The π-calculus without α-conversion

Daniele Varacca, Nobuko Yoshida
Imperial College London

Abstract

Dynamic creation of new names is one of the distinguishing features of the π-calculus. The notion of α-conversion of bound names plays an important role in formalising name creation. This paper shows a small but surprising fact on expressiveness of the π-calculus: for a powerful fragment of π-calculi where communication happens internally and linearly, α-conversion is not necessary. We show that this linear π-calculus with internal mobility can be fully abstractly translated into a CCS where names need not be generated dynamically during the computation ("at run time"), but they can be instantiated during the typing ("at compile time"). This opens a possibility that we can reuse many fruitful results established in CCS to the π-calculus. In particular we could provide an Event Structure-based semantics of the π-calculus, which is a difficult task in the presence of α-conversion.

1 Introduction and motivations

Dynamic creation of new names and name passing are the main features that distinguish the π-calculus from CCS. Syntactically, the creation of new names is performed by a combination of α-conversion and scope opening. Consider the following transition rule, where $\equiv_\alpha$ denotes α-equivalence, while β is a label denoting the nature of the transition:

$$\frac{P \equiv_\alpha P' \quad P \xrightarrow{\beta} Q}{P' \xrightarrow{\beta} Q}$$

This apparently innocent rule is often necessary for allowing processes to communicate. When a process outputs a new name, the receiver and the sender agree on such a name, which then becomes known only to them. In order to do this, an application of the rule above may be necessary. Consider the process:

$$C \stackrel{\text{def}}{=} a(x).P \mid a(y).Q \mid (\nu z)\overline{a}\langle z\rangle.R$$

where $\mid$ denotes parallel composition, $a(x).P$ represents input, (νz) is name restriction and $\overline{a}\langle z\rangle.R$ represents an output. Note that the name output by the third subprocess is bound, and can be instantiated by any *fresh* name. The process C can do a silent transition to either of the two following processes:

$$(\nu w)(P[w/x] \mid R[w/z]) \mid a(y).Q$$

$$(\nu w)(Q[w/y] \mid R[w/z]) \mid a(x).P$$

where w is a fresh name. Dynamically, a new name is created which R shares either with P or with Q. Note that at least one of the subprocesses involved must perform an α-conversion on a bound name.

This paper shows a small but surprising fact: in some restricted but powerful fragment of the π-calculus, dynamic α-conversion is no longer necessary. For example consider the process:

$$(\nu a)(a(x).P \mid (\nu z)\overline{a}\langle z\rangle.R)$$

where a does not appear in P and R. Such name a is called *linear* in the literature [13, 1, 25, 20]. Since the there is only one possible communication on a, we could choose the name the subprocesses agree upon *before* the communication actually happens, and restrict it to make sure no one else knows such name:

$$(\nu a)(\nu w)(a\langle w\rangle.P[w/x] \mid \overline{a}\langle w\rangle.R[w/z])$$

The input does not need to be binding, and instead of name passing, we could speak of name *sharing*. This simplification is not possible on the process C above, as it is not possible to decide statically which communication will actually happen. However the subclass of processes we consider does not include process C.

The first restriction we impose is that we only study Sangiorgi's πI-calculus [18, 19], a version of the π-calculus where only fresh names can be communicated. The theory of the πI-calculus is considerably simpler than the one of the full calculus, and it is closer to the theory of CCS. In the transition system for the π-calculus, three kinds of actions are present: input, free output and bound output. In the πI-calculus only input and bound output are present, allowing a complete symmetry between input and output, analogous to the one featured by CCS. Also, there is only one natural notion of congruent bisimulation for the πI-calculus, while this is not the case for the full calculus [15]. Moreover, when using recursion instead of replication, the πI-calculus is as expressive as the full calculus [18, 19].

The second restriction we impose is by means of a type system. Various typing systems for the π-calculus have been studied in the literature. The typing system we consider in this work restricts the universe of processes to the so-called *linear* ones [13, 20]. The expressive power of linearly typed π-calculus is strictly less that the one of the full calculus. However linear processes have enough power to *fully abstractly* encode a family of the functional calculi, such as PCF [1], a simply typed λ-calculus with sums and products [25], the System F [2] and full control calculi [11]. In practice, the linear type discipline is used for guaranteeing deadlock-freedom [25, 12], proving aggressive optimisations of encodings [13, 20] and secure information flow analysis [9, 10, 26].

Even if we restrict our calculus to a linear πI-calculus, to erase dynamic α-conversion is non-trivial due to possibly infinite computation which generates infinite many new names. For example, consider the following πI-calculus term:

$$P_\omega = \mathtt{Fw}\langle ab\rangle \mid \mathtt{Fw}\langle ba\rangle \mid (\nu z)\overline{a}\langle z\rangle.z.\overline{c}$$

where $\mathtt{Fw}\langle ab\rangle \stackrel{\text{def}}{=} !a(x).(\nu y)\overline{b}\langle y\rangle.y.\overline{x}$. The process P_ω is perfectly typable by the linear typing system. Here $!a(x).P$ represents an input receptor which creates a copy of P when it is invoked by a message on a. Using the operational semantics of the π-calculus, formally presented in the next section, the reader can easily check that, after an initial interaction at a, P_ω generates infinitely many different bound names because of the name restriction under the prefix.

This paper shows that a family of linear πI-calculi, which includes the above P_ω, can be fully abstractly translated into a linearly typed CCS where names need not be generated dynamically during the computation ("at run time"), but they can be instantiated during the typing ("at compile time"). This opens a possibility to smoothly inherit many accumulated theories studied in CCS to the π-calculus, in particular it allows to provide an Event Structure semantics of π-calculus on the same lines as the semantics of CCS [24]. This has been technically difficult to achieve before, and it is still difficult for the full π-calculus, due to the α-conversion rule.

2 A linear version of the π calculus

This section briefly summarises a linear version of the π-calculus in [1]. Although this summary is technically self-contained, the reader may refer to [1] for detailed illustration and more examples.

2.1 Syntax and Reduction

The π-calculus used in this abstract is a subset of the standard π-calculus [16]. The following gives the reduction rule of the π-calculus:

$$x(\tilde{y}).P \,|\, \overline{x}\langle\tilde{v}\rangle.Q \longrightarrow P\{\tilde{v}/\tilde{y}\} \,|\, Q$$

Here $\tilde{y}$ denotes a potentially empty vector $y_1 ... y_n$. Operationally, this reduction represents the consumption of a message by a receptor.

As anticipated, we consider a restricted version of the π-calculus, where only bound names are passed in interaction. Besides producing a simpler and more elegant theory, this restriction allows tighter control of sharing and aliasing without losing essential expressiveness, making it easier to administer name usage in more stringent ways. The resulting calculus is called the πI-calculus in the literature [19] and has the same expressive power as the version with free name passing (as proved Section 6 in [25]). Syntactically we restrict an output to the form $(\nu\,\tilde{y})\overline{x}\langle\tilde{y}\rangle.P$ (where names in $\tilde{y}$ are pairwise distinct), which we henceforth write $\overline{x}(\tilde{y}).P$. For dynamics, we have the following forms of reduction by the restriction to the bound output. The reduction relation $\longrightarrow$ is generated from the following rules, closed under output prefix, restriction and parallel composition (modulo $\equiv$):

$$\begin{array}{rcl} x(\tilde{y}).P \,|\, \overline{x}(\tilde{y}).Q & \longrightarrow & (\nu\,\tilde{y})(P \,|\, Q) \\ !x(\tilde{y}).P \,|\, \overline{x}(\tilde{y}).Q & \longrightarrow & !x(\tilde{y}).P \,|\, (\nu\,\tilde{y})(P \,|\, Q) \end{array}$$

After communication, $\tilde{y}$ are shared between P and Q. The formal grammar of the calculus is defined below. Below and henceforth $x, y, \ldots$ range over a countable set of names.

$$P ::= x(\tilde{y}).P \;\mid\; \overline{x}(\tilde{y}).P \;\mid\; P \,|\, Q \;\mid\; (\nu\,x)P \;\mid\; \mathbf{0} \;\mid\; !x(\tilde{y}).P$$

$x(\tilde{y}).P$ (resp. $\overline{x}(\tilde{y})P$) is an input (resp. output). $P \,|\, Q$ is a parallel composition, $(\nu\,x)P$ is a restriction and $!x(\tilde{y}).P$ is a replicated input. We omit the empty vector: for example, $\overline{a}$ stands for $\overline{a}()$. The bound/free names are defined as usual. We assume that names in a vector $\tilde{y}$ are pairwise distinct. We also use the standard convention that in any process the set of bound names and the set of free names are disjoint. This is not restrictive as processes are identified up to α-equivalence.

The definitions of structural equality $\equiv$ is standard, and in particular it includes the monoidal laws for $|$ as well as α-equivalence $\equiv_\alpha$ [16, 1, 25, 9].

2.2 Types and Typings

This subsection reviews the basic idea of the linear type discipline in [1]. We can easily extend the result in this paper to other family of the linear calculi studied in [1, 25, 9].

This type discipline allows a precise embedding of functional computation in the π-calculus by restricting process behaviour to be a *confluent* one. To realise confluence, we use:

(A) for each linear name there are a unique input and a unique output; or

(B) for each replicated name there is a unique stateless replicated input with zero or more dual outputs.

Example. As an example for the first condition, let us consider:

$$Q_1 \stackrel{\text{def}}{=} \overline{a}.b \,|\, \overline{a}.c \,|\, a \qquad\qquad Q_2 \stackrel{\text{def}}{=} b.\overline{a} \,|\, c.\overline{b} \,|\, a.(\overline{c} \,|\, \overline{e})$$

Then Q_1 is not typable as a appears twice as output, while Q_2 is typable since each channel appears at most once as input and output. As an example of the second constraint, let us consider the following two processes:

$$Q_3 \stackrel{\text{def}}{=} !b.\overline{a} \,|\, !b.\overline{c} \qquad\qquad Q_4 \stackrel{\text{def}}{=} !b.\overline{a} \,|\, \overline{b} \,|\, !c.\overline{b}$$

Q_3 is untypable because b is associated with two replicators: but Q_4 is typable since, while output at b appears twice, a replicated input at b appears only once.

Types. Channel types are inductively made up from type variables and action modes. The four *action modes* speak about the directional and quantitative channel usage:

$\downarrow$	Linear input,	$\uparrow$	Linear output,
!	Server at replicated input,	?	Client requests to !.

Input modes are $\downarrow, !$, while $\uparrow, ?$ are *output modes*. We let $p, p', \ldots$ denote modes. We define $\overline{p}$, the *dual* of p, by: $\overline{\downarrow} = \uparrow$, $\overline{!} = ?$ and $\overline{\overline{p}} = p$. Then the syntax of types are given as follows:

$$\begin{array}{rcl} \sigma & ::= & (\tilde{\sigma})^p \\ \tau & ::= & \sigma \mid \updownarrow \end{array}$$

where $\tilde{\sigma}$ is a vector of types. The type $\updownarrow$ denotes a channel that cannot be used by other processes running in parallel. It cannot be nested inside input/output types. We write $MD(\tau)$ for the outermost mode of τ. The *dual of* τ, written $\overline{\tau}$, is the result of dualising all action modes. Duality is not defined for $\updownarrow$. A type environment Γ is a finite mapping from channels to channel types. The domain of Γ is denoted as $Dom(\Gamma)$. Sometimes we will write $x \in \Gamma$ to mean $x \in Dom(\Gamma)$.

Types restrict the composability of processes: for example, for parallel composition, if P is typed under environment Γ_1, Q is under Γ_2 and $\Gamma_1 \odot \Gamma_2$ is defined for a partial operator $\odot$ with the resulting Γ, then we assign Γ to $P \mid Q$. If $\Gamma_1 \odot \Gamma_2$ is not defined, the composition is not allowed. Formally, $\odot$ is the partial commutative operation on Γ_1 and Γ_2 where $\Gamma_1 \odot \Gamma_2 \stackrel{\text{def}}{=} \Gamma$ is defined as follows:

(1)
- if $\Gamma_1(x) = (\tilde{\tau})^{\downarrow}$ and $\Gamma_2(x) = (\tilde{\overline{\tau}})^{\uparrow}$ then $\Gamma(x) = \updownarrow$, and symmetrically;
- if $\Gamma_1(x) = (\tilde{\tau})^{\downarrow}$ and $x \notin Dom(\Gamma_2)$ then $\Gamma(x) = (\tilde{\tau})^{\downarrow}$, and symmetrically;
- if $\Gamma_1(x) = (\tilde{\tau})^{\uparrow}$ and $x \notin Dom(\Gamma_2)$ then $\Gamma(x) = (\tilde{\tau})^{\uparrow}$, and symmetrically;

(2)
- if $\Gamma_1(x) = (\tilde{\tau})^{!}$ and $\Gamma_2(x) = (\tilde{\overline{\tau}})^{?}$ then $\Gamma(x) = (\tilde{\tau})^{!}$, and symmetrically;
- if $\Gamma_1(x) = (\tilde{\tau})^{?}$ and $\Gamma_2(x) = (\tilde{\tau})^{?}$ then $\Gamma(x) = (\tilde{\tau})^{?}$.

(3) undefined in any other cases (if any of the other cases arises, then the whole $\Gamma_1 \odot \Gamma_2$ is not defined).

Intuitively, the rules in (2) say that a server should be unique, but an arbitrary number of clients can request interactions. The rules in (1) say that once we compose input-output linear channels, the channel becomes uncomposable. Note that (3) says other compositions are undefined. (1) and (2) ensures the two constraints (A) and (B) in § 2.2.

Typing System Typing rules are defined in Figure 1. The (Zero) rule types $\mathbf{0}$. As $\mathbf{0}$ has no free names, it is not being given any channel types. In (Par), $\Gamma_1 \odot \Gamma_2$ guarantees the consistent channel usage like linear inputs being only composed with linear outputs, etc. In (Res), we do not allow $\uparrow$, ? or $\downarrow$-channels to be restricted since they carry actions which expect their dual actions to exist in the environment. (WeakOut) and (WeakCl) weaken with ?-names or $\updownarrow$-names, respectively, since these modes do not require *further* interaction. (LIn) ensures that x occurs precisely once. (LOut) is dual. (RIn) is the same as (LIn) except that no free linear channels are suppressed. This is because a linear channel under replication could be used more than once. (ROut) is similar with (LOut). Note we need to apply (WeakOut) before the first application of (ROut).

$$\frac{}{\mathbf{0} \triangleright \emptyset}\ \text{Zero} \qquad \frac{P_i \triangleright \Gamma_i \quad (i=1,2)}{P_1 \mid P_2 \triangleright \Gamma_1 \odot \Gamma_2}\ \text{Par} \qquad \frac{a \notin \Gamma \quad MD(\tau) = !, \updownarrow \quad P \triangleright \Gamma, a : \tau}{(\nu a) P \triangleright \Gamma}\ \text{Res}$$

$$\frac{P \triangleright \Gamma \quad x \notin \Gamma}{P \triangleright \Gamma, x : (\tilde{\tau})^{?}}\ \text{WeakOut} \qquad \frac{P \triangleright \Gamma \quad x \notin \Gamma}{P \triangleright \Gamma, x : \updownarrow}\ \text{WeakCl} \qquad \frac{P \triangleright \Gamma, \tilde{y} : \tilde{\tau} \quad a \notin \Gamma}{a(\tilde{y}).P \triangleright \Gamma, a : (\tilde{\tau})^{\downarrow}}\ \text{LIn}$$

$$\frac{P \triangleright \Gamma, \tilde{y} : \tilde{\tau} \quad a \notin \Gamma}{\overline{a}(\tilde{y}).P \triangleright \Gamma, a : (\tilde{\tau})^{\uparrow}}\ \text{LOut} \qquad \frac{P \triangleright \Gamma, \tilde{y} : \tilde{\tau} \quad a \notin \Gamma \quad \forall (x : \tau) \in \Gamma.\ MD(\tau) = ?}{!a(\tilde{y}).P \triangleright \Gamma, a : (\tilde{\tau})^{!}}\ \text{RIn} \qquad \frac{P \triangleright \Gamma, a : (\tilde{\tau})^{?}, \tilde{y} : \tilde{\tau}}{\overline{a}(\tilde{y}).P \triangleright \Gamma, a : (\tilde{\tau})^{?}}\ \text{ROut}$$

Figure 1: Linear Typing Rules

2.3 A Typed Labelled Transition Relation

Typed transitions describe the observations a typed observer can make of a typed process. The typed transition relation is a proper subset of the untyped transition relation, while not restricting $\boldsymbol{\tau}$-actions: hence typed transitions restrict observability, not computation. Let the set of *labels* $\alpha, \beta, \ldots$ be given by the following grammar:

$$\alpha, \beta, \ldots \quad ::= \quad \boldsymbol{\tau} \ \mid\ x(\tilde{y}) \ \mid\ \overline{x}(\tilde{y})$$

$x(\tilde{y})$ (resp. $\overline{x}(\tilde{y})$) stands for a bound input (resp. output).

The standard untyped transition relation is defined in Figure 2. We need to define in which way an environment restricts observability. In order to do this we define the relation Γ allows α as follows:

for all Γ, Γ allows $\boldsymbol{\tau}$;
if $MD(\Gamma(x)) = \downarrow, !$, then Γ allows $x(\tilde{y})$;
if $MD(\Gamma(x)) = \uparrow, ?$, then Γ allows $\overline{x}(\tilde{y})$.

Intuitively, labels only allowed when the type environment is coherent with them. Whenever Γ allows α, we define $\Gamma \setminus \alpha$ as follows:

for all Γ, $\Gamma \setminus \boldsymbol{\tau} = \Gamma$;
if $\Gamma = \Delta, x : (\tilde{\tau})^{\downarrow}$, then $\Gamma \setminus x(\tilde{y}) = \Delta, \tilde{y} : \tilde{\tau}$;
if $\Gamma = \Delta, x : (\tilde{\tau})^{\uparrow}$, then $\Gamma \setminus \overline{x}(\tilde{y}) = \Delta, \tilde{y} : \tilde{\tau}$;
if $\Gamma = \Delta, x : (\tilde{\tau})^{!}$, then $\Gamma \setminus x(\tilde{y}) = \Gamma, \tilde{y} : \tilde{\tau}$;
if $\Gamma = \Delta, x : (\tilde{\tau})^{?}$, then $\Gamma \setminus \overline{x}(\tilde{y}) = \Gamma, \tilde{y} : \tilde{\tau}$.

The environment $\Gamma \setminus \alpha$ is what remains after the transition labelled by α has happened. Linear channels are consumed, while replicated channels are not consumed. The new previously bound channels are released.

Then the typed transition, written $P \triangleright \Gamma \xrightarrow{\alpha} Q \triangleright \Gamma'$ is defined by adding the constraint:

$$\frac{P \xrightarrow{\alpha} Q}{P \triangleright \Gamma \xrightarrow{\alpha} Q \triangleright \Gamma \setminus \alpha}$$

Note in particular that the rule applies only when Γ allows α. The above rule does not allow a linear input action and an output action when there is a complementary channel in the process. For example, if a process has $x : (\tilde{\tau})^{!}$ in its action type, then output at x is excluded since such actions can never be observed in a typed context (cf. Section 4.2 and Appendix E of [1]). For a concrete example, consider the process $\overline{a}.b \mid \overline{b}.a$ which is typed in the environment $a : \updownarrow, b : \updownarrow$. Although the process has some untyped transition, none of them is allowed by the environment.

The above rule is well defined in the following sense.

$$\overline{a}(\tilde{y}).P \xrightarrow{\bar{a}(\tilde{y})} P \qquad a(\tilde{y}).P \xrightarrow{a(\tilde{y})} P \qquad !a(\tilde{y}).P \xrightarrow{a(\tilde{y})} P \mid !a(\tilde{y}).P$$

$$\frac{P \xrightarrow{\beta} P' \quad \beta \neq x(\tilde{y}), \overline{x}(\tilde{y})}{(\nu x)P \xrightarrow{\beta} (\nu x)P'} \qquad \frac{P \xrightarrow{\beta} P'}{P \mid Q \xrightarrow{\beta} P' \mid Q} \qquad \frac{Q \xrightarrow{\beta} Q'}{P \mid Q \xrightarrow{\beta} P \mid Q'} \qquad \frac{P \xrightarrow{a(\tilde{y})} P' \quad Q \xrightarrow{\overline{a}(\tilde{y})} Q'}{P \mid Q \xrightarrow{\tau} (\nu \tilde{y})(P' \mid Q')}$$

$$\frac{P \equiv_\alpha P' \quad P \xrightarrow{\alpha} Q}{P' \xrightarrow{\alpha} Q}$$

Figure 2: Labelled Transition System for the πI-Calculus

Proposition 2.1 *If $P \triangleright \Gamma$, $P \xrightarrow{\alpha} Q$ and Γ allows α, then $Q \triangleright \Gamma \setminus \alpha$*

PROOF: by induction on the rules in Figure 2.

Finally we define the notion of typed bisimulation. Let R be a symmetric relation between judgments. We say that R is a bisimulation if the following is satisfied:

- if $(P \triangleright \Gamma) R (P' \triangleright \Gamma')$, then $\Gamma = \Gamma'$;
- whenever $(P \triangleright \Gamma) R (P' \triangleright \Gamma)$, $P \triangleright \Gamma \xrightarrow{\beta} Q \triangleright \Gamma \setminus \beta$, then there exists Q' such that $P' \triangleright \Gamma \xrightarrow{\beta} Q' \triangleright \Gamma \setminus \beta$, and $(Q \triangleright \Gamma \setminus \beta) R (Q' \triangleright \Gamma \setminus \beta)$.

If there exists a bisimulation between two judgments, we say that they are bisimilar $(P \triangleright \Gamma) \approx (P' \triangleright \Gamma)$.

3 Name Sharing CCS

In this section we argue that, in the presence of linear types, α-conversion is not necessary, and channels need not be created dynamically at run time, but they can be decided statically.

3.1 Syntax

We introduce a variant of CCS where synchronisation happens if processes share some "confidential" names. Syntactically this looks like name passing, with the difference that processes decide their confidential names before communicating, and there is not α-conversion. If the chosen names do not coincide, the processes do not synchronise.

The other differences with standard CCS is that we allow infinite parallel composition and infinite restriction. The former is necessary in order to translate replicated processes. The standard intuition in the π-calculus is that the process $!P$ represents the parallel composition of infinitely many copies of P. We need to represent this explicitly in order to be able to provide each copy with different confidential names. Infinite restriction is also necessary, because we need to restrict all confidential names that are shared between two processes in parallel, and these are in general infinitely many.

We call this language Name Sharing CCS, or NCCS.

P	::=	$a\langle\tilde{y}\rangle.P$	input
		$\overline{a}\langle\tilde{y}\rangle.P$	output
		$\prod_{i\in I} P_i$	infinite parallel composition
		$P \setminus S$	infinite restriction
		$\mathbf{0}$	zero

$$\overline{a}\langle\tilde{y}\rangle.P \xrightarrow{\overline{a}\langle\tilde{y}\rangle} P \qquad a\langle\tilde{y}\rangle.P \xrightarrow{a\langle\tilde{y}\rangle} P \qquad \frac{P \xrightarrow{\beta} P' \quad subj(\beta) \notin S}{P \setminus S \xrightarrow{\beta} P' \setminus S}$$

$$\frac{P_n \xrightarrow{\alpha} P'}{\prod_{i\in\mathbb{N}} P_i \xrightarrow{\alpha} (\prod_{i\in\mathbb{N}\setminus\{n\}} P_i) \mid P'} \qquad \frac{P_n \xrightarrow{a\langle\tilde{y}\rangle} P' \quad P_m \xrightarrow{\overline{a}\langle\tilde{y}\rangle} P''}{\prod_{i\in\mathbb{N}} P_i \xrightarrow{\tau} (\prod_{i\in\mathbb{N}\setminus\{n,m\}} P_i) \mid P' \mid P''}$$

Figure 3: Labelled Transition System for Name Sharing CCS

We sometime denote parallel composition where the indexing set is $\{1,2\}$, by $P_1 \| P_2$. As before $\tilde{y}$ denotes a finite sequence of distinct names $y_1 \ldots y_n$, whenever the length of the sequence and the identity of the individual names do not matter. We will also sometimes abuse the notation by using set theoretic notions applied to the sequences. So, for instance, $\tilde{y} \cap \tilde{y}' = \emptyset$ means that for all i, j $y_i \neq y'_j$. S denotes a set of names. Finally, processes are identified up to a straightforward structural congruence, which includes the rule $(P \setminus S) \setminus T \equiv P \setminus (S \cup T)$, but no notion of α-equivalence.

Given an input $\beta = a\langle\tilde{y}\rangle$, we say that a is the subject of β, written $a = subj(\beta)$ while $\tilde{y}$ are the confidential names, written $\tilde{y} = conf(\beta)$. Similarly for output. A name is *confidential* in P if it appears in a confidential position inside P.

The operational semantics is the one of CCS, and it is shown in Figure 3. As in CCS, prefixes generate inputs and outputs. Processes in parallel can proceed independently or synchronise over complementary actions. Restriction inhibits input and output over a particular set of names. The only difference with CCS is the presence of "confidential names" that are used only for synchronisation. Note also that only the subject of an action is taken into account for restriction.

Example. For instance the process

$$(a\langle x\rangle.P \mid \overline{a}\langle y\rangle.R) \setminus \{a\}$$

cannot perform any transition, because x and y do not match. The process

$$(a\langle x\rangle.P \mid \overline{a}\langle x\rangle.Q \mid \overline{a}\langle x\rangle.R) \setminus \{a\}$$

can perform two different initial τ transitions. Since the name x is not bound, it does not become private to the subprocesses involved in the communication.

3.2 Types

Types are generated by the following grammar

σ	$::=$	$(\tilde{y} : \tilde{\sigma})^{\downarrow}$	linear input
	$\mid$	$(\tilde{y} : \tilde{\sigma})^{\uparrow}$	linear output
	$\mid$	$(\{\tilde{y}_h : \tilde{\sigma}_h \mid h \in H\})^{!}$	replicated input
	$\mid$	$(\{\tilde{y}_h : \tilde{\sigma}_h \mid h \in H\})^{?}$	replicated output
τ	$::=$	$\sigma \mid \updownarrow$	closed channel

Linear types specify the tuple of confidential names that are shared during the *only* communication allowed. Replicated types specify a (possibly empty) set[1] of tuples, one for each communication. In the replicated type we assume that for distinct h, h',

[1]Technically we need to use indexed families, up to the name of the indexing set. This is due to the possibly of having many copies of the empty tuple.

$\tilde{y}_h \cap \tilde{y}_{h'} = \emptyset$. A type environment is a finite function from names to types. A "singleton" environment has the form $x : \tau$. We say a name is *confidential* for Γ if it appears inside a type in the range of Γ. A name is *public* if it is in the domain of Γ.

A type environment Γ is well formed, if the following are satisfied:

- confidential names and public names are distinct;
- all confidential names appear exactly once.

In the following we consider only well formed environments. We can perform injective renaming on environments: if σ is an injective endofunction on names which leaves $Dom(\Gamma)$ alone, then $\Gamma[\sigma]$ is the the environment where every confidential name x has been replaced with $\sigma(x)$.

Before introducing the typing rules, we have to define the multiple substitution of names in a replicated output type. For any set K, let $F_K : Names \to \mathscr{P}(Names)$ be a function such that, for every name x, there is a bijection between K and $F_K(x)$. Concretely we can represent $F_K(x) = \{x^k \mid k \in K\}$. In the following we assume that each set K is associated to a unique F_K, and that for distinct x, y, $F_K(x) \cap F_K(y) = \emptyset$.

Given a type τ, and an index k, define τ^k as follows:

- $(\{\tilde{y}_h : \tilde{\tau}_h \mid h \in H\})^{!,k} = (\{\tilde{y}_h^k : \tilde{\tau}_h^k \mid h \in H\})^?$, where $\tilde{y}_h = (y_{i,h})_{i \in I}$ and $\tilde{y}_h^k = (y_{i,h}^k)_{i \in I}$;
- and similarly for all other types.

Given a set K and a replicated output type $\tau = (\{\tilde{y}_h : \tilde{\tau}_h \mid h \in H\})^?$, the type $\tau[K]$ is defined as $(\{\tilde{y}_h^k : \tilde{\tau}_h^k \mid h \in H, k \in K\})^?$. The expression $\tau[K]$ is undefined in all other cases, that is when τ is the uncomposable type, an input or a linear output.

This definition is extended to environments by $\Gamma[K](x) = \Gamma(x)[K]$, which is thus defined only when for every $x \in Dom(\Gamma)$, $MD(\Gamma(x)) = ?$. We will also assume that all names in the range of the substitution are fresh, in the sense that no name in the range of F_K appears in the domain of Γ. Under this assumption we easily have that if Γ is well formed and if $\Gamma[K]$ is defined, then $\Gamma[K]$ is also well formed.

3.3 Matching Confidential Names

We have to define when two types match each other. In the π-calculus two types match each other if they are dual. In the types of NCCS, also confidential names have to match. The matching of two types will also produce a set of names that are to be considered as "closed", as they are used both for input and for output. Finally, after two types have matched, they produce a "residual" type.

We define the relation $match[\tau, \sigma] \to S$, symmetric in the first two arguments, and the partial function $res[\tau, \sigma]$ as follows:

- let $\tau = ((x_i)_{i \in I} : (\tau_i)_{i \in I})^\uparrow$ and $\sigma = ((y_j)_{j \in J} : (\tau_j)_{j \in J})^\downarrow$. Then $match[\tau, \sigma] \to S$ if $I = J$, for all $i \in I$, $x_i = y_i$ and $match[\tau_i, \sigma_i] \to S_i$, and $S = \bigcup_I (S_i \cup \{x_i\})$. In such a case $res[\tau, \sigma] = \updownarrow$;
- let $\tau = (\{(x_i^k)_{i \in I} : (\tau_i^k)_{i \in I} \mid k \in K\})^?$ and $\sigma = (\{(y_j^h)_{j \in J} : (\tau_j^h)_{j \in J} \mid h \in H\})^!$. Then $match[\tau, \sigma] \to S$ if $I = J$, $K \subseteq H$, for all $i \in I, k \in K$, $x_i^k = y_i^k$ and $match[\tau_i^k, \sigma_i^k] \to S_i^k$, and $S = \bigcup_{I,K} (S_i^k \cup \{x_i^k\})$. In such a case $res[\tau, \sigma] = (\{(y_j^h)_J : (\tau_j^h)_J \mid h \in H \setminus K\})^!$.

When two linear types match, the residual type is uncomposable. In the case of the replicated types, the residual type is a replicated input that contains the confidential names not involved in the matching.

$$\frac{}{0 \triangleright \emptyset}\ \text{Zero} \qquad \frac{P \triangleright \Gamma \quad x \notin \Gamma}{P \triangleright \Gamma, x : (\emptyset)^?}\ \text{WeakOut} \qquad \frac{P \triangleright \Gamma \quad x \notin \Gamma}{P \triangleright \Gamma, x : \updownarrow}\ \text{WeakCl}$$

$$\frac{P \triangleright \Gamma, a : \tau \quad MD(\tau) = !, \updownarrow}{P \setminus a \triangleright \Gamma}\ \text{Res}$$

$$\frac{P \triangleright \Gamma, \tilde{y} : \tilde{\tau} \quad a \notin \Gamma \quad \tilde{w}\ \text{fresh}}{\overline{a}\langle \tilde{w} \rangle.P[\tilde{w}/\tilde{y}] \triangleright \Gamma, a : (\tilde{w} : \tilde{\tau})^{\uparrow}}\ \text{LOut} \qquad \frac{P \triangleright \Gamma, \tilde{y} : \tilde{\tau} \quad a \notin \Gamma \quad \tilde{w}\ \text{fresh}}{a\langle \tilde{w} \rangle.P[\tilde{w}/\tilde{y}] \triangleright \Gamma, a : (\tilde{w} : \tilde{\tau})^{\downarrow}}\ \text{LIn}$$

$$\frac{P \triangleright \Gamma, \tilde{y} : \tilde{\tau}, a : (\{\tilde{w}_h : \tilde{\tau}_h \mid h \in H\})^? \quad \tilde{w}_* \text{ fresh}}{\overline{a}\langle \tilde{w}_* \rangle.P[\tilde{w}_*/\tilde{y}] \triangleright \Gamma, a : (\{\tilde{w}_h : \tilde{\tau}_h \mid h \in H \uplus \{*\}\})^?}\ \text{ROut}$$

$$\frac{P \triangleright \Gamma, \tilde{y} : \tilde{\tau} \quad a \notin \Gamma \quad Y = \tilde{y} \cup conf(\Gamma)}{\prod_{k \in K} a\langle \tilde{y}^k \rangle.P[Y^k/Y] \triangleright \Gamma[K], a : (\{\tilde{y}^k : \tilde{\tau}^k \mid k \in K\})^!}\ \text{RIn}$$

$$\frac{P_i \triangleright \Gamma_i \quad (i = 1,2) \quad S = cl(\Gamma_1, \Gamma_2)}{(P_1 \| P_2) \setminus S \triangleright \Gamma_1 \odot \Gamma_2}\ \text{Par}$$

Figure 4: Typing Rules for NCCS

3.4 Typing Rules

We are now ready to write the rules. The main feature of these rules is that, roughly speaking, α-conversion is performed during the typing (at compile time) rather that at run time. The formulation of the rules in this way makes it easier to define the correspondence with the π calculus. Also, parallel composition is well typed only if the names used for communication have matching types, and if the matched names are restricted. This makes sure that communication can happen, and that the shared names are indeed private to the processes involved.

The rules are shown in Figure 4. The environment $\Gamma_1 \odot \Gamma_2 \stackrel{\text{def}}{=} \Gamma$ and the set of names $cl(\Gamma_1, \Gamma_2)$ are jointly defined as follows:

- if $x \notin Dom(\Gamma_1)$, then $\Gamma(x) = \Gamma_2(x)$, $S_x = \emptyset$ and symmetrically;
- if $\Gamma_1(x) = \tau, \Gamma_2(x) = \sigma$ and $match[\tau, \sigma] \to S$, then $\Gamma(x) = res[\tau, \sigma]$ and $S_x = S$;
- if $\Gamma_1(x) = (\{\tilde{y}_k : \tilde{\tau}_k \mid k \in K\})^?$ and $\Gamma_2(x) = (\{\tilde{y}_h : \tilde{\tau}_h \mid h \in H\})^?$ and for every $h \in H, k \in K$ we have $\tilde{y}_k \cap \tilde{y}_h = \emptyset$ and $conf(\tilde{\tau}_k) \cap conf(\tilde{\tau}_h) = \emptyset$ then $\Gamma(x) = (\{\tilde{y}_h : \tilde{\tau}_h \mid h \in H \cup K\})^?$ and $S_x = \emptyset$;
- if any of the other cases arises, then Γ is not defined;
- $cl(\Gamma_1, \Gamma_2) = \bigcup_{x \in Dom(\Gamma_1, \Gamma_2)} S_x$.

3.5 Typed Semantics

The relation Γ allows α is defined similarly to the π-calculus, while the definition of the environment $\Gamma \setminus \alpha$ requires an adaptation to the new types.

- $\Gamma \setminus \tau = \Gamma$;
- if $\Gamma = \Delta, x : (\tilde{y} : \tilde{\tau})^{\downarrow}$, then $\Gamma \setminus x(\tilde{y}) = \Delta, \tilde{y} : \tilde{\tau}$;
- if $\Gamma = \Delta, x : (\tilde{y} : \tilde{\tau})^{\uparrow}$, then $\Gamma \setminus \overline{x}(\tilde{y}) = \Delta, \tilde{y} : \tilde{\tau}$;
- if $\Gamma = \Delta, x : (\{\tilde{y}_h : \tilde{\tau}_h \mid h \in H \uplus \{*\}\})^!$,
  then $\Gamma \setminus x(\tilde{y}_*) = \Delta, \tilde{y}_* : \tilde{\tau}_*, (\{\tilde{y}_h : \tilde{\tau}_h \mid h \in H\})^!$;

- if $\Gamma = \Delta, x : (\{\tilde{y}_h : \tilde{\tau}_h \mid h \in H \uplus \{*\}\})^?$,
  then $\Gamma \setminus \vec{x}(\tilde{y}) = \Delta, \tilde{y}_* : \tilde{\tau}_*, (\{\tilde{y}_h : \tilde{\tau}_h \mid h \in H\})^!$.

Again the typed transition, written $P \triangleright \Gamma \xrightarrow{\alpha} Q \triangleright \Gamma'$ is defined by adding the constraint:

$$\frac{P \xrightarrow{\alpha} Q}{P \triangleright \Gamma \xrightarrow{\alpha} Q \triangleright \Gamma \setminus \alpha}$$

Proposition 3.1 *If $P \triangleright \Gamma$, $P \xrightarrow{\alpha} Q$ and Γ allows α, then $Q \triangleright \Gamma \setminus \alpha$*

The definition of bisimulation and bisimilarity between typed processes is completely analogous to the one of the π-calculus. Bisimilarity is denoted as $P \triangleright \Gamma \approx Q \triangleright \Gamma$.

4 Correspondence between the calculi

4.1 Translation

We are now ready to translate the π-calculus into Name Sharing CCS. The translation is parametrised over a fixed choice for the confidential names. This parametrisation is necessary because π-calculus terms are identified up to α-conversion, and so the identity of bound names is irrelevant, while in Name Sharing CCS, the identity of confidential names is important.

The translation is a family of partial functions $pc[\![-]\!]^\Delta$, indexed by a NCCS type environment Δ, that take a judgment of the π-calculus and return a judgment of NCCS. The functions are only partial because for some choice of names, the parallel composition in NCCS will not be typed.

We define the translation by induction on the derivation of the typing judgment. Without loss of generality, we will assume that all the weakenings are applied to the empty process.

The translation is defined in Figure 5. There, we assume that $pc[\![P \triangleright \Gamma]\!]^\Delta = \hat{P} \triangleright \Delta$, and that $y \in Dom(\Gamma) \Longrightarrow y \in Dom(\Delta)$. In particular, in the translation of the replicated output, we assume $pc[\![P \triangleright \Gamma, a : (\tilde{\tau})^?, \tilde{y} : \tilde{\tau}]\!]^{\Delta, \tilde{y}:\hat{\tilde{\tau}}_*, a:(\{\tilde{w}_i:\hat{\tilde{\tau}}_i \mid i \in I\})^?} = \hat{P} \triangleright \Delta, \tilde{y} : \hat{\tilde{\tau}}_*, a : (\{\tilde{w}_i : \hat{\tilde{\tau}}_i \mid i \in I\})^?$. When the assumptions are not satisfied, the translation is not defined. We also put $Y = conf(\hat{P})$, and $S = cl(\Delta_1 \odot \Delta_2)$. Note the way bound variables become confidential information.

We said that the translation is only a partial function. In particular, for the wrong choice of Δ_1, Δ_2, the translation of the parallel composition could be undefined, because $\Delta_1 \odot \Delta_2$ may be undefined. However it is always possible to find suitable Δ_1, Δ_2. This is the core Lemma of our paper: it is the formalisation of the intuition that "α-conversion can be done at compile time".

Lemma 4.1 *For every judgment $P \triangleright \Gamma$ in the π-calculus, there exists an environment Δ such that $pc[\![P \triangleright \Gamma]\!]^\Delta$ is defined. Moreover, for every injective fresh renaming σ, if $pc[\![P \triangleright \Gamma]\!]^\Delta$ is defined then $pc[\![P \triangleright \Gamma]\!]^{\Delta[\sigma]}$ is defined.*

Example. As an example, consider the process P_ω defined in the introduction

$$P_\omega = \mathtt{Fw}\langle ab \rangle \mid \mathtt{Fw}\langle ba \rangle \mid (\nu z)\overline{a}\langle z \rangle.z.\overline{c} \ .$$

We have $\mathtt{Fw}\langle ab \rangle \triangleright a : ((\,)^\uparrow)^!, b : ((\,)^\downarrow)^?$, so that $P_\omega \triangleright a, b : ((\,)^\uparrow)^!, c : (\,)^\uparrow$. One possible translation for $\mathtt{Fw}\langle ab \rangle \triangleright a : ((\,)^\uparrow)^!, b : ((\,)^\downarrow)^?$ is

$$P_1 = \prod_{k \in K} a\langle x^k \rangle.\overline{b}\langle y^k \rangle.y^k.\vec{x}^k \triangleright a : (\{x^k : (\,)^\uparrow \mid k \in K\})^!, b : (\{y^k : (\,)^\downarrow \mid k \in K\})^?$$

$$
\begin{array}{rcl}
pc[\![0 \triangleright x_i : (\tau_i)^?, y_j : \updownarrow]\!]^{x_i:(\emptyset)^?, y_j:\updownarrow} & = & 0 \triangleright x_i : (\emptyset)^?, y_j : \updownarrow \\
pc[\![(\nu a)P \triangleright \Gamma]\!]^{\Delta} & = & \hat{P} \setminus a \triangleright \Delta \\
pc[\![\overline{a}(\tilde{y}).P \triangleright \Gamma, a : (\tilde{\tau})^{\uparrow}]\!]^{\Delta, a:(\tilde{z}:\hat{\tilde{\tau}})^{\uparrow}} & = & \overline{a}\langle \tilde{z} \rangle.\hat{P}[\tilde{z}/\tilde{y}] \triangleright \Delta, a : (\tilde{z} : \hat{\tilde{\tau}})^{\uparrow} \\
pc[\![a(\tilde{y}).P \triangleright \Gamma, a : (\tilde{\tau})^{\downarrow}]\!]^{\Delta, a:(\tilde{z}:\hat{\tilde{\tau}})^{\downarrow}} & = & a\langle \tilde{z} \rangle.\hat{P}[\tilde{z}/\tilde{y}] \triangleright \Delta, a : (\tilde{z} : \hat{\tilde{\tau}})^{\downarrow} \\
pc[\![!a(\tilde{y}).P \triangleright \Gamma, a : (\tilde{\tau})^{!}]\!]^{\Delta[K], a:(\{\tilde{y}^k:\hat{\tilde{\tau}}^k \mid k \in K\})^{!}} & = & \\
\multicolumn{3}{c}{\prod_{k \in K} a\langle \tilde{y}^k \rangle.\hat{P}[\tilde{y}^k/\tilde{y}][Y^k/Y] \triangleright \Delta[K], a : (\{\tilde{y}^k : \hat{\tilde{\tau}}^k \mid k \in K\})^{!}} \\
pc[\![\overline{a}(\tilde{y}).P \triangleright \Gamma, a : (\tilde{\tau})^{?}]\!]^{\Delta, a:(\{\tilde{w}_i:\tilde{\tau}_i \mid i \in I \uplus \{*\}\})^{?}} & = & \\
\multicolumn{3}{c}{\overline{a}\langle \tilde{w}_* \rangle.\hat{P}[\tilde{w}_*/\tilde{y}] \triangleright \hat{\Gamma}, a : (\{\tilde{w}_i : \tilde{\tau}_i \mid i \in I \uplus \{*\}\})^{?}} \\
pc[\![P_1 \| P_2 \triangleright \Gamma_1 \odot \Gamma_2]\!]^{\Delta_1 \odot \Delta_2} & = & (\hat{P_1} \| \hat{P_2}) \setminus S \triangleright \Delta_1 \odot \Delta_2
\end{array}
$$

Figure 5: Translation from π to NCCS

while for $\mathtt{Fw}\langle ba \rangle \triangleright b : ((\,)^{\uparrow})^{!}, a : ((\,)^{\downarrow})^{?}$ is

$$P_2 = \prod_{h \in H} b\langle z^h \rangle.\overline{a}\langle w^h \rangle.w^h.\overline{z}^h \triangleright b : (\{z^h : ()^{\uparrow} \mid h \in H\})^{!}, a : (\{w^h : ()^{\downarrow} \mid h \in H\})^{?}$$

Assuming there are two "synchronising" injective functions $f : K \to H, g : H \to K$, such that $y^k = z^{f(k)}, w^h = x^{g(h)}$ (if not, we can independently perform a fresh injective renaming on both environments), we obtain that the corresponding types for a, b match, so that we can compose the two environments. Finally pick one $\hat{k} \in K \setminus g(H)$. We obtain the following translation of P_ω:

$$((P_1 \mid P_2) \setminus S \mid \overline{a}\langle x^{\hat{k}} \rangle.x^{\hat{k}}.\overline{c}) \setminus \{x^{\hat{k}}\}$$

where $S = \{y^k \mid k \in K\} \cup \{w^h \mid h \in H\}$. The above process is typed in the following environment

$$a : (\{x^k : ()^{\downarrow} \mid k \in K \setminus (g(H) \cup \{\hat{k}\})\})^{!}, b : (\{z^h : ()^{\downarrow} \mid h \in H \setminus f(K)\})^{!}, c : ()^{\uparrow} \ .$$

The reader can check that any transition of P_ω is matched by a corresponding transition of its translation. This is what we formally show next.

4.2 Adequacy

To show the correctness of the translation, we first prove the correspondence between the labelled transition semantics.

Theorem 4.2 *Suppose $P \triangleright \Gamma \xrightarrow{\beta} P' \triangleright \Gamma'$ in the π-calculus, and that $pc[\![P \triangleright \Gamma]\!]^{\Delta}$ is defined. Then for every injective fresh renaming σ $pc[\![P \triangleright \Gamma]\!]^{\Delta[\sigma]} \xrightarrow{\beta[\sigma]} pc[\![P' \triangleright \Gamma']\!]^{\Delta[\sigma] \setminus \beta[\sigma]}$.*

Conversely, suppose $pc[\![P \triangleright \Gamma]\!]^{\Delta} \xrightarrow{\beta} Q \triangleright \Delta \setminus \beta$. Then there exists P' such that $P \triangleright \Gamma \xrightarrow{\beta} P' \triangleright \Gamma \setminus \beta$ and $pc[\![P' \triangleright \Gamma \setminus \beta]\!]^{\Delta \setminus \beta} = Q \triangleright \Delta \setminus \beta$.

Moreover we have that bisimilar terms are mapped into bisimilar terms.

Theorem 4.3 (Full Abstraction) *Suppose $P \triangleright \Gamma \approx P' \triangleright \Gamma$ holds in the π-calculus. Then for any Δ such that $pc[\![P \triangleright \Gamma]\!]^{\Delta}$, and $pc[\![P' \triangleright \Gamma]\!]^{\Delta}$ are defined we have $pc[\![P \triangleright \Gamma]\!]^{\Delta} \approx pc[\![P' \triangleright \Gamma]\!]^{\Delta}$.*

Conversely, suppose that for some Δ, $pc[\![P \triangleright \Gamma]\!]^{\Delta} \approx pc[\![P' \triangleright \Gamma]\!]^{\Delta}$. Then $P \triangleright \Gamma \approx P' \triangleright \Gamma$.

5 Conclusion

Extensions The above analysis considers a simple but expressive fragment of the π-calculus. The linear typing system can be applied to more general frameworks. For example we could allow the existence of a nondeterministic internal sum $P \oplus Q$ such that

$$P \oplus Q \xrightarrow{\tau} P \quad \text{or} \quad P \oplus Q \xrightarrow{\tau} Q$$

This process is well has type Γ only when both P and Q have the same type Γ. The translation into the corresponding extension of NCCS makes sure that on both sides of the choice, corresponding channels are assigned the same confidential names.

Another addition consists in an operator of external choice, or *branching* [23, 8, 1, 25, 9, 26]. The input prefix provides different continuations, $a(\tilde{y}) \sum_{i \in I} \mathtt{in}_i.P_i$, while the output is selecting one of the branches $\overline{a}(\tilde{y})\mathtt{in}_j.P$. The reduction semantics of the branching is defined by the following rule:

$$a(\tilde{y}) \sum_{i \in I} \mathtt{in}_i.P_i \,|\, \overline{a}(\tilde{y})\mathtt{in}_j.Q \xrightarrow{\tau} (\nu\, \tilde{y})(P_j \,|\, Q)$$

The type system is extended in a straightforward way, in order to make sure that we only compose a selection with the proper branching. Similarly to the internal choice, the branching input is typed only when all its continuations have the same type. Again the translation into NCCS makes sure that on all branches, corresponding channels are assigned the same confidential names.

Related and Future work This work arises in the context of a wider program aiming at applying causal model of concurrency, such as Event Structures [24], or Petri nets to the π-calculus. In causal models, behavioural properties such as confluence and sequentiality have an intuitive and simple definition. For instance, in Event Structures, confluence amounts to *conflict freeness*.

Event structures were tailored around early models for concurrency, where dynamic creation of new names was not an issue. By reducing the linear π-calculus to a variant of CCS, we have made the first step towards an Even Structure model of the linear π-calculus. In turn, since we can encode functional programming languages into the linear π-calculus, this will allow us to give a causal (or "true concurrent") interpretation of the λ-calculus. This line of research is already being pursued, from a completely different starting point by Melliès [14], and it would interesting to compare the resulting theory with the one in [14].

Another direction of research aims at extending the linear π-calculus with some form of probabilistic choice. The starting point of this research is the notion of probabilistic Event Structures, by Varacca,Völzer and Winskel [22]. The study carried out in [22] suggests that in the probabilistic framework, the proper notion of confluence is the notion of *confusion freeness* (well known in the Petri net community [17, 3]). It seems difficult to express probabilistic confluence within interleaving models for probability, such as Segala's automata [21], used by Herescu and Palamidessi in their work on the probabilistic π-calculus [7]. A proper notion of probabilistic linear π-calculus could provide insight on the different possible ways of adding probabilistic choice to functional paradigms, such as, for instance, the probabilistic λ-calculus proposed by Di Pierro, Hankin and Wiklicky [4].

There are other works specifically addressing α-conversion in the literature. Ferrari, Montanari and Quaglia [6] study a version of the π-calculus with explicit substitutions. Their system allows them to produce a semantics using SOS rules only (the standard α-conversion rule is not SOS). Fernandez, Gabbay and Mackie [5] address the α-conversion rule in the context of nominal rewriting. In both cases, α-conversion is still performed dynamically during the computation, and not statically by a typing system.

References

[1] Martin Berger, Kohei Honda, and Nobuko Yoshida. Sequentiality and the π-calculus. In *Proc. TLCA'01*, volume 2044 of *LNCS*, pages 29–45, 2001.

[2] Martin Berger, Kohei Honda, and Nobuko Yoshida. Genericity and the π-Calculus. In *Proc. FOSSACS'03*, number 2620 in LNCS, pages 103–119. Springer, 2003. The full version is to appear in *Journal of ACM Acta Informatica*, 2005.

[3] Jörg Desel and Javier Esparza. *Free Choice Petri Nets*. Cambridge University Press, 1995.

[4] Alessandra Di Pierro, Chris Hankin, and Herbert Wiklicky. Probabilistic lambda calculus and quantitative program analysis. *Journal of Logic and Computation*, 2005. To appear.

[5] Maribel Fernandez, Murdoch J. Gabbay, and Ian Mackie. Nominal rewriting systems. In *PPDP04*. ACM Press, 2004.

[6] Gian Luigi Ferrari, Ugo Montanari, and Paola Quaglia. A pi-calculus with explicit substitutions. *Theoretical Computer Science*, 168(1):53–103, 1996.

[7] Mihaela Herescu and Catuscia Palamidessi. Probabilistic asynchronous π-calculus. In *Proceedings of 3rd FoSSaCS*, volume 1784 of *LNCS*, pages 146–160. Springer, 2000.

[8] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type disciplines for structured communication-based programming. In *ESOP'98*, volume 1381 of *LNCS*, pages 22–138. Springer, 1998.

[9] Kohei Honda and Nobuko Yoshida. A uniform type structure for secure information flow. In *POPL'02*, pages 81–92. ACM Press, 2002. Full version available at www.doc.ic.ac.uk/˜yoshida.

[10] Kohei Honda and Nobuko Yoshida. Noninterference Through Flow Analysis. *Journal of Functional Programming*, 15(2):293–349, 2005.

[11] Kohei Honda, Nobuko Yoshida, and Martin Berger. Control in the π-calculus. In *Proc. CW'04*. ACM Press, 2004.

[12] Naoki Kobayashi. A type system for lock-free processes. *Information and Computation*, 177(2):122–159, 2002.

[13] Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. Linearity and the Pi-Calculus. *ACM Transactions on Programming Languages and Systems*, 21(5):914–947, 1999.

[14] Paul-André Melliès. Asynchronous games 2: The true concurrency of innocence. In Philippa Gardner and Nobuko Yoshida, editors, *CONCUR*, volume 3170 of *LNCS*, pages 448–465. Springer, 2004.

[15] Massimo Merro. *Locality in the pi-calculus and applications to distributed objects*. PhD thesis, INRIA Sophia-Antipolis and EU Marie Curie TMR, 2004.

[16] Robin Milner. *Communicating and Mobile Systems: The Pi Calculus*. Cambrige University Press, 1999.

[17] Grzegorz Rozenberg and P.S. Thiagarajan. Petri nets: Basic notions, structure, behaviour. In *Current Trends in Concurrency*, volume 224 of *LNCS*, pages 585–668. Springer, 1986.

[18] Davide Sangiorgi. Internal mobility and agent passing calculi. In *Proc. ICALP'95*, volume 944 of *LNCS*, pages 672–683. Springer, 1995.

[19] Davide Sangiorgi. πI: A symmetric calculus based on internal mobility. In *Proc. TAPSOFT'95*, volume 915 of *LNCS*, pages 172–186. Springer, 1995.

[20] Davide Sangiorgi. The name discipline of uniform receptiveness. In *ICALP'97*, volume 1256 of *LNCS*, pages 303–313. Springer, 1997.

[21] Roberto Segala. *Modeling and Verification of Randomized Distributed Real-Time Systems*. PhD thesis, M.I.T., 1995.

[22] Daniele Varacca, Hagen Völzer, and Glynn Winskel. Probabilistic event structures and domains. In *Proceedings of 15th CONCUR*, volume 3170 of *LNCS*, pages 481–496. Springer, 2004.

[23] Vasco Vasconcelos. Typed concurrent objects. In *Proc. ECOOP'94*, volume 821 of *LNCS*, pages 100–117. Springer, 1994.

[24] Glynn Winskel and Mogens Nielsen. Models for concurrency. In *Handbook of logic in Computer Science*, volume 4. Clarendon Press, 1995.

[25] Nobuko Yoshida, Martin Berger, and Kohei Honda. Strong Normalisation in the π-Calculus. In *LICS'01*, pages 311–322. IEEE, 2001. The full version in *Information and Computation*., 191 (2004) 145–202, Elsevier.

[26] Nobuko Yoshida, Kohei Honda, and Martin Berger. Linearity and bisimulation. In *FoSSaCs02*, volume 2303 of *LNCS*, pages 417–433. Springer, 2002.