}(P) \subseteq \widetilde{x}$. The transition rule for constants is:

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

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. The comparison between replication and recursion is interesting. These operators are notational devices to represent syntactically-infinite objects. Replication yields infinity in width (for instance, $!\,\alpha.\,P$ stands for $\alpha.\,P \mid \alpha.\,P \mid \ldots$). Recursion, by contrast, can also capture infinity in depth: For instance, if $D \overset{\text{def}}{=} (x)\, \overline{x}(y).\, D\langle y\rangle$, then $D\langle x_1\rangle$ stands for $\overline{x}_1(x_2).\, \overline{x}_2(x_3).\, \ldots \overline{x}_n(x_{n+1}) \ldots$. In this sense, comparing replication and recursion means comparing infinity in width with infinity in depth.

Milner [Mil91] has showed that in the π-calculus replication and recursion yield the same expressive power, provided that the number of recursive definitions is finite. We shall prove in Section 6.2 that in πI recursion is strictly more powerful than replication. We call $\pi\mathrm{I}^\omega$ the language with replication.

Definition 4.2 ($\pi\mathbf{I}^\omega$) *The class of πI^ω processes is defined by adding the production*

$$P \quad ::= \quad !\,P$$

to the grammar of Definition 2.1.

The transition rule for replication is

$$\frac{P \mid\, !\,P \xrightarrow{\alpha} P'}{!\,P \xrightarrow{\alpha} P'}\,.$$

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, PS93, VH93, Tur96]. We shall see that, however, the polyadic πI enjoys a few properties, for instance on the typing, which do not hold in the polyadic π-calculus.

The operational semantics and the algebraic theory of the polyadic πI are straightforward generalisations of those of the monadic πI, and will be omitted.

4.2.1 The polyadic πI

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:

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

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

$$x(y_1, y_2).\, P \mid \overline{x}(y_1, y_2).\, Q \;\xrightarrow{\tau}\; (\boldsymbol{\nu}\, y_1, y_2\,)(P \mid Q)$$

can be simulated using monadic interactions and an auxiliary fresh name w:

$$\begin{array}{ll} x(w).\, w(y_1).\, w(y_2).\, P \mid \overline{x}(w).\, \overline{w}(y_1).\, \overline{w}(y_2).\, Q & \xrightarrow{\tau}\xrightarrow{\tau}\xrightarrow{\tau} \\ \quad (\boldsymbol{\nu}\, w, y_1, y_2\,)(P \mid Q) \sim (\boldsymbol{\nu}\, y_1, y_2\,)(P \mid Q)\,. & \end{array}$$

4.2.2 The typing system

Having polyadicity, we need to impose some discipline on names so to avoid run-time arity mismatchings in interactions, as for $x(y).\, P \,|\, \overline{x}(y, z).\, Q$. In the π-calculus, this discipline is achieved by means of a *typing system* (in the literature it is sometimes called a *sorting system*; in this paper we shall prefer the word “type” to “sort”). In its basic form, a typing allows us to specify the arity of a name and, recursively, of the names carried by that name. Each name is assigned a type. And each type, say S, is assigned a tuple of types: These are the types of a tuple of names which can be carried by a name of type S. A process which respects a typing will never give rise to run-time errors on the usage of names.

Names of “equal” type can be replaced for one another in a well-typed process, and the resulting process will still be well-typed. There are two main approaches to defining equality between types. In the *by-name* typing [Mil91], each type is given a unique name (i.e., an identification); two types are equal if they have the same name. In the *by-structure* approach [PS93, VH93, Tur96], two types are equal if they are structurally so; in other words, types are viewed as abbreviations for regular trees and equality between types means equality between the underlying regular trees. (There is a close analogy with the by-name and by-structure approaches to the treatment of equality between data types in programming languages.)

In the π-calculus, a by-structure typing represents a special case of a by-name typing, namely the one which makes fewest distinctions among names. The difference between the two systems has semantic consequences. In a by-name typing, distinctions among names can be imposed so to confine the set of free names which can be received in an input. For instance, assigning names x and y different types validates the equation

$$z(x).\, (x \mid \overline{y}) \sim_\pi z(x).\, (x.\, \overline{y} + \overline{y}.\, x) \tag{3}$$

(only names of the same type as x can be received at z, which excludes y — compare (3) with (2) of Section 2.3). By contrast, in a by-structure typing x and y cannot be separated — both are used just for pure synchronisation hence have the same structural type: Therefore equality (3) fails (for the same reason that (2) fails).

In πI, due to the different interpretation of an input — no free name can be received — the by-name and by-structure typing are semantically the same. We shall therefore follow the by-structure approach, because it is more appealing from a mathematical point of view.

Definition 4.3 *The following is our language for types:*

$$S \quad ::= \quad (\widetilde{S}) \;\Big|\; X \;\Big|\; \mu X : S \quad .$$

We use S and T to range over types. X is a type-variable; $\mu X : S$ is a recursive type. Since we want to view type expressions as abbreviations for regular trees, we require that the body of a recursive type $\mu X : S$ be *contractive* in the recursion variable X: Either X does not appear at all in S, or else it appears inside at least one set of brackets. $T\{^{S}\!/X\}$ denotes the capture-avoiding substitution of S for X in T.

Results by Courcelle [Cou83] guarantee that the unfolding of a type generates a unique tree and that the tree is regular, i.e., it only has finitely many different subtrees. We use () to denote the tree whose root has no son; and $(\mathcal{T}_1 \ldots \mathcal{T}_n)$ for the tree whose root has n sons and the i-th son (from left to right) is the root of the tree $\mathcal{T}_i$. We define equality between trees following [Cou83, PS93, VH93].

Definition 4.4 *To each type S we associate a tree called $\mathcal{T}[S]$, which is the unique tree satisfying the following equations:*

1. *if $S = (S_1, \ldots, S_n)$, $n \geq 0$, then $\mathcal{T}[S] = (\mathcal{T}[S_1], \ldots, \mathcal{T}[S_n])$;*
2. *if $S = \mu X.S'$ then $\mathcal{T}[S] = \mathcal{T}[S'\{^{\mu X.S'}\!/X\}]$.*

Two types S and T are equal, *written $S \asymp T$, if $\mathcal{T}[S]$ and $\mathcal{T}[T]$ are syntactically equal.*

We write $x : S$ and $D : S$ if name x and constant D have type S, respectively. Intuitively, $x : (S_1, \ldots, S_n)$ means that x carries n-uples of names whose i-th component has type S_i; similarly, $D : (S_1, \ldots, S_n)$ means that D accepts n-uples of names as parameters, and the i-th name has type S_i.

Definition 4.5 *A* typing *is finite set of assignments of types to names and constants:*

$$\Gamma \quad ::= \quad \emptyset \;\Big|\; \Gamma, x : S \;\Big|\; \Gamma, D : S\,.$$

Names and constants appearing in a typing Γ are always taken to be pairwise distinct; this justifies an abuse of notation whereby Γ is regarded as a finite function from names and constants to types: $\Gamma[x]$ (resp. $\Gamma[D]$) is the type assigned to x (resp. D) by Γ. The ordering of assignments in Γ is ignored.

Definition 4.6 *A process P in πI or in πI^ω is* well-typed for Γ *if $\Gamma \vdash P$ can be inferred from the rules of Table 3. P is* well-typed *if there is Γ s.t. P is well-typed for Γ.*

Note that if P is well-typed for Γ, then all free names and constants in P are nominated in Γ.

Lemma 4.7 *If $\Gamma \vdash P$ and $y \notin fn(P)$, then $\Gamma, y : S \vdash P$ for all S.* ☐

Lemma 4.8 *Let P be a πI or πI^ω process.*

1. *If $\Gamma \vdash P$ and $P \xrightarrow{x(\widetilde{y})} P'$ or $P \xrightarrow{\overline{x}(\widetilde{y})} P'$, then there is $\widetilde{S}$ s.t. $\Gamma[x] \asymp (\widetilde{S})$ and $\Gamma, \widetilde{y} : \widetilde{S} \vdash P'$.*
2. *If $\Gamma \vdash P$ and $P \xrightarrow{\tau} P'$, then $\Gamma \vdash P'$.*

Proof: By transition induction. For rule PAR, Lemma 4.7 is needed. ☐

$$\frac{\Gamma[x] \asymp (\widetilde{S}) \qquad \Gamma\,,\, \widetilde{y} : \widetilde{S} \vdash P}{\Gamma \vdash x(\widetilde{y}).\,P\,,\, \Gamma \vdash \overline{x}(\widetilde{y}).\,P} \qquad\qquad \frac{\Gamma \vdash P}{\Gamma \vdash \tau.\,P}$$

$$\frac{\Gamma \vdash P \qquad \Gamma \vdash Q}{\Gamma \vdash P \mid Q} \qquad\qquad \frac{\Gamma\,,\, x : S \vdash P\,,\ \text{for some } S}{\Gamma \vdash \boldsymbol{\nu}\, x\, P} \quad \frac{\Gamma \vdash P_i\,,\ i \in I}{\Gamma \vdash \sum_{i \in I} P_i}$$

$$\frac{\Gamma[D] \asymp (\Gamma[\widetilde{x}]) \qquad \widetilde{y} : \Gamma[\widetilde{x}] \vdash P}{\Gamma \vdash D\langle \widetilde{x} \rangle} \ \text{if } D \overset{\text{def}}{=} (\widetilde{y})\, P \qquad\qquad \frac{\Gamma \vdash P}{\Gamma \vdash !\, P}$$

Table 3: The typing rules for the operators of πI and $\pi\mathrm{I}^{\omega}$

5 The encoding of the λ-calculus

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 [San95b, BS94] can be easily adapted to πI. More interesting is the encoding of the λ-calculus and of agent-passing calculi into πI or related calculi. We look at the λ-calculus here, and at agent-passing calculi in Sections 7 and 8.

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

$$M := x \;\Big|\; \lambda x.M \;\Big|\; MM$$

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 free 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. The two encodings are presented in Table 4. 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, San95a].

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.

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

……

The encoding into πI:

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

Table 4: The encodings of the linear lazy λ-calculus

The 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 ending up as the subject of a prefix.

The 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 $c \to b$. 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 $\widetilde{u} \to \widetilde{v}$ to abbreviate $u_1 \to v_1 \mid \ldots \mid u_n \to v_n$.

If a and b are names of the same type, then we define:

$$a \to b \stackrel{\text{def}}{=} a(\widetilde{u}).\,\overline{b}(\widetilde{v}).\,\widetilde{v} \to \widetilde{u}$$

(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 5.1 below.

When reasoning about encoding $\mathcal{P}$, one does not have to remember the definition of links; the algebraic properties of links in Lemma 5.1 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 5.1 *Let M be a linear λ-term.*

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

Proof: (Sketch).

1. By exhibiting the appropriate bisimulation.
2. By induction on the structure of M. In the basic case, when $M = x$, assertion (1) is needed. The other cases only require the use of the inductive assumption and a few simple algebraic laws.
3. By induction on the structure of M. In the basic case, M is variable, say $M = x$, and we have

$$\begin{array}{rcl} \boldsymbol{\nu}\, r\,(r \rightarrow p \mid \mathcal{P}[\![x]\!]_r) & = & \\ \boldsymbol{\nu}\, r\,(r \rightarrow p \mid \overline{x}(q).\, q \rightarrow r) & \sim & \\ \overline{x}(q).\, \boldsymbol{\nu}\, r\,(q \rightarrow r \mid r \rightarrow p) & \approx & \\ \overline{x}(q).\, q \rightarrow p & = & \mathcal{P}[\![M]\!]_p \end{array}$$

   where the last transformation uses assertion (1).

   The case when M is an application can be accommodated using simple algebraic laws and assertion (1). If M is an abstraction, say $M = \lambda x.N$, we have

$$\begin{array}{rc} \boldsymbol{\nu}\, r\,(r \rightarrow p \mid \mathcal{P}[\![\lambda x.N]\!]_r) & = \\ \boldsymbol{\nu}\, r\,(r \rightarrow p \mid \overline{r}(w).\, w(x, q).\, \mathcal{P}[\![N]\!]_q) & \end{array}$$

   and, unfolding the definition of $r \rightarrow p$,

$$= \;\boldsymbol{\nu}\, r\,\Big(r(w).\, \overline{p}(w').\, w' \rightarrow w \mid \overline{r}(w).\, w(x, q).\, \mathcal{P}[\![N]\!]_q\Big)\,.$$

   Using one reduction and the definition of links we get

$$\begin{array}{cl} \approx & \overline{p}(w').\, \boldsymbol{\nu}\, w\,\Big(w' \rightarrow w \mid w(x, q).\, \mathcal{P}[\![N]\!]_q\Big) \\ = & \overline{p}(w').\, \boldsymbol{\nu}\, w\,\Big(w'(y, q').\, \overline{w}(x, q).\, (x \rightarrow y \mid q \rightarrow q') \mid w(x, q).\, \mathcal{P}[\![N]\!]_q\Big)\,. \end{array}$$

   Similar algebraic transformations give

$$\begin{array}{cl} \approx & \overline{p}(w').\, w'(y, q').\, \boldsymbol{\nu}\, x, q\,\Big(x \rightarrow y \mid q \rightarrow q' \mid \mathcal{P}[\![N]\!]_q\Big) \\ \sim & \overline{p}(w').\, w'(y, q').\, \boldsymbol{\nu}\, x\,\Big(x \rightarrow y \mid \boldsymbol{\nu}\, q\,(q \rightarrow q' \mid \mathcal{P}[\![N]\!]_q)\Big)\,. \end{array}$$

Finally, from the inductive assumption on $\mathcal{P}[\![N]\!]_q$,

$$\approx \overline{p}(w').\, w'(y,q').\, \boldsymbol{\nu}\, x \left(x \to y \mid \mathcal{P}[\![N]\!]_{q'}\right)$$

and, from assertion (2) on $\mathcal{P}[\![N]\!]_{q'}$,

$$\begin{array}{ll} \approx & \overline{p}(w').\, w'(y,q').\, \mathcal{P}[\![N\{y/x\}]\!]_{q'} \\ \equiv_\alpha & \mathcal{P}[\![\lambda x.N]\!]_p\,. \end{array}$$

□

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 5.1(3).

Theorem 5.2 *For all M and N and p, we have* $\mathcal{P}[\![(\lambda.xM)N]\!]_p \approx \mathcal{P}[\![M\{N/x\}]\!]_p$.

Proof: (Sketch). First, using some reductions, one can show that

$$\mathcal{P}[\![(\lambda.xM)N]\!]_p \approx \boldsymbol{\nu}\, q, x\, (\mathcal{P}[\![M]\!]_q \mid q \to p \mid x(r).\, \mathcal{P}[\![N]\!]_r)\,.$$

The right-hand side is weakly bisimilar with $\boldsymbol{\nu}\, x\, (\mathcal{P}[\![M]\!]_p \mid x(r).\, \mathcal{P}[\![N]\!]_r)$ using Lemma 5.1(3). Finally,

$$\boldsymbol{\nu}\, x\, (\mathcal{P}[\![M]\!]_p \mid x(r).\, \mathcal{P}[\![N]\!]_r) \approx \mathcal{P}[\![M\{N/x\}]\!]_p$$

can be proved by induction on the structure of M and, again, using Lemma 5.1(3). □

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

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

In this way, Lemma 5.1 and Theorem 5.2 remain true for the full lazy λ-calculus.

The encoding $\mathcal{P}$ uses the following recursive types S_l, S_t and S_a for location names, trigger names and auxiliary names like w:

$$\begin{array}{lcl} S_l & \overset{\text{def}}{=} & (S_a) \\ S_t & \overset{\text{def}}{=} & (S_l) \\ S_a & \overset{\text{def}}{=} & (S_t, S_l)\,. \end{array}$$

We shall see in Section 4 that processes in $\pi\mathrm{I}^\omega$ (the calculus with replication in place of recursive agent definition) can be typed with non-recursive types. Since recursive types appear to be necessary to encode the λ-calculus, at least if we require that the encoding is compositional and that each λ-term has a single port to input its argument, we think that there are no encodings of the λ-calculus into $\pi\mathrm{I}^\omega$ with these same properties.

5.1 Encoding the strong 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):

$$\mathcal{P}[\![\lambda x.M]\!]_p \overset{\text{def}}{=} \boldsymbol{\nu}\, q, x \left(\overline{p}(w).\, w(y,r).\, (q \rightarrow r \mid x \rightarrow y) \mid \mathcal{P}[\![M]\!]_q\right).$$

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 A hierarchy of calculi based on internal mobility

6.1 Non-recursive and order-bounded types

Dropping recursion, the language of types (Definition 4.3) simply becomes:

$$S \quad ::= \quad (\widetilde{S})\,.$$

We call them *non-recursive types*, and we call a typing that only uses non-recursive types a *non-recursive typing.* Note that equality $\asymp$ between non-recursive types coincides with syntactic equality.

Definition 6.1 *The* order *of a non-recursive type S is the maximal level of bracket nesting in the definition of S.*

Example 6.2 *Type* $()$ *has order* 1 *and type* $(\,()\,,\,(\,()\,)\,)$ *has order* 3.

If only non-recursive types are used, it makes sense to concentrate on the processes of the language $\pi\mathrm{I}^{\omega}$ (i.e., the recursion-free processes, Definition 4.2): As we shall see in Section 6.2, the confinement to non-recursive types does not affect the typability of processes in $\pi\mathrm{I}^{\omega}$, whereas it affects that of processes in πI. We can discriminate processes according to the order of the types needed in the typing; we thus obtain a hierarchy of calculi. We use ω to denote the first ordinal limit; $n < \omega$ means that n is a positive integers. A non-recursive typing which does not include assignments to constants is a *πI^{ω} typing.*

Definition 6.3 (calculi $\{\pi\mathbf{I}^n\}_{n<\omega}$) *A process $P \in \pi I^{\omega}$ is in πI^n, $n < \omega$, if, for some πI^{ω} typing Γ, there is a derivation proof of $\Gamma \vdash P$ in which all types used (including those in Γ) have order n or less than n.*

That is, the typability of processes in $\pi\mathrm{I}^n$ can be established utilising types of order at most n.

Lemma 6.4 *If $P \in \pi I^n$, then also $P \in \pi I^m$ for all $m \geq n$.* $\square$

Thus $\pi\mathrm{I}^1$ represents the core of CCS, for in $\pi\mathrm{I}^1$ names can only be used for pure synchronisation. $\pi\mathrm{I}^2$ 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. Informally, let us say that a name *depends on* another name if the latter carries the former; for instance, in $x(y).\,\overline{y}(z).\,z.\,\mathbf{0}$, name y depends on x and z depends on y. 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 $\pi\mathrm{I}^m$, for all $m \geq 3$.

Dependency chains are important w.r.t. mobility. If a process has dependency chains of length n at most, then also its traces (i.e., the sequences of actions that the process can perform) have dependency chains of length n at most. In a trace, a dependency between names indicates the creation of a link — hence the creation of mobility. (For instance, if P can perform the action $\overline{x}(z)$, then an interaction in which this action is consumed creates a new link, called z in P.) Similarly, in a trace a dependency chain of length n indicates $n-1$ nested creations of links. Therefore, if a process P, simulating a process Q, has to reproduce the mobility that Q creates, then the dependency chains in traces of P should be at least as long as those in traces of Q (they could be longer, since the creation of a new link by Q might be simulated in more than one step by P, as in the encoding of polyadic communications with monadic communications in Section 4.2).

Following this criterion, in Theorems 6.7 and 6.8 below we show that the calculi $\{\pi\mathrm{I}^n\}_n$ form a strict hierarchy of expressiveness classes: Processes in $\pi\mathrm{I}^{n+1}$ exhibit a "higher degree" of mobility than processes in $\pi\mathrm{I}^n$.

For future investigations, we would like to see if there are stronger formulations of the non-expressiveness results in this and in the following subsection which did not require to deal explicitly with link creation (i.e., the dependency chains among names).

Definition 6.5

- *A* trace *is a sequence of actions $\alpha_1, \ldots, \alpha_n$ s.t. for all $i \neq j$, $bn(\alpha_i) \cap bn(\alpha_j) = \emptyset$.*
- *Let $\ell = \alpha_1, \ldots, \alpha_m$ be a trace and $x \notin bn(\alpha_i)$ for all i. We say that ℓ* has a dependency chain of length 1 from x *if there is $1 \leq i \leq m$ s.t. x is the subject of α_i. We say that ℓ* has a dependency chain of length n from x, *for $n > 1$, if there is $1 \leq i \leq m$ s.t. x is the subject of α_i and there is a name y in the object part of α_i s.t. the trace $\alpha_{i+1}, \ldots, \alpha_m$ has a dependency chain of length $n-1$ from y.*
- *We say that a trace ℓ* has a dependency chain of order n *if there is a name x s.t. ℓ has a dependency chain of length n from x.*

For instance, trace $x(y_1, y_2)\,,\, y_1\,,\, y_2(w)\,,\, \overline{w}$ has various dependency chains of length 2 (among which, the one determined by names x and y_1) and a maximal dependency chain of length 3 (determined by names x, y_2 and w).

Definition 6.6 *A trace $\alpha_1, \ldots, \alpha_n$ is a* trace of the process P_1 *if there are processes $P_2, \ldots, P_{n+1}$ s.t. $P_i \xrightarrow{\alpha_i} P_{i+1}$, for all $1 \leq i \leq n$.*

Theorem 6.7 *There is a trace of a process in* πI^n*,* $n < \omega$*, with a dependency chain of length* n*.*

Proof: Take process $x_1(x_2).\,\cdots.\,x_{n-1}(x_n).\,x_n.\,\mathbf{0}$, and its trace $x_1(x_2),\ldots,x_{n-1}(x_n),x_n$. $\square$

Theorem 6.8 *No trace of a process in* πI^n*,* $n < \omega$*, has a dependency chain of length* $n+1$ *or greater than* $n+1$*.*

Proof: In this proof, we write order$[S]$ for the order of type S. Let Γ, P and x be any $\pi\mathrm{I}^\omega$ typing, process and name, respectively, and suppose that $\Gamma \vdash P$ and that order$[\Gamma[x]] \le n$. We show that any trace $\alpha_1,\ldots,\alpha_m$ of P has dependency chains of length at most n from x. The theorem follows immediately from this claim because if $P \in \pi\mathrm{I}^n$ then $\Gamma \vdash P$, for some Γ which only contains types of order at most n: Therefore, by the claim, a trace of P has dependency chains of length at most n from any name nominated in Γ (which is enough, because, by Lemma 4.8, P can only perform visible actions at names nominated in Γ).

We prove the above claim by induction on n. For the case $n = 1$ just consider that order$[\Gamma[x]] = 1$ means that $\Gamma[x] = ()$: By Lemma 4.8(1), all actions at x have empty object part.

For the case $n > 1$, we use induction on the length m of the trace $\alpha_1,\ldots,\alpha_m$. The case $m = 1$ is trivial: By definition, a trace of length 1 has dependency chains of length at most 1. Now, the case $m > 1$. We suppose that α_1 is an action at x; the case in which α_1 is not an action at x is simpler. Since $\alpha_1,\ldots,\alpha_m$ is a trace of P there is P_1 s.t. $P \xrightarrow{\alpha_1} P_1$ and $\alpha_2,\ldots,\alpha_m$ is a trace of P_1. By Lemma 4.8, if $\alpha_1 = x(\widetilde{y})$ or $\alpha_1 = \overline{x}(\widetilde{y})$, then for some $\widetilde{S}$, $\Gamma[x] \asymp (\widetilde{S})$ and

$$\Gamma, \widetilde{y} : \widetilde{S} \vdash P_1\,. \tag{4}$$

Moreover, since order$[(\widetilde{S})] \le n$, for all $S \in \widetilde{S}$ we have

$$\mathrm{order}[S] \le n - 1\,. \tag{5}$$

Now, a dependency chain from x for the trace $\alpha_1,\ldots,\alpha_m$ is either all contained in $\alpha_2,\ldots,\alpha_m$, or it is split between the traces α_1 and $\alpha_2,\ldots,\alpha_m$. We can show that in both cases the chain must have length at most n. In the former case, this follows from (4) and the inductive assumption on the length of the trace. In the latter case, from (4) and (5), by the induction on the order n, we infer that the trace $\alpha_2,\ldots,\alpha_m$ has dependency chains of length at most $n-1$ from any $y \in \widetilde{y}$; since $\widetilde{y}$ is the object part of α_1, it follows that any dependency chain for $\alpha_1,\ldots,\alpha_m$ starting in α_1 has length at most n. $\square$

6.2 Recursion versus replication

Since replication is a special case of recursion, every process in $\pi\mathrm{I}^\omega$ can be simulated by a process in $\pi\mathrm{I}$. We show that the converse is not possible. (In the π-calculus, the simulation of constants using replications [Mil91, section 3.1] exploits free outputs; for instance a constant application $D\langle z\rangle$ becomes a free output $\overline{x}\langle z\rangle.\,\mathbf{0}$, where x is the channel in which the replication coding the body of the definition of D is located.)

According to the definition of $\pi\mathrm{I}^\omega$ (Definition 4.2), recursive types are allowed in the typing of $\pi\mathrm{I}^\omega$ processes; the lemma below shows that recursive types are in fact not needed.

Lemma 6.9 *If* $P \in \pi I^\omega$ *and well-typed, then also* $P \in \pi I^n$*, for some* $n < \omega$*.*

Proof: By induction on the structure of P. If $P = \mathbf{0}$, then for any n, $P \in \pi\mathrm{I}^n$. If $P = \sum_{i \in I} P_i$ with I non-empty, and $P_i \in \pi\mathrm{I}^{n_i}$, then for $m = max\{n_i \ : \ i \in I\}$ we have $P \in \pi\mathrm{I}^m$ (the maximum exists because I is finite). Parallel composition is handled similarly. If $P = x(\widetilde{y}).\, P'$ or $P = \overline{x}(\widetilde{y}).\, P'$ and $P' \in \pi\mathrm{I}^n$, then $P \in \pi\mathrm{I}^{n+1}$. Finally, if $P \in \pi\mathrm{I}^n$, then also $!\,P$ and $\tau.P$ are in $\pi\mathrm{I}^n$. $\square$

Theorem 6.10 *Suppose $P \in \pi I^\omega$. Then there is $n < \omega$ s.t. no traces of P have a dependency chain of length $n+1$ or greater than $n+1$.*

Proof: By Lemma 6.9, if $P \in \pi\mathrm{I}^\omega$, then $P \in \pi\mathrm{I}^n$, for some n. Then the result follows from Theorem 6.8. $\square$

On the other hand, using recursion we can define a process like $D\langle x_1 \rangle$, for $D \overset{\mathrm{def}}{=} (x)\, \overline{x}(y).\, D\langle y \rangle$, which has traces with dependency chains of unbounded length. For instance, we have

$$D\langle x_1 \rangle \xrightarrow{\overline{x}_1(x_2)} \ldots \xrightarrow{\overline{x}_{n-1}(x_n)} D\langle x_n \rangle \xrightarrow{\overline{x}_n(x_{n+1})} \ldots \qquad (6)$$

Theorem 6.10 and (6) show that recursion cannot be encoded in terms of replication.

The typability of process $D\langle x_1 \rangle$ in (6) requires recursive types. We expect that recursion and replication become interdefinable if only non-recursive types are allowed, and even if a bound on the order of types is imposed.

7 Agent-passing calculi

In an *agent-passing* process calculus, agents, i.e., terms of language, can be passed around. (Sometimes, agent-passing process calculi are called *higher-order* process calculi in the literature.) The agent-passing paradigm inherits from the λ-calculus the idea that a computation step involves instantiation of variables with terms.

For our study of agent-passing process calculi we use the *Higher-Order π-calculus*, a development of the π-calculus introduced in [San92]. Since we want to compare purely-agent-passing calculi with purely-name-passing calculi, we disallow the name-passing features of the Higher-Order π-calculus, namely communication of names and abstraction on names. We call the resulting calculus the *Strictly-Higher-Order π-calculus*, briefly HOπ^ω.

7.1 The Strictly-Higher-Order π-calculus

The following is the grammar of untyped HOπ^ω agents. It combines the familiar CCS-like process constructs — sum, prefixing, parallel composition, restriction and replication — with the λ-calculus constructs — abstraction, application and variable. X, Y, Z and W range over the set of variables.

$$\begin{array}{rcl} A & ::= & \sum_{i \in I} \alpha_i .\, A_i \ \big| \ A \,|\, A \ \big| \ \boldsymbol{\nu}\, x\, A \ \big| \ !\, A \ \big| \ (X)\, A \ \big| \ A\langle A \rangle \ \big| \ X \\ \alpha & ::= & \tau \ \big| \ x(\widetilde{X}) \ \big| \ \overline{x}\langle \widetilde{A} \rangle \end{array}$$

The abstraction construct $(X)\, A$ allows us to define parametrised behaviours, that is, functions from agents to agents. The application construct $A_1\langle A_2 \rangle$ allows us to assign an argument A_2 to an abstraction A_1.

Agents only are exchanged in communications; through an output prefix $\overline{x}\langle\widetilde{A}\rangle$, the tuple of agents $\widetilde{A}$ is emitted; through an input prefix $x(\widetilde{X})$, a tuple of agents is received and instantiates variables $\widetilde{X}$. The angle brackets in an output prefix (as opposed to round brackets) are to emphasise that this is not a binding construct. The definite asymmetry between input and output constructs and, consequently, that of the communication rule of $\mathrm{HO}\pi^\omega$, is a heritage of the λ-calculus, whose basic computational step, beta reduction, is strongly asymmetric.

We abbreviate $(X_1)\ldots(X_n)A$ as $(X_1,\ldots,X_n)A$ and, similarly, $A\langle A_1\rangle\ldots\langle A_n\rangle$ as $A\langle A_1,\ldots,A_n\rangle$. An abstraction $(\widetilde{X})\,A$ and an input prefix $x(\widetilde{X}).\,A$ bind all free occurrences of variables $\widetilde{X}$ in A. An agent is *open* if it may have free variables in it; *closed* otherwise. Abstraction has the highest precedence among the operators, application the lowest; thus $(X)\,A\langle B\rangle$ means $(X)\,(A\langle B\rangle)$, and $\boldsymbol{\nu}\,x\,A\langle B\rangle$ means $\boldsymbol{\nu}\,x\,(A\langle B\rangle)$. The notations introduced for π-calculus, regarding substitutions, tuples, brackets, etc., extend to $\mathrm{HO}\pi^\omega$in the expected way.

Remark 7.1 The $\mathrm{HO}\pi^\omega$ language without replication is enough to write processes with an infinite behaviour (even if well-typedness of the expressions is required) for the same reason why the paradoxical operator `Y` can be written within the λ-calculus. We incorporated replication in the syntax because it will facilitate the comparison with the name-passing calculi $\{\pi\mathrm{I}^n\}_n$, whose operators include replication, in Section 8. ☐

We shall only consider well-typed $\mathrm{HO}\pi^\omega$ terms. We ascribe types to $\mathrm{HO}\pi^\omega$ expressions following the type assignment of the simply-typed λ-calculus. The process-type $()$ is our only first-order (i.e., basic) type. We adopt a bracket-nesting notation for functional types — rather than an arrow notation — mainly to have the same language for types used in Section 4.2, namely:

$$S ::= (\widetilde{S})\ .$$

A term of type $S = (S_1\ \ldots\ S_n)$ takes a sequence of terms of type $S_1\ \ldots\ S_n$ as arguments before becoming a process. Using an arrow-notation, type S would be written as

$$\widehat{S_1} \longrightarrow \ldots \longrightarrow \widehat{S_n} \longrightarrow ()$$

or, "uncurrying" it, as $\widehat{S_1} \times \ldots \times \widehat{S_n} \longrightarrow ()$, where $\widehat{S_i}$ is the arrow-translation of S_i.

A term of type $()$ is a *process*; a term of type $(\widetilde{S})$, for $\widetilde{S}$ non-empty, is an *abstraction*; processes and abstractions are *agents*. P, Q, R and T range over processes; F and G over abstractions; A and B over agents.

Example 7.2 $F \overset{\mathrm{def}}{=} (X)\,(P \mid X)$ *is an abstraction of type* $(\,()\,)$, *and represents a function from processes to processes where the process-argument is run in parallel with* P *in the process-result.*

$G \overset{\mathrm{def}}{=} (X)\,(P \mid X\langle Q\rangle)$ *has type* $(\,(\,()\,)\,)$, *and takes abstractions of the same type as* F *as argument.*

Definition 7.3 *A* $\mathrm{HO}\pi^\omega$ typing *is a finite sequence of assignments of types to names and variables:*

$$\Gamma \quad ::= \quad \emptyset \ \Big|\ \Gamma, x : S \ \Big|\ \Gamma, X : S$$

Definition 7.4 *Let* A *be a* $HO\pi^\omega$ *agent and* Γ *a* $HO\pi^\omega$ *typing. Then* A has type S in Γ *if* $\Gamma \vdash A : S$ *can be inferred from the rules in Table 5;* A *is* well-typed for Γ *if there is a type* S *s.t.* $\Gamma \vdash A : S$ *holds.*

A $HO\pi^\omega$ *agent* A *is* well-typed *if there is* Γ *s.t.* A *is well-typed for* Γ.

$$\begin{array}{cc}
\dfrac{\Gamma[X] = S}{\Gamma \vdash X : S} & \dfrac{\Gamma, X : S \vdash A : (\widetilde{S})}{\Gamma \vdash (X)A : (S, \widetilde{S})} \\[2ex]
\dfrac{\Gamma \vdash A_1 : (S, \widetilde{S}) \quad \Gamma \vdash A_2 : S}{\Gamma \vdash A_1\langle A_2 \rangle : (\widetilde{S})} & \dfrac{\Gamma \vdash A : ()}{\Gamma \vdash \tau.\, A : ()} \\[2ex]
\dfrac{\Gamma[x] = (\widetilde{S}) \quad \Gamma \vdash \widetilde{A} : \widetilde{S} \quad \Gamma \vdash A_1 : ()}{\Gamma \vdash \overline{x}\langle \widetilde{A} \rangle.\, A_1 : ()} & \dfrac{\Gamma[x] = (\widetilde{S}) \quad \Gamma, \widetilde{X} : \widetilde{S} \vdash A : ()}{\Gamma \vdash x(\widetilde{X}).\, A : ()} \\[2ex]
\dfrac{\Gamma \vdash A_1 : () \quad \Gamma \vdash A_2 : ()}{\Gamma \vdash A_1 \mid A_2 : ()} & \dfrac{\Gamma, x : S \vdash A : () \text{ , for some } S}{\Gamma \vdash \boldsymbol{\nu}\, x\, A : ()} \\[2ex]
\dfrac{\forall i,\ \Gamma \vdash A_i : ()}{\Gamma \vdash \sum_{i \in I} A_i : ()} & \dfrac{\Gamma \vdash A : ()}{\Gamma \vdash !\, A : ()}
\end{array}$$

Table 5: Typing rules for HOπ^ω

On the rules in Table 5, note that only the abstraction and application operators take a generic agent as argument; all remaining operators (prefixing, sum, parallel composition, restriction and replication) take processes.

Following the λ-calculus terminology, we call an expression $\big((X)\, A_1\big)\langle A_2\rangle$ a *beta-redex*; normalisation is the operation of consumption of beta redexes, by which the meaning of an expression is disclosed.

Definition 7.5 Beta-conversion*, written* $\succ$*, is the least precongruence on HO*π^ω *agents generated by the rule*

$$\big((X)\, A_1\big)\langle A_2\rangle \succ A_1\{{}^{A_2}\!/\!{}_X\}\,.$$

An agent A_1 *is in* normal form *if there is no* A_2 *s.t.* $A_1 \succ A_2$*. The reflexive and transitive closure of* $\succ$ *is* $\succ^*$*.*

Remark 7.6 The word "normal form" is used in this section and in Section 3.3 (for the proof of completeness of the axiomatisation of πI) for rather different purposes: They reflect the different uses of the word in the process algebra and in the λ-calculus communities. ☐

The lemmas below are proved using standard techniques from the typed λ-calculus [Bar84].

Lemma 7.7 (subject reduction) *If* $\Gamma \vdash A : S$ *and* $A \succ B$*, then also* $\Gamma \vdash B : S$*.* ☐

Lemma 7.8 (uniqueness of normal forms) *For every well-typed HO*π^ω *agent* A *there is a unique normal form* A' *s.t.* $A \succ^* A'$*.* ☐

Lemma 7.9 (termination) *Every sequence of beta conversions starting from a well-typed term* A *eventually leads to the normal form of* A*.* ☐

Definition 7.10 *The unique normal form to which an agent A_1 can be beta converted is called* the normal form of A_1*; we write $A_1 \rhd_\beta A_2$ if A_2 is the normal form of A_1.*

Example 7.11 (continues Example 7.2) *let F and G be defined as in Example 7.2; then for any process R, the normal form of $F\langle R\rangle$ is $P \mid R$; the normal form of $G\langle F\rangle$ is $P \mid P \mid Q$.*

Since normalisation holds, in the following we often restrict our attention to agents in normal form. It is therefore useful to see what normal forms look like. The grammar below describes their syntax.

$$
\begin{array}{rccl}
\text{(processes)} & P & ::= & \sum_{i \in I} \alpha_i . P_i \;\big|\; P \mid P \;\big|\; \boldsymbol{\nu}\, x\, P \;\big|\; !\, P \;\big|\; X \langle \widetilde{A} \rangle \\
\text{(prefixes)} & \alpha & ::= & \tau \;\big|\; x(\widetilde{X}) \;\big|\; \overline{x}\langle \widetilde{A} \rangle \\
\text{(agents)} & A & ::= & P \;\big|\; F \\
\text{(abstractions)} & F & ::= & (\widetilde{X})\, P \;\big|\; (\widetilde{X})\, X \langle \widetilde{A} \rangle
\end{array}
$$

where in the last production, namely $F ::= (\widetilde{X})\, X\langle \widetilde{A}\rangle$, expression $X\langle \widetilde{A}\rangle$ represents a partial application — i.e., $\widetilde{A}$ does not include all arguments that X requires (this production shows that any variable X is a normal form). Note that in a normal form the operator of an application is always a variable.

Remark 7.12 The presentation of the HOπ^ω in this paper is slightly different from that of the Higher-Order π-calculus in [San92]. There, the grammar for normal forms is taken to be the basic syntax of the calculus. Here, we arrive at normal forms through normalisation. $\square$

7.2 Operational semantics

The transition system defining the operational semantics of closed well-typed HOπ^ω processes is presented in Table 6; we have omitted the symmetric of rules PAR and COM. Output transitions are of the form $P \xrightarrow{(\boldsymbol{\nu}\, \widetilde{y}\,)\overline{x}\langle \widetilde{A}\rangle} Q$, where $\widetilde{A}$ is the tuple of agents which are emitted, and $\widetilde{y}$ are private names which occur free in $\widetilde{A}$ and which are carried out from their current scope. We use μ to range over actions (not to be confused with α, which ranges over prefixes). We denote by $\mathrm{bn}(\mu)$ and $\mathrm{n}(\mu)$ the *bound names* and *names* of μ. If μ is a silent or an input action, then $\mathrm{bn}(\mu) = \emptyset$; if μ is an output, say $\mu = (\boldsymbol{\nu}\, \widetilde{y}\,)\overline{x}\langle \widetilde{A}\rangle$, then $\mathrm{bn}(\mu) = \widetilde{y}$. The names of μ are the set of all names which appear in μ.

7.3 A hierarchy of agent-passing process calculi

Similarly to what we did for $\pi\mathrm{I}^\omega$, so from HOπ^ω we define a hierarchy of calculi using the order of their typing. We recall that the order of a type is the maximal level of bracket nesting in its syntactic form.

Definition 7.13 (calculi $\{\mathbf{HO}\pi^n\}_{n<\omega}$) *An agent $A \in$ HOπ^ω is in HOπ^n, $n < \omega$ if, for some typing Γ and type S, there is a derivation proof for $\Gamma \vdash A : S$ in which all types used (including S and the types in Γ) have order n or less than n.*

In HOπ^1 no value is exchanged in communications. Calculus HOπ^1 coincides with $\pi\mathrm{I}^1$ and is the core of CCS. In HOπ^2 only processes can be passed as values in communications; HOπ^2 is the core of Thomsen's Plain CHOCS [Tho90]. The difference between HOπ^1 (resp. HOπ^2) and CCS (resp. Plain

$$\text{ALP: } \frac{P \equiv_\alpha P' \quad P' \xrightarrow{\mu} Q}{P \xrightarrow{\mu} Q} \qquad \text{BETA: } \frac{P \rhd_\beta P' \quad P' \xrightarrow{\mu} Q}{P \xrightarrow{\mu} Q}$$

$$\text{PRE: } \alpha.\, P \xrightarrow{\alpha} P \qquad \text{PAR: } \frac{P \xrightarrow{\mu} P'}{P \mid Q \xrightarrow{\mu} P' \mid Q}\ \mathrm{bn}(\mu) \cap \mathrm{fn}(Q) = \emptyset$$

$$\text{COM: } \frac{P \xrightarrow{(\boldsymbol{\nu}\, \widetilde{y}\,)\overline{x}\langle \widetilde{A} \rangle} P' \quad Q \xrightarrow{x(\widetilde{X})} Q'}{P \mid Q \xrightarrow{\tau} \boldsymbol{\nu}\, \widetilde{y}\, (P' \mid Q'\{\widetilde{A}/\widetilde{X}\})}\quad \widetilde{y} \cap \mathrm{fn}(Q) = \emptyset$$

$$\text{RES: } \frac{P \xrightarrow{\mu} P'}{\boldsymbol{\nu}\, x\, P \xrightarrow{\mu} \boldsymbol{\nu}\, x\, P'}\ x \notin n(\mu) \qquad \text{OPEN: } \frac{P \xrightarrow{(\boldsymbol{\nu}\, \widetilde{y}\,)\overline{z}\langle \widetilde{A} \rangle} P'}{\boldsymbol{\nu}\, x\, P \xrightarrow{(\boldsymbol{\nu}\, x, \widetilde{y}\,)\overline{z}\langle \widetilde{A} \rangle} P'}\ x \neq z, x \in \mathrm{fn}(\widetilde{A}) - \widetilde{y}$$

$$\text{SUM: } \frac{P_i \xrightarrow{\mu} P_i', \quad i \in I}{\sum_{i \in I} P_i \xrightarrow{\mu} P_i'} \qquad \text{REP: } \frac{P \mid\, !\, P \xrightarrow{\mu} P'}{!\, P \xrightarrow{\mu} P'}$$

Table 6: The transition system for HOπ^ω

CHOCS) is that the latter also has a relabeling operator, and it uses recursion in place of replication. In HOπ^3, processes and process abstractions can be communicated as values (a process abstraction takes a process as argument and yields back another process; an example is the agent F in Example 7.2).

Lemma 7.14 *If $P \in HO\pi^n$, then also $P \in HO\pi^m$ for all $m \geq n$.* ☐

8 Comparison between agent-passing calculi and name-passing calculi

In this section, we let n range over $\{1, 2, \ldots, n \ldots,\} \cup \{\omega\}$. We compare the expressiveness of the calculi $\{\mathrm{HO}\pi^n\}_n$ with that of the calculi $\{\pi\mathrm{I}^n\}_n$. It turns out that $\pi\mathrm{I}^n$ is slightly more powerful than HOπ^n. To obtain an exact correspondence, we cut down the class $\pi\mathrm{I}^n$, by imposing a few syntactic conditions on the usage of names in processes. The resulting calculus is called $\pi\mathrm{I}^{n-}$. We shall show that $\pi\mathrm{I}^{n-}$ and HOπ^n have, operationally, the same expressiveness: We exhibit encodings $\{\![\,]\!\}$ and $[\![\,]\!]$, from HOπ^n to $\pi\mathrm{I}^{n-}$, and from $\pi\mathrm{I}^{n-}$ to HOπ^n, in which actions of a source process are mimicked by the corresponding target process, and vice versa.

The encodings $\{\![\,]\!\}$ and $[\![\,]\!]$ are presented in Sections 8.1 and 8.2. First, we introduce the calculi $\{\pi\mathrm{I}^{n-}\}_n$.

Definition 8.1 *Let P be a process in πI^ω. An occurrence of a name in P is a* name-variable *if such occurrence is bound by an input prefix of P.*

Example 8.2 *The name-variables have been underlined in the process* $a(b).\, \big(\underline{\overline{b}}(c).\, c(d) \mid \underline{\overline{b}}(e).\, e(f).\, \underline{f}\big)$.

We suppose that y and z are fresh names. We also assume that Trig_A is the process

$$\mathrm{Trig}_A^{(m)} \stackrel{\mathrm{def}}{=} \begin{cases} !\,y^{(1)}.\,\{\![A]\!\} & \text{if } m = 1 \\ !\,y^{(m)}(z^{(m-1)}).\,\{\![Q]\!\} & \text{if } m > 1 \text{ and } A =_\eta (Z^{(m-1)})\,Q\,. \end{cases}$$

Then $\{\![\,]\!\}$ is defined structurally as follows:

$$\begin{array}{rclrcl}
\{\![\overline{x}^{(n)}\langle A^{(n-1)}\rangle.\,Q]\!\} & \stackrel{\mathrm{def}}{=} & \overline{x}^{(n)}(y^{(n-1)}).\,(\{\![Q]\!\} \mid \mathrm{Trig}_A^{(n-1)}) & \{\![\overline{x}^{(1)}.\,Q]\!\} & \stackrel{\mathrm{def}}{=} & \overline{x}^{(1)}.\,\{\![Q]\!\} \\
\{\![x^{(n)}(Z^{(n-1)}).\,Q]\!\} & \stackrel{\mathrm{def}}{=} & x^{(n)}(z^{(n-1)}).\,\{\![Q]\!\} & \{\![x^{(1)}.\,Q]\!\} & \stackrel{\mathrm{def}}{=} & x^{(1)}.\,\{\![Q]\!\} \\
\{\![Z^{(n)}\langle A^{(n-1)}\rangle]\!\} & \stackrel{\mathrm{def}}{=} & \overline{z}^{(n)}(y^{(n-1)}).\,\mathrm{Trig}_A^{(n-1)} & \{\![Z^{(1)}]\!\} & \stackrel{\mathrm{def}}{=} & \overline{z}^{(1)}.\,\mathbf{0} \\
\{\![Q_1 \mid Q_2]\!\} & \stackrel{\mathrm{def}}{=} & \{\![Q_1]\!\} \mid \{\![Q_2]\!\} & \{\![\sum_{i\in I} Q_i]\!\} & \stackrel{\mathrm{def}}{=} & \sum_{i\in I} \{\![Q_i]\!\}
\end{array}$$

$$\{\![\boldsymbol{\nu}\, x^{(n)}\, Q]\!\} \stackrel{\mathrm{def}}{=} \boldsymbol{\nu}\, x^{(n)}\, \{\![Q]\!\} \qquad \{\![\,!\,Q]\!\} \stackrel{\mathrm{def}}{=} \,!\,\{\![Q]\!\} \qquad \{\![\tau.\,Q]\!\} \stackrel{\mathrm{def}}{=} \tau.\,\{\![Q]\!\}$$

Table 7: The encoding $\{\![\,]\!\}$ from $\mathrm{HO}\pi^n$ to $\pi\mathrm{I}^{n-}$, $n \leq \omega$.

Definition 8.3 (calculi $\{\pi\mathbf{I}^{n-}\}_{n\leq\omega}$) *We call πI^{n-}, $n \leq \omega$, the class of processes in πI^n which satisfy the following syntactic constraint: For any subterm Q of a process in πI^{n-} it holds that*

1. *if $Q = x(\widetilde{y}).\,R$, then any $y \in \widetilde{y}$ appears free in R only in output position;*
2. *if $Q = \overline{x}(\widetilde{y}).\,R$, then any $y \in \widetilde{y}$ appears free in R only in input position;*
3. *if $Q = \overline{x}.R$ and x is a name-variable, then $R = \mathbf{0}$.*

Conditions 1 and 2 say that a name activated in a prefix can be used underneath it only with the polarity opposed to that of the prefix. (From condition 1, since name-variables are bound by input prefixes, it follows that they can be used in output position only.) Condition 3 forces all name-variables used for pure synchronisation to have a trivial continuation. Conditions 1 and 2 could also be described using a typing system similar to that proposed in [PS93], where types also carry informations about the input/output usage of names.

The results for $\{\pi\mathrm{I}^n\}_n$ in Section 6 can be easily adapted to $\{\pi\mathrm{I}^{n-}\}_n$ to prove that also these calculi form a hierarchy in expressiveness. (Note that $\pi\mathrm{I}^{n-}$ is a subcalculus of $\pi\mathrm{I}^n$; we do not know more about the relative expressiveness between the hierarchies $\{\pi\mathrm{I}^n\}_n$ and $\{\pi\mathrm{I}^{n-}\}_n$.)

8.1 From $\mathrm{HO}\pi^n$ to $\pi\mathrm{I}^{n-}$

Since every $\mathrm{HO}\pi^\omega$ agent effectively normalises, it suffices to give the compilation $\{\![\,]\!\}$, from $\mathrm{HO}\pi^n$ to $\pi\mathrm{I}^{n-}$, on processes that are in normal form. That is, formally we assume the rule

$$\{\![Q]\!\} \stackrel{\mathrm{def}}{=} \{\![Q']\!\} \text{ if } Q \rhd_\beta Q'\,.$$

The compilation of a $\mathrm{HO}\pi^\omega$ process P which is well-typed for a typing Γ is defined structurally on P with the rules in Table 7. In these rules, names and agents occurring in P are annotated with the order of the type which is assigned to them in a correct derivation of $\Gamma \vdash P : ()$; these orders are used in the definition of the agent Trig_A. Similarly, we annotated the names of $\{\![P]\!\}$; this will make straightforward to check that $\{\![P]\!\}$ is well-typed (Proposition 8.6). (In the table, metavariables $Q, Q_i, \ldots$ stand for a process and hence have no order annotation). The compilation $\{\![\;]\!\}$ is only defined on the subclass of $\mathrm{HO}\pi^n$ agents in which abstractions have arity one (i.e., they take exactly one argument) and names have arity at most one (i.e., they carry at most one agent). This is purely to make the compilation and the operational correspondence for it more readable; the generalisation to the calculus with arbitrary arities does not give any problem. In the definition of Trig_A in Table 7, $=_\eta$ is a form of eta-conversion used to make all possible arguments of an abstraction explicit. Relation $F_1 =_\eta F_2$, between unary abstractions in normal form, is defined as follows: A unary abstraction of order $m > 1$ in normal form, and with annotated type orders, is either of the form $(Z^{(m-1)})\, Q$ or is a variable $X^{(m)}$; if $F_1 = (Z^{(m-1)})\, Q$, then $F_2 \stackrel{\mathrm{def}}{=} F_1$; if $F_1 = X^{(m)}$ then $F_2 \stackrel{\mathrm{def}}{=} (Z^{(m-1)})\, X^{(m)}\langle Z^{(m-1)}\rangle$.

In the compilation, the communication of an agent A is translated as the communication of a private name which acts as a pointer to (the translation of) A and which the recipient can use to trigger a copy of (the translation of) A. When restricted to $\mathrm{HO}\pi^n$ agents, the compilation coincides with that used in [San92] to translate the full Higher-Order π-calculus down to the π-calculus.

Example 8.4 (from $\mathbf{HO}\pi^2$ to $\pi\mathbf{I}^{2-}$) *Let* $R \stackrel{\mathrm{def}}{=} \overline{v}.\mathbf{0}$ *and* $P \stackrel{\mathrm{def}}{=} \overline{w}\langle R\rangle.\, \mathbf{0} \mid w(X).\, X$. *It holds that* $P \xrightarrow{\tau} R$ *(we garbage-collect* $\mathbf{0}$ *processes). The translation of* P *is*

$$\{\![P]\!\} = \overline{w}(y).\; !\, y.R \mid w(x).\, \overline{x}$$

and we have

$$\begin{aligned} \{\![P]\!\} \quad &\xrightarrow{\tau} \quad \boldsymbol{\nu}\, y\, (\,!\, y.R \mid \overline{y}) && (7)\\ &\sim \quad \tau.\, \boldsymbol{\nu}\, y\, (R \mid \;!\, y.R) && (8)\\ &\sim \quad \tau.\, R && (9)\\ &\approx \quad R \end{aligned}$$

where (7) is derived form the law $\boldsymbol{\nu}\, x\, (\,!\, x(\widetilde{z}).\, P \mid \overline{x}(\widetilde{z}).\, Q) \sim \tau.\, \boldsymbol{\nu}\, x, \widetilde{z}\, (P \mid \;!\, x(\widetilde{z}).\, P \mid Q)$, *(8) from the law* $\boldsymbol{\nu}\, x\, (P \mid \;!\, x(\widetilde{z}).\, Q) \sim P$ *if* $x \notin \mathit{fn}(P)$, *and (9) from the law* $\tau.P \approx P$.

We recall that $\rhd_\beta$ (Definition 7.10) indicates the occurrence of some beta conversion.

Example 8.5 (from $\mathbf{HO}\pi^3$ to $\pi\mathbf{I}^{3-}$) *Let* $R \stackrel{\mathrm{def}}{=} \overline{v}.\mathbf{0}$, $F \stackrel{\mathrm{def}}{=} (X)\, (X \mid X)$, *and* $P \stackrel{\mathrm{def}}{=} \overline{w}\langle F\rangle.\, \mathbf{0} \mid w(Y).\, Y\langle R\rangle$. *We have* $P \xrightarrow{\tau} F\langle R\rangle \rhd_\beta R \mid R$ *(we garbage-collect* $\mathbf{0}$ *processes). Compiling* P, *we get*

$$\{\![P]\!\} = \overline{w}(y).\; !\, y(x).\, (\overline{x} \mid \overline{x}) \mid w(y).\, \overline{y}(x).\; !\, x.R\,.$$

Using algebraic laws similar to those in the previous example, we infer

$$\begin{aligned} \{\![P]\!\} \quad &\xrightarrow{\tau} \quad \boldsymbol{\nu}\, y\, \big(\,!\, y(x).\, (\overline{x} \mid \overline{x}) \mid \overline{y}(x).\; !\, x.R\big)\\ &\sim \quad \tau.\, (\boldsymbol{\nu}\, y, x\,)\big(\overline{x} \mid \overline{x} \mid \;!\, y(x).\, (\overline{x} \mid \overline{x}) \mid \;!\, x.R\big)\\ &\sim \quad \tau.\, \boldsymbol{\nu}\, x\, \big(\overline{x} \mid \overline{x} \mid \;!\, x.R\big)\\ &\sim \quad \tau.\, \tau.\, \tau.\, \boldsymbol{\nu}\, x\, \big(R \mid R \mid \;!\, x.R\big)\\ &\sim \quad \tau.\, \tau.\, \tau.\, \big(R \mid R\big)\\ &\approx \quad R \mid R\,. \end{aligned}$$

Note that $P \in HO\pi^3$ and that $\{\![P]\!\} \in \pi I^{3-}$ (intuitively, the latter because the longest dependency chain in $\{\![P]\!\}$ has length 3, involving names w, y, and x).

In the above examples, the occurrences of $\approx$ show that the computation by a process $\{\![P]\!\}$ may require more steps (i.e., more reductions) than the corresponding computation by P. But if we do not weight internal work, then P and $\{\![P]\!\}$ have the "same" behaviour.

We extend $\{\![\,]\!\}$ to typing as follows: If Γ is a HOπ^ω typing, then $\{\![\Gamma]\!\}$ is the πI$^\omega$ typing obtained from Γ by replacing all variable assignments $X : S$ in Γ with the name assignments $x : S$.

Proposition 8.6

1. *If P is in $HO\pi^\omega$ and is well-typed for Γ, then $\{\![P]\!\}$ is in $\pi I^{\omega-}$ and is well-typed for $\{\![\Gamma]\!\}$.*
2. *If $P \in HO\pi^n$, then $\{\![P]\!\} \in \pi I^{n-}$.*

Proof: Assertions (1) and (2) are evident from the order annotations used in the rules of Table 7. These annotations show that names and agents are mapped onto names of the same order. Thus, if there is a derivation $\Gamma \vdash P : ()$ in which only types of order at most n are used, then there is a derivation of $\{\![\Gamma]\!\} \vdash \{\![P]\!\}$ with the same property.
Moreover, process $\{\![P]\!\}$ is in πI$^{\omega-}$ because the three conditions in Definition 8.3, on input/output usage of names, are met. As for condition 3, note that name-variables used for pure synchronisation are only introduced in the translation of variables of order 1 (rule $\{\![Z]\!\} \overset{\text{def}}{=} \overline{z}.\mathbf{0}$); hence they always prefix the $\mathbf{0}$ process. ☐

Since our compilation is a special case of that in [San92] from Higher-Order π-calculus to π-calculus, the correctness of the latter implies the correctness of the former. Below, some proofs are only sketched.

Lemma 8.7 (operational correspondence for $\{\![\,]\!\}$ on first-order visible actions) *For all $P \in HO\pi^n$:*

1. *If $P \xrightarrow{x} P'$ (resp. $P \xrightarrow{\overline{x}} P'$), then $\{\![P]\!\} \xrightarrow{x} \{\![P']\!\}$ (resp. $\{\![P]\!\} \xrightarrow{\overline{x}} \{\![P']\!\}$).*
2. *the converse, i.e., if $\{\![P]\!\} \xrightarrow{x} P''$ (resp. $\{\![P]\!\} \xrightarrow{\overline{x}} P''$), then there is P' s.t. $P \xrightarrow{x} P'$ (resp. $P \xrightarrow{\overline{x}} P'$) and $P'' = \{\![P']\!\}$.* ☐

Lemma 8.8 (operational correspondence for $\{\![\,]\!\}$ on higher-order inputs) *For all $P \in HO\pi^n$:*

1. *If $P \xrightarrow{x(Y)} P'$, then $\{\![P]\!\} \xrightarrow{x(y)} \{\![P']\!\}$;*
2. *the converse, i.e., if $\{\![P]\!\} \xrightarrow{x(y)} P''$, then there is P' s.t. $P \xrightarrow{x(Y)} P'$ and $P'' = \{\![P']\!\}$.*

Proof: Straightforward transition induction. ☐

In the two lemmas below, Trig$_A$ is the process defined in Table 7; we omit however the order annotations.

Lemma 8.9 (operational correspondence for $\{\![\,]\!\}$ on higher-order outputs) *For all $P \in HO\pi^n$:*

1. *If $P \xrightarrow{\boldsymbol{\nu}\,\widetilde{z}\,\overline{x}\langle A\rangle} P'$, then $\{\![P]\!\} \xrightarrow{\overline{x}(y)} \sim \boldsymbol{\nu}\,\widetilde{z}\,(\{\![P']\!\} \mid \mathrm{Trig}_A)$;*

2. *the converse, i.e., if* $\{\![P]\!\} \xrightarrow{\overline{x}(y)} P''$*, then there are* $\widetilde{z}$*,* A *and* P' *s.t.* $P \xrightarrow{\boldsymbol{\nu}\, \widetilde{z}\, \overline{x}\langle A\rangle} P'$ *and* $P'' \sim \boldsymbol{\nu}\, \widetilde{z}\, (\{\![P']\!\} \mid \mathrm{Trig}_A)$*.*

Proof: By transition induction. Details can be found in [San92, Lemma 5.2.2]. $\square$

Lemma 8.10 *Let* $P \in HO\pi^{\omega}$ *and* $y \notin fn(P)$*. Then for all* $HO\pi^{\omega}$ *agents* A *of the same type as* Y*, it holds that* $\boldsymbol{\nu}\, y\, (\{\![P]\!\} \mid \mathrm{Trig}_A) \approx \boldsymbol{\nu}\, y\, \{\![P\{A/Y\}]\!\}$*.*

Proof: The proof relies on a few non-trivial distributivity properties of replications. Details can be found in [San92, see Lemmas 5.2.2, Theorem 4.4.7 and Theorem 5.2.1(3)]. $\square$

We can now present the main result for $\{\![\,]\!\}$, namely the full abstraction w.r.t. reductions.

Theorem 8.11 (full abstraction for $\{\![\,]\!\}$ on reductions) *For all* $P \in HO\pi^n$*:*

1. *If* $P \xrightarrow{\tau} P'$*, then* $\{\![P]\!\} \xrightarrow{\tau} \approx \{\![P']\!\}$*;*

2. *the converse, i.e., if* $\{\![P]\!\} \xrightarrow{\tau} P''$*, then there is* P' *s.t.* $P \xrightarrow{\tau} P'$ *and* $P'' \approx \{\![P']\!\}$*.*

Proof: Another transition induction. In the basic case (rule `COM`) one needs Lemmas 8.7-8.10. $\square$

8.2 From $\pi\mathrm{I}^{n-}$ to $\mathrm{HO}\pi^n$

The translation $\{\![\,]\!\}$ from $\mathrm{HO}\pi^n$ to $\pi\mathrm{I}^{n-}$, in Section 8.1, used *name-pointers* to model the communication of agents. The translation $[\![\,]\!]$ from $\pi\mathrm{I}^{n-}$ to $\mathrm{HO}\pi^n$, in this section, uses simple *agent-continuations* to model the communication of private names, in the following way. Suppose that a process of $\pi\mathrm{I}^{n-}$ sends a name y, and that the recipient uses y to send another name z and then becomes the process P. In the translation, the communication of y is replaced by the communication of a continuation which has two parameters. The recipient instantiates the first parameter with the continuation for z and the second parameter with (the translation of) P. Continuations for names whose type has order 1, i.e., names used for pure synchronisation, have one parameter only (since by condition 3 in the definition of $\pi\mathrm{I}^{n-}$, such names can only prefix the $\mathbf{0}$ process — that is, the process called P above in this case is always $\mathbf{0}$).

Example 8.12 (from $\pi\mathrm{I}^{2-}$ to $\mathrm{HO}\pi^2$) *If* $P \overset{\mathrm{def}}{=} \overline{x}(y).\, y.\, \mathbf{0} \mid x(y).\, \overline{y}.\, \mathbf{0}$*, then we have*

$$\begin{array}{rcll} P & \xrightarrow{\tau} & \boldsymbol{\nu}\, y\, (y.\, \mathbf{0} \mid \overline{y}.\, \mathbf{0}) & \overset{\mathrm{def}}{=} P_1 \\ & \xrightarrow{\tau} & \boldsymbol{\nu}\, y\, (\mathbf{0} \mid \mathbf{0}) & \overset{\mathrm{def}}{=} P_2\,. \end{array}$$

If $\mathrm{Cont}_y \overset{\mathrm{def}}{=} \overline{y}.\, \mathbf{0}$*, then the translation of* P *is*

$$[\![P]\!] \overset{\mathrm{def}}{=} \boldsymbol{\nu}\, y\, \overline{x}\langle \mathrm{Cont}_y\rangle.\, y.\, \mathbf{0} \mid x(Y).\, Y$$

and we have

$$\begin{array}{rcll} [\![P]\!] & \xrightarrow{\tau} & \boldsymbol{\nu}\, y\, (y.\mathbf{0} \mid \mathrm{Cont}_y) & \\ & = & \boldsymbol{\nu}\, y\, (y.\mathbf{0} \mid \overline{y}.\mathbf{0}) & = [\![P_1]\!] \\ & \xrightarrow{\tau} & \boldsymbol{\nu}\, y\, (\mathbf{0} \mid \mathbf{0}) & = [\![P_2]\!]\,. \end{array}$$

Note that $[\![P]\!] \in HO\pi^2$*, for in* $[\![P]\!]$ *only processes are exchanged.*

Example 8.13 (from $\pi\mathbf{I}^{3-}$ to $\mathbf{HO}\pi^{3}$) *If* $P \overset{\text{def}}{=} \overline{x}(y).\, y(z).\, \overline{z}.\, \mathbf{0} \mid x(y).\, \overline{y}(z).\, z.\, \mathbf{0}$, *then we have*

$$\begin{array}{rcll} P & \xrightarrow{\tau} & \boldsymbol{\nu}\, y\, \big(y(z).\, \overline{z}.\, \mathbf{0} \mid \overline{y}(z).\, z.\, \mathbf{0}\big) & \overset{\text{def}}{=} P_1 \\ & \xrightarrow{\tau} & (\boldsymbol{\nu}\, y, z\,)(\overline{z}.\, \mathbf{0} \mid z.\, \mathbf{0}) & \overset{\text{def}}{=} P_2 \\ & \xrightarrow{\tau} & (\boldsymbol{\nu}\, y, z\,)(\mathbf{0} \mid \mathbf{0}) & \overset{\text{def}}{=} P_3\,. \end{array}$$

If $\mathrm{Cont}_y \overset{\text{def}}{=} (W, U)\, \overline{y}\langle W\rangle.\, U$ *and* $\mathrm{Cont}_z \overset{\text{def}}{=} \overline{z}.\, \mathbf{0}$, *then the translation of* P *is*

$$\llbracket P \rrbracket \;\overset{\text{def}}{=}\; \boldsymbol{\nu}\, y\, \overline{x}\langle \mathrm{Cont}_y\rangle.\, y(Z).\, Z \mid x(Y).\, \boldsymbol{\nu}\, z\, Y\langle \mathrm{Cont}_z, z.\mathbf{0}\rangle\,.$$

We have

$$\begin{array}{rcll} \llbracket P \rrbracket & \xrightarrow{\tau} & \boldsymbol{\nu}\, y\, \big(y(Z).\, Z \mid \boldsymbol{\nu}\, z\, \mathrm{Cont}_y\langle \mathrm{Cont}_z, z.\mathbf{0}\rangle\big) & \\ & \rhd_\beta & \boldsymbol{\nu}\, y\, \big(y(Z).\, Z \mid \boldsymbol{\nu}\, z\, \overline{y}\langle \mathrm{Cont}_z\rangle.\, z.\, \mathbf{0}\big) & = \llbracket P_1 \rrbracket \\ & \xrightarrow{\tau} & (\boldsymbol{\nu}\, y, z\,)(\mathrm{Cont}_z \mid z.\mathbf{0}) & \\ & = & (\boldsymbol{\nu}\, y, z\,)(\overline{z}.\mathbf{0} \mid z.\mathbf{0}) & = \llbracket P_2 \rrbracket \\ & \xrightarrow{\tau} & (\boldsymbol{\nu}\, y, z\,)(\mathbf{0} \mid \mathbf{0}) & = \llbracket P_3 \rrbracket\,. \end{array}$$

There is a one-to-one match between actions of P and of $\llbracket P \rrbracket$: Therefore, the correspondence is even stronger than that for compilation $\{\![\,]\!\}$, for $\{\![\,]\!\}$ may cause an expansion of the number of reductions.

In the definition of the encoding $\llbracket\,\rrbracket$, the difference between names and name-variables is important: $\pi\mathrm{I}^{n-}$ names are mapped onto $\mathrm{HO}\pi^{n}$ names, whereas $\pi\mathrm{I}^{n-}$ name-variables are mapped onto $\mathrm{HO}\pi^{n}$ variables. The encoding is presented in Table 8. As for the compilation $\{\![\,]\!\}$, so in the definition of $\llbracket\,\rrbracket$ names and agents of source and target processes are annotated with the order of their type, according to some typing proof of the source process. To ease readability, $\llbracket\,\rrbracket$ is only defined on the subclass of $\pi\mathrm{I}^{n-}$ processes whose names have at most arity one; the generalisation to the calculus with arbitrary arities is straightforward. The encoding is parametrised over a finite set of names, ranged over by V. Occurrences of names in this set have to be treated as name-variables in the translation. The set can be increased in the rule for input prefix, and decreased in the rule for output prefix; the set is left unchanged in the other rules. We abbreviate $V \cup \{y\}$ as $V \cup y$, $V - \{y\}$ as $V - y$, $\llbracket P \rrbracket_{\{y\}}$ as $\llbracket P \rrbracket_y$, and $\llbracket P \rrbracket_\emptyset$ as $\llbracket P \rrbracket$. Note that the $\mathrm{HO}\pi^{\omega}$ agents returned by $\llbracket\,\rrbracket$ are in normal form.

We extend $\llbracket\,\rrbracket_V$ to typing as follows: If Γ is a $\pi\mathrm{I}^{\omega}$ typing, then $\llbracket \Gamma \rrbracket_V$ is the $\mathrm{HO}\pi^{\omega}$ typing obtained from Γ by adding the variable assignments $X : S$, for all name assignments $x : S$ in Γ s.t. $x \in V$.

Proposition 8.14

1. *If* P *is in* $\pi I^{\omega-}$ *and is well-typed for* Γ, *then* $\llbracket P \rrbracket_V$ *is in* $HO\pi^{\omega}$ *and is well-typed for* $\llbracket \Gamma \rrbracket_V$.
2. *If* $P \in \pi I^{n-}$, *then* $\llbracket P \rrbracket_V \in HO\pi^{n}$.

Proof: Similar argument to that for Proposition 8.6. □

To state in a precise way the results of operational correspondence on visible actions for the encoding $\llbracket\,\rrbracket$, we extend it to actions as follows. Below, Cont_y is the agent defined in Table 8; we omit here the order annotations.

$$\llbracket \alpha \rrbracket = \begin{cases} \alpha & \text{if } \alpha = x \text{ or } \alpha = \overline{x} \text{ or } \alpha = \tau\,, \\ x(Y) & \text{if } \alpha = x(y)\,, \\ \boldsymbol{\nu}\, y\, \overline{x}\langle \mathrm{Cont}_y\rangle & \text{if } \alpha = \overline{x}(y)\,. \end{cases}$$

Let $\mathrm{Cont}_y^{(m)}$ be the agent

$$\mathrm{Cont}_y^{(m)} \overset{\mathrm{def}}{=} \begin{cases} \overline{y}^{(1)}.\,\mathbf{0} & \text{if } m = 1 \\ (W^{(m-1)}, Z^{(1)})\,\overline{y}^{(m)}\langle W^{(m-1)}\rangle.\,Z^{(1)} & \text{if } m > 1 \end{cases}$$

Then $\llbracket\,\rrbracket_V$ is defined structurally as follows:

$$\begin{array}{rcl}
\llbracket \overline{x}^{(n)}(y^{(n-1)}).\,Q \rrbracket_V & \overset{\mathrm{def}}{=} & \begin{cases} \boldsymbol{\nu}\, y^{(n-1)}\,\overline{x}^{(n)}\langle \mathrm{Cont}_y^{(n-1)}\rangle.\,\llbracket Q\rrbracket_{V-y} & \text{if } x \notin V \\ \boldsymbol{\nu}\, y^{(n-1)}\,X^{(n)}\langle \mathrm{Cont}_y^{(n-1)}, \llbracket Q\rrbracket_{V-y}\rangle & \text{if } x \in V \end{cases} \\
\llbracket \overline{x}^{(1)}.\,Q \rrbracket_V & \overset{\mathrm{def}}{=} & \begin{cases} \overline{x}^{(1)}.\,\llbracket Q\rrbracket_V & \text{if } x \notin V \\ X^{(1)} & \text{if } x \in V (*) \end{cases} \\
\llbracket x^{(n)}(y^{(n-1)}).\,Q \rrbracket_V & \overset{\mathrm{def}}{=} & x^{(n)}(Y^{(n-1)}).\,\llbracket Q\rrbracket_{V\cup y} \qquad \llbracket x^{(1)}.\,Q\rrbracket_V \overset{\mathrm{def}}{=} x^{(1)}.\,\llbracket Q\rrbracket_V \\
\llbracket Q_1 \mid Q_2 \rrbracket_V & \overset{\mathrm{def}}{=} & \llbracket Q_1\rrbracket_V \mid \llbracket Q_2\rrbracket_V \qquad \llbracket \sum_{i\in I} Q_i\rrbracket_V \overset{\mathrm{def}}{=} \sum_{i\in I} \llbracket Q_i\rrbracket_V
\end{array}$$

$$\llbracket \boldsymbol{\nu}\, x^{(n)}\, Q\rrbracket_V \overset{\mathrm{def}}{=} \boldsymbol{\nu}\, x^{(n)}\,\llbracket Q\rrbracket_V \qquad \llbracket\, !\, Q\rrbracket_V \overset{\mathrm{def}}{=} !\,\llbracket Q\rrbracket_V \qquad \llbracket \tau.\,Q\rrbracket_V \overset{\mathrm{def}}{=} \tau.\,\llbracket Q\rrbracket_V$$

(*) note that by condition 3 of definition of $\pi\mathrm{I}^{n-}$, if $x \in V$ (i.e., x is a name-variable) then $Q = \mathbf{0}$.

Table 8: The encoding $\llbracket\,\rrbracket$ from $\pi\mathrm{I}^{n-}$ to $\mathrm{HO}\pi^n$, $n \le \omega$.

Lemma 8.15 (operational correspondence for $\llbracket\,\rrbracket$ on visible actions) *For all $P \in \pi I^{n-}$:*

1. *If $P \xrightarrow{\alpha} P'$, for $\alpha \neq \tau$, then $\llbracket P\rrbracket \xrightarrow{\llbracket\alpha\rrbracket} \llbracket P'\rrbracket_V$, where $V = \{y\}$ if α is an input action with bound name y, and $V = \emptyset$ otherwise;*
2. *the converse, i.e., if $\llbracket P\rrbracket \xrightarrow{\mu} P''$ and $\mu \neq \tau$, then there are α and P' s.t. $P \xrightarrow{\alpha} P'$ and $\mu = \llbracket\alpha\rrbracket$, $P'' = \llbracket P'\rrbracket_V$, where $V = \{y\}$ if α is an input action with bound name y, and $V = \emptyset$ otherwise.*

Proof: By transition induction. $\square$

Lemma 8.16 *If $y \notin V$, then $\llbracket P\rrbracket_{V\cup y}\{\mathrm{Cont}_y/Y\} \rhd_\beta \llbracket P\rrbracket_V$, for any $P \in \pi I^{n-}$.*

Proof: We proceed by induction on the structure of P. The most interesting case is when the outermost operator of P is an output prefix at y, say $P = \overline{y}(z).\,P'$ (the case $P = \overline{y}.P'$ is simpler). We have $\mathrm{Cont}_y \overset{\mathrm{def}}{=} (W, Z)\,\overline{y}\langle W\rangle.\,Z$ and, supposing $y \neq z$:

$$\begin{array}{rcl}
\llbracket P\rrbracket_{V\cup y}\{\mathrm{Cont}_y/Y\} & = & \big(\boldsymbol{\nu}\, z\, Y\langle \mathrm{Cont}_z, \llbracket P'\rrbracket_{(V\cup y)-z}\rangle\big)\{\mathrm{Cont}_y/Y\} \\
& \rhd_\beta & \boldsymbol{\nu}\, z\,\overline{y}\langle \mathrm{Cont}_z\rangle.\,(\llbracket P'\rrbracket_{(V\cup y)-z}\{\mathrm{Cont}_y/Y\}) \\
& = & \boldsymbol{\nu}\, z\,\overline{y}\langle \mathrm{Cont}_z\rangle.\,(\llbracket P'\rrbracket_{(V-z)\cup y}\{\mathrm{Cont}_y/Y\})\,.
\end{array}$$

On the other hand, we have $\llbracket P\rrbracket_V = \boldsymbol{\nu}\, z\,\overline{y}\langle \mathrm{Cont}_z\rangle.\,\llbracket P'\rrbracket_{V-z}$. From the inductive assumption,
$\llbracket P'\rrbracket_{(V-z)\cup y}\{\mathrm{Cont}_y/Y\} \rhd_\beta \llbracket P'\rrbracket_{V-z}$; this concludes the case. $\square$

Theorem 8.17 (full abstraction for $\llbracket\,\rrbracket$ on reductions) *For all $P \in \pi I^{n-}$:*

1. *If* $P \xrightarrow{\tau} P'$, *then* $[\![P]\!] \xrightarrow{\tau} \rhd_\beta [\![P']\!]$*;*

2. *The converse, i.e., if* $[\![P]\!] \xrightarrow{\tau} P''$, *then there is* P' *s.t.* $P \xrightarrow{\tau} P'$ *and* $P'' \rhd_\beta [\![P']\!]$*.*

Proof: The proofs of the two assertions are similar, and proceed by transition induction. The most interesting case is given by rule `COM`, when the interacting names have a type of order greater than 1. We examine this case, for assertion (1). Thus suppose $P = P_1 \mid P_2$, and

$$\frac{P_1 \xrightarrow{\overline{x}(y)} P_1' \quad P_2 \xrightarrow{x(y)} P_2'}{P_1 \mid P_2 \xrightarrow{\tau} \boldsymbol{\nu}\, y\, (P_1' \mid P_2')} \,.$$

We have $[\![P_1 \mid P_2]\!] = [\![P_1]\!] \mid [\![P_2]\!]$ and, from Lemma 8.15,

$$[\![P_1]\!] \xrightarrow{\boldsymbol{\nu}\, y\, \overline{x}\langle \mathrm{Cont}_y \rangle} [\![P_1']\!] \,, \qquad [\![P_2]\!] \xrightarrow{x(Y)} [\![P_2']\!]_y \,.$$

From these, and the rule `COM` of $\mathrm{HO}\pi^\omega$, we deduce

$$[\![P_1]\!] \mid [\![P_2]\!] \xrightarrow{\tau} \boldsymbol{\nu}\, y\, ([\![P_1']\!] \mid [\![P_2']\!]_y\{^{\mathrm{Cont}_y}\!/_Y\})$$

By Lemma 8.16, $[\![P_2']\!]_y\{^{\mathrm{Cont}_y}\!/_Y\} \rhd_\beta [\![P_2']\!]$. Summarising, we have:

$$[\![P_1 \mid P_2]\!] = [\![P_1]\!] \mid [\![P_2]\!] \xrightarrow{\tau} \rhd_\beta \boldsymbol{\nu}\, y\, ([\![P_1']\!] \mid [\![P_2']\!]) = [\![P']\!]$$

which concludes the case. ☐

9 Conclusions and future work

The work in this paper leads to a classification of name-passing process calculi according to the "degree" of mobility permitted: π-calculus permits both internal and external mobility; $\pi\mathrm{I}$ permits internal mobility; $\pi\mathrm{I}^\omega$ permits internal but not recursive mobility; $\pi\mathrm{I}^n$, $n < \omega$, permits internal mobility of order n at most; $\pi\mathrm{I}^1$, which is the core of CCS, does not permit mobility at all.

This scale can be used for comparative assessments of calculi as well as of processes. For instance, the modelling of the *locality* relation in [San95b] only utilises internal mobility of order 3, whereas the modelling of the *causality* relation in [BS94] requires at least internal mobility of order 4; this reflects the fact that causality is a more sophisticated relation than locality. Other examples of use of the scale come from Section 5 of this paper, where we argued that the encoding of the untyped λ-calculus requires at least recursive internal mobility, and from Section 8, where we studied the expressiveness of agent-passing calculi.

We have also presented a hierarchy of agent-passing process calculi: In $\mathrm{HO}\pi^\omega$ agents of arbitrary order can be communicated; in $\mathrm{HO}\pi^n$, $n < \omega$, agents of order n at most can be communicated. Roughly, $\mathrm{HO}\pi^1$ coincides with $\pi\mathrm{I}^1$ and CCS, and $\mathrm{HO}\pi^2$ — where only processes can be communicated — with Thomsen's Plain CHOCS. We have proved that there is a strong connection, in terms of expressiveness, between this hierarchy of agent-passing calculi and the hierarchy of name-passing calculi $\pi\mathrm{I}^1$, $\pi\mathrm{I}^2$, ..., $\pi\mathrm{I}^\omega$, i.e., the calculi using internal and non-recursive mobility. Note in particular the correspondence between $\mathrm{HO}\pi^2$ and $\pi\mathrm{I}^{2-}$: Process passing only gives little expressiveness more than CCS.

These are results of *relative* expressiveness. Further work is needed, both to complete the comparison among the above-mentioned calculi, and to understand their *absolute* expressiveness. We are particularly interested in the expressiveness of πI, which we expect to be rather close to that of the π-calculus. We have showed that, besides agent-passing calculi, also data values and the λ-calculus can be modelled in πI. The translation of the λ-calculus is obtained by refining Milner's encoding into the π-calculus, which makes non-trivial use of the free-output construct — disallowed in πI. Therefore, we hope that the encoding might also give insights into the comparison between πI and π-calculus.

When discussing the calculi $\pi\mathrm{I}^n$, we have proved non-expressiveness results among them by explicitly taking into account the patterns of creation of mobility. A challenging problem for future research will be to establish similar results using a more extensional criterion i.e., without looking at link creations.

For the translation of the λ-calculus, we 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, 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.

It would be interesting to see how to recast the calculi and the hierarchies of them presented in this paper in the framework of *action calculi* [Mil93]. These have been proposed by Milner as a unifying framework for representing a variety of models of interaction, including Petri nets and the π-calculus. Milner [Mil94] is investigating classifications of action calculi according to their dynamics. This line of research is still in its early stages and it is premature to draw precise comparisons, but we should at least observe that one of the central classifying objects in [Mil94] bears some resemblance to πI.

Another topic for future research is how to increase the expressiveness of agent-passing process calculi. The most powerful agent-passing process calculus considered in this paper is $\mathrm{HO}\pi^\omega$; we have seen that its expressiveness is not greater than that of $\pi\mathrm{I}^\omega$. To increase the expressiveness of $\mathrm{HO}\pi^\omega$ — so as to get closer to that of πI — one might add recursive types to $\mathrm{HO}\pi^\omega$. This extension, however, could destroy properties of $\mathrm{HO}\pi^\omega$, like normalisation, which are important when reasoning about behavioural equivalence between $\mathrm{HO}\pi^\omega$ processes.

Acknowledgments I have benefited from discussions with Gerard Boudol, Claudio Calvelli, Robin Milner, David N. Turner and David Walker, and from the suggestions of the anonymous referees.

References

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

[AGR92] E. Astesiano, A. Giovini, and G. Reggio. Observational structures and their logic. *Theoretical Computer Science*, 96:249–283, 1992.

[Ama93] R. Amadio. On the reduction of CHOCS bisimulation to π-calculus bisimulation. In E. Best, editor, *Proceedings of CONCUR '93*, volume 715 of *Lecture Notes in Computer Science*. Springer Verlag, 1993.

[Bar84] H. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic*. North Holland, 1984. Revised edition.

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

[BS94] M. Boreale and D. Sangiorgi. A fully abstract semantics for causality in the π-calculus. Technical Report ECS–LFCS–94–297, LFCS, Dept. of Comp. Sci., Edinburgh Univ., 1994. An extract has appeared in *Proc. STACS'95*, LNCS 900, Springer Verlag.

[Cou83] B. Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25:95–169, 1983.

[DKV91] P. Degano, S. Kasangian, and S. Vigna. Applications of the calculus of trees to process description languages. In *Proc. of the CTCS '91 Conference*, volume 530 of *Lecture Notes in Computer Science*, pages 281–301. Springer Verlag, 1991.

[FMQ94] G. Ferrari, U. Montanari, and P. Quaglia. A π-calculus with explicit substitutions: the late semantics. In I. Prívara, B. Rovan, and P. Ružička, editors, *Proc. MFCS'94*, volume 841 of *Lecture Notes in Computer Science*. Springer Verlag, 1994.

[Hen93] M. Hennessy. A fully abstract denotational model for higher-order processes. In *8th LICS Conf.* IEEE Computer Society Press, 1993.

[Jon93] C.B. Jones. A π-calculus semantics for an object-based design notation. In E. Best, editor, *Proceedings of CONCUR '93*, volume 715 of *Lecture Notes in Computer Science*, pages 158–172. Springer Verlag, 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. Also in *Logic and Algebra of Specification*, ed. F.L. Bauer, W. Brauer and H. Schwichtenberg, Springer Verlag, 1993.

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

[Mil93] R.. Milner. Action calculi, or syntactic action structures. In *Proc MFCS'93*, volume 711 of *Lecture Notes in Computer Science*, pages 105–121. Springer Verlag, 1993.

[Mil94] R. Milner. Dynamic classification of action calculi. Handwritten notes, September, 1994.

[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.

[San95a] D. Sangiorgi. Lazy functions and mobile processes. Technical Report RR-2515, INRIA-Sophia Antipolis, 1995. available via anonymous ftp from `cma.cma.fr` as `pub/papers/davide/RR-2515.ps.Z`.

[San95b] D. Sangiorgi. Locality and non-interleaving semantics in calculi for mobile processes. *Theoretical Computer Science*, 1995. To appear. An extract appeared in *Proc. TACS '94*, Lecture Notes in Computer Science 789, Springer Verlag.

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

[Tur96] N.D. Turner. Forthcoming PhD thesis, Department of Computer Science, University of Edinburgh, 1996.

[VH93] V.T. Vasconcelos and K. Honda. Principal typing schemes in a polyadic π-calculus. In E. Best, editor, *Proceedings of CONCUR '93*, volume 715 of *Lecture Notes in Computer Science.* Springer Verlag, 1993.

[Wal95] D. Walker. Objects in the π-calculus. *Information and Computation*, 116(2):253–271, 1995.