# Typed $\pi$-calculus at work: a proof of Jones's parallelisation transformation on concurrent objects

Davide Sangiorgi
INRIA-Sophia Antipolis
B.P. 93 F-06902 Sophia Antipolis, France
email: davide.sangiorgi@sophia.inria.fr

December 16, 1996

**Abstract**

Cliff Jones [Jon93a] has raised the challenge of how to prove the validity of a certain transformation that increases the concurrent activity in a system of concurrent objects. We present a proof of this transformation that uses a *typed* $\pi$-calculus and *typed behavioural equivalences*. Our type system is that for *receptiveness*; it guarantees that the input-end of certain channels is always ready to receive messages (at least as long as there are processes that could send such messages), and that all messages will be processed using the same continuation. This work is also intended as an example of the usefulness of $\pi$-calculus types for reasoning.

## 1 Introduction

In his invited lecture at FOOL'96, as well as in papers like [Jon93a], Cliff Jones has given examples of transformations on concurrent objects and has raised the challenge of how to prove their validity. They are transformations between concrete programs as well as general transformation rules. In this paper we show a proof of correctness of the hardest of Jones's concrete transformations, for which we exploit a *typed $\pi$-calculus*.

The transformation involves two classes representing a sequential and a parallel implementation of a symbol table. The table keeps an association of strings to integers (the keys). It allows an operation, called insertion, that has a string $t$ as parameter. As a result, the table returns an integer $n$ to the invoking object; $n$ is the integer associated with $t$ in the table, if there is such an integer, is a new one otherwise, in which case the pair $(t, n)$ is stored in the table. In accordance with the semantics of method invocation in the language $\pi o\beta\lambda$ used by Jones, the activity of an invoking object is suspended until it receives the result. The two tables are implemented as a chain of cells each of which has a storage capacity for one pair (string, integer). Only the first cell of the chain can communicate with the environment. If necessary, the last cell can create a new cell which then becomes the last cell of the chain. The integers stored in the cells of the chain are in increasing order.[1] Here is the class whose instances are the cells for the sequential implementation:

class $Seq(n$:int, $s$:string$)$,
   var $Next$:ref$(Seq) \leftarrow nil$

[1] To keep the example as simple as possible while retaining all its challenge, we have made some simplifications over Jones's original problem: our table allows only one operation; the implementations have a chain, rather than tree, structure; the integer and string stored in a cell are supplied at creation time, as parameters in the class definition.

```
   method insert(t:string):int,
      if  s = t   then return n
                  else if  Next = nil then (Next ←new(Seq(n+1,t));
                                               return (n+1) )
                                       else return Next!insert(t)
```

Variable *Next* is a reference to the successive cell in the chain. Command *Next* $\leftarrow$ `new(`*Seq*`(`$n{+}1$`,`$t$`))` creates a new cell which stores integer $n{+}1$ and string $t$; expression *Next*`!`*insert*$(t)$ invokes an insertion on the successive cell, and the result obtained is the value of this expression.

The behaviour of a cell and, hence, of the chain are strictly sequential. An insertion request for a string $t$ ripples down the chain until the result has been found, that is either a cell which stores $t$ is met, or the end of the chain is reached. Then the result ripples up the chain to the first cell where is finally delivered to the client. Between accepting the insertion and returning the result, the table cannot handle other requests. Here is the class definition for the parallel implementation:

```
class Par(n:int, s:string),
   var Next:ref(Seq) ← nil
   method insert(t:string):int,
      if s = t    then return n
                  else if  Next = nil then ( return (n+1);
                                               Next ←new(Seq(n+1,t)) )
                                       else delegate Next!insert(t)
```

There are two modifications: the second `return` is anticipated; the last `return` is replaced by a `delegate`, with which the cell *delegates* to the next one the task of delivering the result. Both transformations (especially the second) introduce potential parallelism. The first frees a client from the rendez-vous before the activity of the table is completed. The second has the consequences that (1) the result of an insertion need not ripple up to the first cell of the chain, and (2) the table can process several insertions at the same time, since it can accept a new request even if the result of the previous one has not been delivered yet.

Although *Par* is more efficient than *Seq* they are *behaviourally equivalent*, since their difference is not observable under any $\pi o\beta\lambda$ context. The challenge is: how to prove it? Jones suggested to use the $\pi$-calculus because it has a rich algebraic theory and it allows simple interpretations of $\pi o\beta\lambda$ [Jon93b, Wal95b]. But the translations of *Seq* and *Par*, say $[\![Seq]\!]$ and $[\![Par]\!]$, are *not* behaviourally equivalent in the ordinary $\pi$-calculus, *even w.r.t the very weak notion of trace equivalence*. Indeed, a proof of equality between $[\![Seq]\!]$ and $[\![Par]\!]$ is interesting not only at the $\pi o\beta\lambda$ level but also at the $\pi$-calculus level — the challenge here is understanding *in what sense* are $[\![Seq]\!]$ and $[\![Par]\!]$ equivalent.

We prove that $[\![Seq]\!]$ and $[\![Par]\!]$ are equivalent by moving to a *typed* $\pi$-calculus. The type system is that for *uniform receptiveness*, briefly receptiveness, recently studied by the author [San96a]. Imposing a type system has the advantage of cutting down the number of legal contexts. With the type system for receptiveness, the remaining contexts are unable to separate $[\![Seq]\!]$ from $[\![Par]\!]$. A very important property of receptiveness is that the resulting typed $\pi$-calculus has a tractable theory, based on simple extensions of the proof techniques of the ordinary $\pi$-calculus. We exploit this theory, of which we give a brief account in Section 3, to prove that $[\![Seq]\!]$ and $[\![Par]\!]$ are behaviourally equivalent.

Liu and Walker have already given $\pi$-calculus proofs of this transformation [LW95]. Our proof is shorter and, we hope, easier to read. We overcome the problem of the inequality between $[\![Seq]\!]$ and $[\![Par]\!]$ in the ordinary $\pi$-calculus by adding types, which capture appropriate behavioural properties of the legal contexts, and then exploiting the theory of the typed $\pi$-calculus. By contrast, Liu and Walker proceed in a purely operational way, that is, using the transitional semantics of the ordinary $\pi$-calculus to state the properties and to prove that they are satisfied by all translations of $\pi o\beta\lambda$ contexts; exploiting these properties, they are able to show that $[\![Seq]\!]$ and $[\![Par]\!]$ cannot be distinguished in translations of $\pi o\beta\lambda$ contexts. Recently, Philippou and Walker have gone forth and proved a general transformation rule on $\pi o\beta\lambda$ program [PW96]; it remains to be seen whether receptiveness can be helpful also in the validation of this rule.

## 2 $\pi$-calculus: general background

We use small letters $p, q, r, \ldots$, to range over $\pi$-calculus names, and capitol letters $P, Q, R$ to range over $\pi$-calculus processes. Below is the $\pi$-calculus grammar; the operators are those of inaction, input and output prefix, conditional, parallel composition, restriction and replication:

$$\begin{array}{rcl} P & := & 0 \;\Big|\; p(\tilde{q}).\,P \;\Big|\; \overline{p}\langle\tilde{q}\rangle.\,P \;\Big|\; \texttt{if } p = q \texttt{ then } P_1 \texttt{ else } P_2 \\ & & \;\Big|\; P_1 \mid P_2 \;\Big|\; \boldsymbol{\nu} p\, P \;\Big|\; !p(\tilde{q}).\,P \end{array}$$

A tilde denotes a tuple. When the tilde is empty, the surrounding brackets $()$ and $\langle\rangle$ will be omitted. We use $\alpha$ to range over prefixes. We refer to [Mil91] for detailed discussions on the operators of the language. We only remind that a replication $!p(\tilde{q}).\,P$ means, intuitively, that an arbitrary number of copies of $p(\tilde{q}).\,P$ are available. We assign parallel composition the lowest precedence among the operators. An input $p(\tilde{q}).\,P$ and a restriction $\boldsymbol{\nu} q\, P$ bind all free occurrences of names $\tilde{q}$ and $q$ in $P$. These binders give rise in the expected way to the definitions of *free* and *bound* names of a term. The definitions of substitution and alpha conversion are standard too, with renaming possibly involved to avoid captures.

We omit the labeled transition system of the $\pi$-calculus. We only recall that transitions are of the form $P \xrightarrow{\mu} P'$, where the action $\mu$ can be of three forms: $\tau$ (meaning that $P$ can evolve to $P'$ through some internal activity), $p\langle\tilde{q}\rangle$ (meaning that $P$ can perform an input at $p$ in which names $\tilde{q}$ are received, and then it becomes $P'$), $\boldsymbol{\nu}\tilde{r}\,\overline{p}\langle\tilde{q}\rangle$ (meaning that $P$ can emit names $\tilde{q}$ at $p$ and become $P'$ and, among these names $\tilde{q}$, there are the private names $\tilde{r}$). In the input and output actions, $p$ is the *subject*. In prefixes and actions, we write $\overline{p}(q)$ for $\boldsymbol{\nu} q\,\overline{p}\langle q\rangle$ (output of a private name).

It is important a discipline to avoid disagreements in what is carried by a given name; for instance we reject expressions like $\overline{p}\langle q_1, q_2\rangle.\,P \mid p(r).\,Q$, due to the arity mismatching in the use of the names. To this purpose, Milner proposed the use of a *sorting system* [Mil91]. In this system, names are partitioned into a collection of *sorts*. Then a sorting function is defined which maps sorts onto sequences of sorts. If a sort $s$ is mapped onto sorts $(t_1, t_2)$ this means that names in $s$ may only carry pairs of names, where the first component is a name of $t_1$, the second a name of $t_2$. In the remainder of the paper we assume that there is a sorting that all processes obey.

*Barbed congruence* [MS92] equates, intuitively, processes that have the same convergence behaviour in all contexts. Its definition uses relation $\Longrightarrow$, the reflexive and transitive closure of $\xrightarrow{\tau}$, and an observation predicate $\Downarrow_p$ that, for each port $p$, detects the possibility of performing a communication with the external environment along $p$. Thus, $P \Downarrow_p$ holds if there are $P', P''$ and an action $\mu$ with subject $p$ s.t. $P \Longrightarrow P' \xrightarrow{\mu} P''$

**Definition 1 (barbed congruence)** Barbed bisimulation *is the largest symmetric relation* $\dot{\approx}$ *on processes s.t.* $P \dot{\approx} Q$ *implies:*

1. *if* $P \Longrightarrow P'$ *then there exists* $Q'$ *such that* $Q \Longrightarrow Q'$ *and* $P' \dot{\approx} Q'$*;*

*2. for each name $p$, $P \Downarrow_p$ iff $Q \Downarrow_p$.*

*$P$ and $Q$ are* barbed congruent, *written $P \approx Q$, if in all contexts $C$ it holds that $C[P] \stackrel{\bullet}{\approx} C[Q]$.*

The laws below are valid for barbed congruence [Mil91, San92]; (1-4) show some useful distributivity properties of private replications; (5) is similar to $\pi$-calculus second $\tau$-law.

**Lemma 2** *Suppose $r$ occurs free in $P, R, Q$ only as subject of output prefixes. Then:*

1. $\boldsymbol{\nu} r\,(P \mid Q \mid !r(\tilde{r}).\,R) \approx \boldsymbol{\nu} r\,(P \mid !r(\tilde{r}).\,R) \mid \boldsymbol{\nu} r\,(Q \mid !r(\tilde{r}).\,R)$;
2. $\boldsymbol{\nu} r\,((!p(\tilde{p}).\,P) \mid !r(\tilde{r}).\,R) \approx !p(\tilde{p}).\,\boldsymbol{\nu} r\,(P \mid !r(\tilde{r}).\,R)$, *if $p \neq r$ and no name in $\tilde{p}$ occurs free in $!r(\tilde{r}).\,R$;*
3. $\boldsymbol{\nu} r\,((\ \texttt{if}\ p = q\ \texttt{then}\ P\ \texttt{else}\ Q) \mid !r(\tilde{r}).\,R) \approx$ $\texttt{if}\ p = q\ \texttt{then}\ (\boldsymbol{\nu} r\,(P \mid !r(\tilde{r}).\,R))\ \texttt{else}\ (\boldsymbol{\nu} r\,(Q \mid !r(\tilde{r}).\,R))$;
4. $\boldsymbol{\nu} r\,((\boldsymbol{\nu} q\, P) \mid !r(\tilde{r}).\,R) \approx \boldsymbol{\nu} q\, \boldsymbol{\nu} r\,(P \mid !r(\tilde{r}).\,R)$, *if $q$ does not occur free in $r(\tilde{r}).\,R$;*
5. $\boldsymbol{\nu} r\,(P \mid !r(\tilde{r}).\,R) \approx P$, *if $r$ does not appear in $P$;*
6. $\alpha.\,\boldsymbol{\nu} r\,(\overline{r}\langle\tilde{q}\rangle.\,P \mid !r(\tilde{r}).\,R) \approx \alpha.\,(P \mid R\{\tilde{q}/\tilde{r}\} \mid !r(\tilde{r}).\,R)$

# 3 Uniform receptiveness: general background

We review uniform receptiveness and some of its algebraic properties. We refer to [San96a] for more details.

A name $x$ is receptive in a process $P$ if at any time $P$ is able of offering an input at $x$ (at least as long as there are processes that could send messages at $x$). The receptiveness of $x$ is uniform if all inputs at $x$ have the same continuation. Receptiveness ensures that any message sent at $x$ can be immediately processed; uniformity ensures that there is a unique way in which a message at $x$ may be processed (that is, the input-end of $x$ is "functional").

These are *semantic* conditions, and are undecidable. To obtain decidable conditions we therefore impose some restrictions. Roughly, we guarantee receptiveness by demanding that the name is available in input-replicated form as soon as created. For instance, $x$ is receptive in

$$P_1 \stackrel{\text{def}}{=} \boldsymbol{\nu} x\,\Big(!x(p).\,P \mid Q\Big) \qquad\qquad P_2 \stackrel{\text{def}}{=} \boldsymbol{\nu} x\,\Big(\overline{r}\langle x\rangle.\,!x(p).\,P\Big) \tag{1}$$

(On the left, name $x$ is created when the external output $\overline{r}\langle x\rangle$ is consumed since, before this, $x$ is frozen.) We guarantee uniformity by demanding that there is only one input occurrence of the name; hence in (1) name $x$ should not occur free in input position in $P$ and $Q$. To preserve this property in a network of processes, we then also demand that only the output capability of the name may be transmitted; that is, as all $\pi$-calculus names, so receptive names can be transmitted but, in contrast with the other names, the recipient of a receptive name can use it only in output (retransmitting the name, or sending a message at it).

In the processes $P_1$ and $P_2$ above, the receptiveness at $x$ is persistent, which is necessary if unboundedly many messages could be sent at $x$. It is useful to consider separately the case in which at most one message can sent. Then the replication in front of the input at $x$ is unnecessary. We call the first form *$\omega$-receptiveness*, the second *linear receptiveness*. The above explanation will

$$\text{(T-INP-1)}\ \frac{\emptyset;\Gamma \vdash_{\mathrm{L}} P}{\emptyset;\Gamma \vdash_{\mathrm{L}} a(b).\,P} \qquad \text{(T-INP-2)}\ \frac{x \notin \Gamma \qquad \emptyset;\Gamma, x \vdash_{\mathrm{L}} P}{\emptyset;\Gamma \vdash_{\mathrm{L}} a(x).\,P}$$

$$\text{(T-INP-3)}\ \frac{x \notin \Gamma \qquad \emptyset;\Gamma \vdash_{\mathrm{L}} P}{\{x\};\Gamma \vdash_{\mathrm{L}} x(b).\,P} \qquad \text{(T-REP)}\ \frac{\emptyset;\emptyset \vdash_{\mathrm{L}} a(b)P}{\emptyset;\emptyset \vdash_{\mathrm{L}} !a(b).\,P}$$

$$\text{(T-RES-1)}\ \frac{x \notin \Gamma \cup \Delta \qquad \Delta, x;\Gamma, x \vdash_{\mathrm{L}} P}{\Delta;\Gamma \vdash_{\mathrm{L}} \boldsymbol{\nu} x\, P} \qquad \text{(T-RES-2)}\ \frac{p \notin \Gamma \cup \Delta \qquad \Delta;\Gamma \vdash_{\mathrm{L}} P}{\Delta;\Gamma \vdash_{\mathrm{L}} \boldsymbol{\nu} p\, P}$$

$$\text{(T-OUT-1)}\ \frac{\emptyset;\Gamma \vdash_{\mathrm{L}} P}{\emptyset;\Gamma \vdash_{\mathrm{L}} \overline{a}\langle b\rangle.\,P,\ \ \overline{a}(b).\,P} \qquad \text{(T-OUT-2)}\ \frac{\emptyset;\Gamma \vdash_{\mathrm{L}} P}{\emptyset;\Gamma, x \vdash_{\mathrm{L}} \overline{a}\langle x\rangle.\,P}$$

$$\text{(T-OUT-3)}\ \frac{\emptyset;\Gamma \vdash_{\mathrm{L}} P}{\emptyset;\Gamma, x \vdash_{\mathrm{L}} \overline{x}\langle b\rangle.\,P,\ \ \overline{x}(b).\,P} \qquad \text{(T-BOUT)}\ \frac{x \notin \Gamma \qquad \{x\};\Gamma \vdash_{\mathrm{L}} P}{\emptyset;\Gamma \vdash_{\mathrm{L}} \overline{a}(x).\,P}$$

$$\text{(T-PAR)}\ \frac{\Delta_1;\Gamma_1 \vdash_{\mathrm{L}} P_1 \qquad \Delta_2;\Gamma_2 \vdash_{\mathrm{L}} P_2}{\Delta_1, \Delta_2;\Gamma_1, \Gamma_2 \vdash_{\mathrm{L}} P_1 \mid P_2} \qquad \text{(T-COND)}\ \frac{\emptyset;\Gamma \vdash_{\mathrm{L}} P \qquad \emptyset;\Gamma \vdash_{\mathrm{L}} Q}{\emptyset;\Gamma \vdash_{\mathrm{L}}\ \texttt{if}\ a = b\ \texttt{then}\ P\ \texttt{else}\ Q}$$

$$\text{(T-NIL)}\ \frac{}{\emptyset;\emptyset \vdash_{\mathrm{L}} 0}$$

Table 1: The typing rules for linear receptiveness.

be enough to understand our use of receptiveness in later sections. For the interested reader, in the remainder of this subsection we discuss the formal system for receptiveness and the associated soundness theorem.

The discipline of receptiveness can be added to any of the main type systems for the $\pi$-calculus. The simplest way is to add it on top of Milner's *sorting* — historically the first of such type systems. In Table 3, we show the typing rules for linear receptiveness, assuming that processes are already well-sorted. The rules are a refinement of those of the type system with linearity [KPT96]. For simplicity, we present them on the monadic calculus (where only one name at a time can be carried), and we assume that: There is a single sort of linear receptive names; linear receptive names carry names that are not linear receptive. These assumptions can be relaxed without difficulties. In the table, $x, y, \ldots$ range over linear receptive names; $a, b, ..$ over names that are not linear receptive; and $p, q, r$ over the set of all names. Judgements have the form $\Delta;\Gamma \vdash_{\mathrm{L}} P$. A typing context is a pair $(\Delta;\Gamma)$ of finite sets of linear receptive names. As sets, the order in which names appear in $\Delta$ and $\Gamma$ does not matter. Intuitively, if $\Delta;\Gamma \vdash_{\mathrm{L}} P$ then $\Delta \cup \Gamma$ are the only linear receptive names which appear free in $P$; process $P$ must use any name in $\Gamma$ *exactly once* in output position (that is, either performing an output at that name or transmitting this capability to another process), and names in $\Delta$ *immediately* and *only once* in input. This intuition is formalised in Theorem 3, which relates types and operational semantics of processes. We write $\Delta - x$ for $\Delta - \{x\}$ and $\Delta, x$ for $\Delta \cup \{x\}$.

Some remarks on the rules: A rule with double conclusion is an abbreviation for the two rules with same premises but separate conclusions. The output capability on a receptive name is transmitted in rule T-OUT-2 and is used in rule T-OUT-3. In rule T-BOUT, the output capability on a newly-created receptive name $x$ is transmitted: Therefore $x$ cannot appear in output position in the continuation $P$, but, since receptive, $x$ must be available in input in $P$. In rule T-PAR, the typing context is split so to type the two components $P_1$ and $P_2$; splitting of the typing is usual in type systems which forbid autoconcurrency on names [KPT96, Hon96]. Test for equality is allowed on plain names (rule T-COND), but not on receptive names; this is typical of type systems where the input and output capabilities on names are separate [PS93] (in this rule, the constraint that the first component of the type environment be empty can be relaxed but it would complicate some of the theory).

The theorem below is the semantic justification of the typing rules. Some explanation of its clauses: (2) says that any receptive name received in an input must be used in output only. Clauses (6) and (7) say that the capability of using a receptive name in output is lost after one such action; moreover if $P$ is creating $x$ then $P$ must guarantee the input-availability of $x$. Clause (1) and (3) show that if $x$ is receptive in $P$ then $P$ is always ready to perform an input at $x$ and, moreover, it can perform exactly one such input. Clause (5) talks about interactions; it has two conclusions, depending on whether or not the interaction occurs at a receptive name free in the process; if so then the name is not free in the derivative.

**Theorem 3 (soundness theorem for linear receptiveness)** *Suppose* $\Delta; \Gamma \vdash_{\mathrm{L}} P$.

1. *if* $x \in \Delta$ *then for all* $a$ *there is a unique* $P'$ *s.t.* $P \xrightarrow{x\langle a \rangle} P'$;
2. *If* $P \xrightarrow{a\langle x \rangle} P'$ *and* $x \notin \Gamma$ *then* $\Delta; \Gamma, x \vdash_{\mathrm{L}} P'$;
3. *if* $P \xrightarrow{x\langle a \rangle} P'$ *then* $x \in \Delta$ *and* $\Delta - x; \Gamma \vdash_{\mathrm{L}} P'$;
4. *if* $P \xrightarrow{a\langle b \rangle} P'$ *or* $P \xrightarrow{\overline{a}\langle b \rangle} P'$ *or* $P \xrightarrow{\overline{a}(b)} P'$, *then* $\Delta; \Gamma \vdash_{\mathrm{L}} P'$;
5. *if* $P \xrightarrow{\tau} P'$ *then either* $\Delta; \Gamma \vdash_{\mathrm{L}} P'$ *or there is* $x \in \Delta \cap \Gamma$ *and* $\Delta - x; \Gamma - x \vdash_{\mathrm{L}} P'$;
6. *if* $P \xrightarrow{\overline{x}\langle a \rangle} P'$ *or* $P \xrightarrow{\overline{x}(a)} P'$ *or* $P \xrightarrow{\overline{a}\langle x \rangle} P'$, *then* $x \in \Gamma$ *and* $\Delta; \Gamma - x \vdash_{\mathrm{L}} P'$;
7. *if* $P \xrightarrow{\overline{a}(x)} P'$ *and* $x \notin \Delta \cup \Gamma$ *then* $\Delta, x; \Gamma \vdash_{\mathrm{L}} P'$;

We briefly discuss the typing rules for $\omega$-receptiveness. We assume that there is a single sort of $\omega$-receptive names, ranged over by $z$. The rules are obtained with a few, but simple, modifications from those for linear receptiveness. Now the interpretation of a judgement $\Delta; \Gamma \vdash_{\omega} P$ is that $P$ must make names in $\Delta$ immediately available, in input-replicated form; whereas it may use names in $\Gamma$ arbitrarily many times in output. We only show the new version of rules T-PAR and T-OUT-2, for parallel composition and free output, and one of the rules for replication:

$$\frac{\Delta_1; \Gamma \vdash_{\omega} P_1 \qquad \Delta_2; \Gamma \vdash_{\omega} P_2}{\Delta_1, \Delta_2; \Gamma \vdash_{\omega} P_1 \mid P_2} \qquad \frac{z \in \Gamma \qquad \emptyset; \Gamma \vdash_{\omega} P}{\emptyset; \Gamma \vdash_{\omega} \overline{a}\langle z \rangle. P} \qquad \frac{\emptyset; \Gamma \vdash_{\omega} P}{\{z\}; \Gamma \vdash_{\omega} !z(a). P}$$

Set $\Gamma$ does not change in the premises and in the conclusions because there is no constraints that its names be used in a single place. Similar modifications are required in the clauses of the soundness theorem. A discipline similar to $\omega$-receptiveness is actually used in the compiler of Pict [PT96], to allow optimisations of the code implementing communications.

In our proof of Jones's transformation we shall use both linear and $\omega$ receptiveness. We therefore need a notation for combining the two type systems. If $\Delta_{\mathrm{L}}$ and $\Gamma_{\mathrm{L}}$ are sets of linear receptive names, $\Delta_{\omega}$ and $\Gamma_{\omega}$ sets of $\omega$-receptive names, and $\Delta = \Delta_{\mathrm{L}} \cup \Delta_{\omega}$, and $\Gamma = \Gamma_{\mathrm{L}} \cup \Gamma_{\omega}$, then we write $\Delta; \Gamma \vdash P$ if both $\Delta_{\mathrm{L}}; \Gamma_{\mathrm{L}} \vdash_{\mathrm{L}} P$ and $\Delta_{\omega}; \Gamma_{\omega} \vdash_{\omega} P$ hold. In the remainder, $x, y$ stand for linearly-receptive names; $z$ for an $\omega$-receptive name; $a, b, c, ..$ for non-receptive names; and $p, q, r$ range over the set of all names.

### 3.1 Barbed congruence with receptiveness

As usual in typed calculi, the definitions of the barbed relations take typing into account, so that the composition of a context and a process be well-typed. In the case of receptiveness, an additional ingredient has to be taken into account, namely the input availability of receptive names. If a process has the possibility of using certain receptive names in output, then a context in which the process is tested should guarantee the input-availability at these names, otherwise the essence of receptiveness — the fact that outputs at receptive names can be immediately consumed — is lost. For this, we define the notion of *complete* processes and contexts. In a complete process, no (linear or $\omega$) receptive name can occur free in output position.

**Definition 4 (complete processes and contexts)** *A process $P$ is* complete *if $\Delta; \emptyset \vdash P$, for some $\Delta$. We say that context $C$ is* complete on $(\Delta;\Gamma)$ *if $C[P]$ is complete, for all $P$ s.t. $\Delta;\Gamma \vdash P$.*

**Definition 5 (barbed congruence under receptiveness)** *Suppose $\Delta;\Gamma \vdash P, Q$. Then we say that $P$ and $Q$ are* receptive barbed congruent on $(\Delta;\Gamma)$, *written $P \approx^{\Delta;\Gamma} Q$, if for each context $C$ which is complete on $(\Delta;\Gamma)$ it holds that $C[P] \ \dot{\approx}\ \ C[Q]$.*

### 3.2 Properties of barbed congruence with receptiveness

The typed barbed congruences pose a less tight requirement on names than the ordinary relations of Section 2, hence they are coarser; that is, we have:

$$\text{Suppose } \Delta;\Gamma \vdash P, Q;\text{ then } P \approx Q \text{ implies } P \approx^{\Delta;\Gamma} Q \tag{2}$$

By definition, barbed congruences are preserved by all $\pi$-calculus operators; we are therefore allowed to use simple inference rules like

$$\text{if all names in } \Gamma \text{ are } \omega\text{-receptive, then } P \approx^{\emptyset;\Gamma} Q \text{ implies } !z(\tilde{a}).\, P \approx^{\{z\};\Gamma}\ !z(\tilde{a}).\, Q \tag{3}$$

In [San96a], characterisations of barbed congruences under receptiveness which do not need quantification over contexts are proved. This is done on a subclass of the well-typed processes characterised by the property that all receptive names which are sent are fresh, and defined by means of syntactic restrictions similar to those in the language $\pi$I [San96b]. Using these characterisations, these laws are shown to be valid [San96a]:

$$\overline{x}\langle\tilde{r}\rangle.\, P \approx^{\Delta;\Gamma} \overline{x}\langle\tilde{r}\rangle \mid P \qquad\qquad \overline{z}\langle\tilde{r}\rangle.\, P \approx^{\Delta;\Gamma} \overline{z}\langle\tilde{r}\rangle \mid P \tag{4}$$

Another important law is this; we write $p \rhd q$ as an abbreviation for $p(\tilde{r}).\, \overline{q}\langle\tilde{r}\rangle$

$$\text{if } x \in \Gamma \text{ and } y \notin \Delta \cup \Gamma \text{ then } \quad \overline{b}\langle\tilde{r}, x\rangle.\, P \approx^{\Delta;\Gamma} \boldsymbol{\nu} y\, \overline{b}\langle\tilde{r}, y\rangle.\, (y \rhd x \mid P) \tag{5}$$

The last fact about typed barbed congruences we shall need is this.

$$\text{Let } z \in \Delta \text{ and } p \text{ not free in } P, Q;\text{ then } P \approx^{\Delta;\Gamma} Q \quad \text{iff} \quad \boldsymbol{\nu} z\, (!p \rhd z \mid P) \approx^{\Delta - z;\Gamma - z} \boldsymbol{\nu} z\, (!p \rhd z \mid Q) \tag{6}$$

(The analogous of (5) and (6) for omega and linear receptiveness holds but we shall not need it.) These are powerful equalities; (4) transforms a "synchronous" output into an "asynchronous" one; (5) transforms the output of a *global* name into the output of a *local* name; (6) transforms a global input into a local input. The validity of these equalities crucially rely on the receptiveness hypothesis on names $x, y, z$; none of the equalities is true in the ordinary $\pi$-calculus; nor is it true in other type systems for the $\pi$-calculus like that with input/output annotations [PS93] or with linearity [KPT96].

## 4 The validation of Jones's transformation

### 4.1 Translating $\pi o\beta\lambda$ into the $\pi$-calculus

Compositional translations of $\pi o\beta\lambda$ (as well as of similar languages, like those in the POOL family [Ame89]) into the $\pi$-calculus (or related calculi) are studied in a few of papers in the literature, including [Jon93b, Wal95a, LW95, Wal95b].

When the semantics of a language $\mathcal{L}$ is given as a compositional translation into the $\pi$-calculus, then the meaning of a (well-formed) program is an equivalence class of $\pi$-calculus processes, where the equivalence relation is the behavioural equivalence "=" chosen on the $\pi$-calculus. If "=" should be useful to validate transformations on the source language $\mathcal{L}$, it must be sufficiently well-behaved w.r.t. the $\pi$-calculus constructs so to be preserved by translations of $\mathcal{L}$ contexts. For then, using $A, B$ to range over well-formed $\mathcal{L}$ terms, from $[\![A]\!] = [\![B]\!]$, by the compositionality of the encoding and the properties of "=", we can infer that for all $\mathcal{L}$ contexts $C$ for which $C[A]$ and $C[B]$ are well-formed programs

$$[\![C[A]]\!] = [\![C[B]]\!].$$

Of course, the other important requirement of "=", needed to prove $[\![A]\!] = [\![B]\!]$, is that it should be a mathematically-tractable relation.

Unfortunately it may happen that the ordinary behavioural equivalences of the $\pi$-calculus are too strong for this purpose. With this paper, we would like to show that equipping the $\pi$-calculus with a type system and moving to a typed behavioural equivalence can be a remedy. We would also like to indicate that the type system for receptiveness can be particularly interesting for this.

We shall not report here the encoding of $\pi o\beta\lambda$ into the $\pi$-calculus; classes *Seq* and *Par* are simple enough for their translations to be understandable. We only explain, informally, a few points of the translation in which receptiveness appears. For presentation purposes, our $\pi o\beta\lambda$ language and its translation into the $\pi$-calculus are simplified w.r.t. the original ones; the missing details are easy to add. The translation of a well-formed $\pi o\beta\lambda$ program does not have free receptive names (i.e., it is a complete process); but it has bound receptive names in the body, in the translation of class declaration and method invocation.

Omitting type annotations, the translation of a $\pi o\beta\lambda$ statement

$$\texttt{class}\ T(\tilde{e})\ \texttt{Body}_T; \mathcal{D}$$

containing the declaration of a class $T$ with parameters $\tilde{e}$ and body $Body_T$, and where $\mathcal{D}$ is the rest of the statement (this may have other class definitions and a root object to start the computation) is

$$\boldsymbol{\nu} z \left( !z(\tilde{e}, a).\, [\![\texttt{Body}_T]\!]_a \mid [\![\mathcal{D}]\!]_{T \to z} \right)$$

where $z$ is the $\pi$-calculus name chosen to represent the name $T$ of the class. A $\pi o\beta\lambda$ expression $\texttt{new}\,(T(\tilde{v}))$, that activates an object of the class $T$ with actual parameters $\tilde{v}$, is translated as an output $\boldsymbol{\nu} b \overline{z}\langle \tilde{v}, b\rangle$ where $b$ is a fresh channel that will refer to the newly-created object. The location $z$ of the class declaration has a unique, and replicated, input occurrence that is immediately available at creation; $z$ is used, therefore, as an $\omega$-receptive name.

Interactions between $\pi o\beta\lambda$ objects are in form of remote procedure calls. The translation of the invocation of a method $a$ of an object has the form

$$\boldsymbol{\nu} x\, \overline{a}\langle \tilde{r}, x\rangle.\, x(\tilde{p}).\, P$$

where $\tilde{r}$ are the actual parameters of the call, $x$ the return name, $\tilde{p}$ the formal parameters for the result of the call. The return name $x$ can be used only once in output, to deliver the result, and its input end is immediately available after creation. It is therefore used as a linear receptive name.

Since the $\pi o\beta\lambda$ translation makes use of receptive names, we can hope that, on encoding of $\pi o\beta\lambda$ terms, barbed congruence under receptiveness allow us to prove more interesting equalities than the ordinary barbed congruence.

### 4.2 The proof

The $\pi$-calculus translation of classes *Seq* and *Par* are the processes below. For simplicity, the translation uses recursion, communication of integers and strings. Both recursion and data values are codable into the $\pi$-calculus [Mil91]; it is more convenient, however, to take basic date values as primitives (it is straightforward to accommodate them in the theory of receptiveness previously sketched). We recall from [Mil91] that the encoding of an expression

$$\alpha.\, A\langle\tilde{v}\rangle$$

that uses a recursive definition

$$A\langle\tilde{w}\rangle \stackrel{\mathrm{def}}{=} \ldots A\langle\tilde{d}\rangle \ldots A\langle\tilde{e}\rangle \ldots$$

(we have made visible the recursive calls) is, assuming $a$ is a fresh name,

$$\alpha.\, \boldsymbol{\nu} a \left(\overline{a}\langle\tilde{v}\rangle \mid (!a(\tilde{w}).\ \ldots \overline{a}\langle\tilde{d}\rangle \ldots \overline{a}\langle\tilde{e}\rangle \ldots)\right)$$

We use $m, n$ to range over integers and variables over integers; $t, s$ for strings and variables over strings.

$$\begin{array}{lcl} [\![Seq]\!] & \stackrel{\mathrm{def}}{=} & !z(a,n,s).\, \mathtt{SC}\langle a, nil, n, s\rangle \\ [\![Par]\!] & \stackrel{\mathrm{def}}{=} & !z(a,n,s).\, \mathtt{PC}\langle a, nil, n, s\rangle \end{array}$$

An output $\overline{z}\langle c, m, t\rangle$ activates an instance of the class (i.e., a cell), which is accessible via $c$ and stores integer $m$ and string $t$. In the definitions of $\mathtt{SC}\langle a,b,n,s\rangle$ and $\mathtt{PC}\langle a,b,n,s\rangle$, channel $a$ is used for the insertion requests, and has two parameters: the string to be inserted and a return channel at which the answer will be delivered. Channel $b$ is used for communications with the next cell; it is *nil* in the last cell.

$$\begin{array}{llll}
\mathtt{SC}\langle a,b,n,s\rangle \stackrel{\mathrm{def}}{=} & & & \\
a(t,x).\ \mathtt{if}\ t=s & \mathtt{then}\ \overline{x}\langle n\rangle.\, \mathtt{SC}\langle a,b,n,s\rangle & & \\
 & \mathtt{else}\ \ \mathtt{if}\ b = nil & \mathtt{then}\ \boldsymbol{\nu} c\, \overline{z}\langle c, n+1, t\rangle.\, \overline{x}\langle n+1\rangle.\, \mathtt{SC}\langle a,c,n,s\rangle & \\
 & & \mathtt{else}\ \boldsymbol{\nu} y\, \overline{b}\langle t, y\rangle.\, y(m).\, \overline{x}\langle m\rangle.\, \mathtt{SC}\langle a,b,n,s\rangle & \\
\mathtt{PC}\langle a,b,n,s\rangle \stackrel{\mathrm{def}}{=} & & & \\
a(t,x).\ \mathtt{if}\ t=s & \mathtt{then}\ \overline{x}\langle n\rangle.\, \mathtt{PC}\langle a,b,n,s\rangle & & \\
 & \mathtt{else}\ \ \mathtt{if}\ b = nil & \mathtt{then}\ \overline{x}\langle n+1\rangle.\, \boldsymbol{\nu} c\, \overline{z}\langle c, n+1, t\rangle.\, \mathtt{PC}\langle a,c,n,s\rangle & \\
 & & \mathtt{else}\ \overline{b}\langle t, x\rangle.\, \mathtt{PC}\langle a,b,n,s\rangle &
\end{array}$$

Processes $[\![Seq]\!]$ and $[\![Par]\!]$ use the $\omega$-receptive name $z$ for activation of new cells, and the linear receptive names $x$ and $y$ in method invocation. $[\![Seq]\!]$ and $[\![Par]\!]$ are not in relation $\approx$, since, intuitively, the traces obtained following the `else` branches of the conditionals are different. But we can prove that $[\![Seq]\!]$ and $[\![Par]\!]$ are equated by barbed congruences under receptiveness; precisely we show that they are in the relation $\approx^{\{z\};\{z\}}$ (name $z$ is the only receptive name which occurs free in the processes).

**Theorem 6** $[\![Seq]\!] \approx^{\{z\};\{z\}} [\![Par]\!]$.

**Proof:** We begin by applying the algebraic laws in (4) to the second and third line of SC and to the second line of PC (with $\Delta = \emptyset$, $\Gamma = \{x, z\}$). Reordering parallel components we obtain the new cells

$$
\begin{array}{lll}
\mathtt{SC1}\langle a,b,n,s\rangle \stackrel{\mathrm{def}}{=} & & \\
a(t,x).\ \mathtt{if}\ t=s & \mathtt{then}\ \overline{x}\langle n\rangle.\,\mathtt{SC1}\langle a,b,n,s\rangle & \\
 & \mathtt{else}\ \ \mathtt{if}\ b=nil & \mathtt{then}\ \overline{x}\langle n+1\rangle \mid \boldsymbol{\nu} c\,(\overline{z}\langle c,n+1,t\rangle \mid \mathtt{SC1}\langle a,c,n,s\rangle) \\
 & & \mathtt{else}\ \boldsymbol{\nu} y\,\overline{b}\langle t,y\rangle.\,y(m).\,(\overline{x}\langle m\rangle \mid \mathtt{SC1}\langle a,b,n,s\rangle) \\
\mathtt{PC1}\langle a,b,n,s\rangle \stackrel{\mathrm{def}}{=} & & \\
a(t,x).\ \mathtt{if}\ t=s & \mathtt{then}\ \overline{x}\langle n\rangle.\,\mathtt{PC1}\langle a,b,n,s\rangle & \\
 & \mathtt{else}\ \ \mathtt{if}\ b=nil & \mathtt{then}\ \overline{x}\langle n+1\rangle \mid \boldsymbol{\nu} c\,(\overline{z}\langle c,n+1,t\rangle \mid \mathtt{PC1}\langle a,c,n,s\rangle) \\
 & & \mathtt{else}\ \overline{b}\langle t,x\rangle.\,\mathtt{PC1}\langle a,b,n,s\rangle
\end{array}
$$

and it holds that

$$
\mathtt{SC}\langle a,b,n,s\rangle \approx^{\emptyset;\{z\}} \mathtt{SC1}\langle a,b,n,s\rangle \tag{7}
$$

$$
\mathtt{PC}\langle a,b,n,s\rangle \approx^{\emptyset;\{z\}} \mathtt{PC1}\langle a,b,n,s\rangle \tag{8}
$$

An important step is now the application of law (5) to PC1 (with $\Delta = \emptyset$, $\Gamma = \{x, z\}$). This, intuitively, has the effect of transforming the delegate statement in the definition of class $Par$ into a return statement. We thus obtain a new cell

$$
\begin{array}{lll}
\mathtt{EC1}\langle a,b,n,s\rangle \stackrel{\mathrm{def}}{=} & & \\
a(t,x).\ \mathtt{if}\ t=s & \mathtt{then}\ \overline{x}\langle n\rangle.\,\mathtt{EC1}\langle a,b,n,s\rangle & \\
 & \mathtt{else}\ \ \mathtt{if}\ b=nil & \mathtt{then}\ \overline{x}\langle n+1\rangle \mid \boldsymbol{\nu} c\,(\overline{z}\langle c,n+1,t\rangle \mid \mathtt{EC1}\langle a,c,n,s\rangle) \\
 & & \mathtt{else}\ \boldsymbol{\nu} y\,\overline{b}\langle t,y\rangle.\,(y(m).\,\overline{x}\langle m\rangle \mid .\,\mathtt{EC1}\langle a,b,n,s\rangle)
\end{array}
$$

and it holds that

$$
\mathtt{PC1}\langle a,b,n,s\rangle \approx^{\emptyset;\{z\}} \mathtt{EC1}\langle a,b,n,s\rangle \tag{9}
$$

By (7-9), and using the inference rule (3), we have now reduced ourselves to proving

$$
!z(a,n,s).\,\mathtt{SC1}\langle a,nil,n,s\rangle \approx^{\{z\};\{z\}} !z(a,n,s).\,\mathtt{EC1}\langle a,nil,n,s\rangle
$$

By (6) and (2) this is true if, for some $p$ fresh,

$$
\boldsymbol{\nu} z\,(!p \rhd z \mid !z(a,n,s).\,\mathtt{SC1}\langle a,nil,n,s\rangle) \approx \boldsymbol{\nu} z\,(!p \rhd z \mid !z(a,n,s).\,\mathtt{EC1}\langle a,nil,n,s\rangle)
$$

From now on we can proceed with the *standard theory* of the ordinary barbed congruence $\approx$. We sketch the remaining of the proof. Using distributivity properties of replication and the $\tau$-law of Lemma 2, we get, for $X \in \{\mathtt{SC1}, \mathtt{EC1}\}$:

$$
\begin{array}{r}
\boldsymbol{\nu} z\,(!p \rhd z \mid !z(a,n,s).\,X\langle a,nil,n,s\rangle) \ \approx \\
!p(a,n,s).\,\boldsymbol{\nu} z\,(X\langle a,nil,n,s\rangle \mid !z(a,n,s).\,X\langle a,nil,n,s\rangle)
\end{array} \tag{10}
$$

Therefore, since $\approx$ is a congruence, it suffices to prove

$$
\begin{array}{l}
\boldsymbol{\nu} z\,(\mathtt{SC1}\langle a,nil,n,s\rangle \mid !z(a,n,s).\,\mathtt{SC1}\langle a,nil,n,s\rangle) \ \approx \\
\boldsymbol{\nu} z\,(\mathtt{EC1}\langle a,nil,n,s\rangle \mid !z(a,n,s).\,\mathtt{EC1}\langle a,nil,n,s\rangle)
\end{array}
$$

In the definition of cell SC1, name $z$ is used only once, to activate another copy of the cell. Using again the distributivity properties of replication, this occurrence of $z$ can be replaced by the copy itself of the cell. This gives

$$\begin{array}{llll}
\mathtt{SC2}\langle a,b,n,s\rangle & \stackrel{\mathrm{def}}{=} & & \\
a(t,x).\ \mathtt{if}\ t=s & \mathtt{then}\ \overline{x}\langle n\rangle.\,\mathtt{SC2}\langle a,b,n,s\rangle & & \\
& \mathtt{else}\ \ \mathtt{if}\ b=nil & \mathtt{then}\ \overline{x}\langle n+1\rangle \mid \boldsymbol{\nu} c\,(\mathtt{SC2}\langle c,nil,n+1,t\rangle \mid \mathtt{SC2}\langle a,c,n,s\rangle) & \\
& & \mathtt{else}\ \boldsymbol{\nu} y\,\overline{b}\langle t,y\rangle.\,y(m).\,(\overline{x}\langle m\rangle \mid \mathtt{SC2}\langle a,b,n,s\rangle) &
\end{array}$$

and we have

$$\boldsymbol{\nu} z\,(\mathtt{SC1}\langle a,nil,n,s\rangle \mid !z(a,n,s).\,\mathtt{SC1}\langle a,nil,n,s\rangle) \quad\approx\quad \mathtt{SC2}\langle a,nil,n,s\rangle$$

Acting similarly on EC1 we obtain the new cell

$$\begin{array}{llll}
\mathtt{EC2}\langle a,b,n,s\rangle & \stackrel{\mathrm{def}}{=} & & \\
a(t,x).\ \mathtt{if}\ t=s & \mathtt{then}\ \overline{x}\langle n\rangle.\,\mathtt{EC2}\langle a,b,n,s\rangle & & \\
& \mathtt{else}\ \ \mathtt{if}\ b=nil & \mathtt{then}\ \overline{x}\langle n+1\rangle \mid \boldsymbol{\nu} c\,(\mathtt{EC2}\langle c,nil,n+1,t\rangle \mid \mathtt{EC2}\langle a,c,n,s\rangle) & \\
& & \mathtt{else}\ \boldsymbol{\nu} y\,\overline{b}\langle t,y\rangle.\,(y(m).\,\overline{x}\langle m\rangle \mid \mathtt{EC2}\langle a,b,n,s\rangle) &
\end{array}$$

We are thus left with having to prove

$$\mathtt{SC2}\langle a,nil,n,s\rangle \quad\approx\quad \mathtt{EC2}\langle a,nil,n,s\rangle \tag{11}$$

For this, we can use the labelled (weak) bisimilarity of the $\pi$-calculus (in the late, early, or open style; on processes with only one free name, like those above, all these bisimilarities are congruence relations and imply barbed congruence). The equality can be established by exhibiting an appropriate bisimulation relation. The proof is fairly simple (the two processes are almost identical); one can use standard reasoning as for systems with an inductive structure ([Mil89, chapter 6]); the *up-to expansion technique* [SM92] can be used to reduce the size of the bisimulation relation — the proofs of the running examples in [SM92] are similar to that of (11). □

*Acknowledgements.* The author would like to thank Cliff Jones and Benjamin Pierce for several useful discussions during the early stages of this study. This research has been supported by FRANCE TÉLÉCOM, CTI-CNET 95-1B-182, "Modélisation de Systèmes Mobiles".

## References

[Ame89] P. America. Issues in the design of a parallel object-oriented language. *Formal Aspects of Computing*, 1(4):366–411, 1989.

[Hon96] K. Honda. Composing processes. In *Proc. 23th POPL*. ACM Press, 1996.

[Jon93a] C.B. Jones. Constraining interference in an object-based design method. In M.-C. Gaudel and J.-P. Jouannaud, editors, *Proc. TAPSOFT'93*, volume 668 of *Lecture Notes in Computer Science*, pages 136–150. Springer Verlag, 1993.

[Jon93b] C.B. Jones. A $\pi$-calculus semantics for an object-based design notation. In E. Best, editor, *Proc. CONCUR '93*, volume 715 of *Lecture Notes in Computer Science*, pages 158–172. Springer Verlag, 1993.

[KPT96] N. Kobayashi, B.C. Pierce, and D.N. Turner. Linearity and the pi-calculus. In *Proc. 23th POPL*. ACM Press, 1996.

[LW95] X. Liu and D. Walker. Confluence of processes and systems of objects. In Mosses P. et al. editor, *Proc. TAPSOFT'95*, volume 915 of *Lecture Notes in Computer Science*, pages 217–231. Springer Verlag, 1995.

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

[Mil91] R. Milner. The polyadic $\pi$-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.

[MS92] R. Milner and D. Sangiorgi. Barbed bisimulation. In W. Kuich, editor, *19th ICALP*, volume 623 of *Lecture Notes in Computer Science*, pages 685–695. Springer Verlag, 1992.

[PS93] B. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. In *Proc. 8th LICS Conf.*, pages 376–385. IEEE Computer Society Press, 1993. To appear in *Journal of Mathem. Structures in Computer Science.*

[PT96] B. C. Pierce and D. N. Turner. Pict: A programming language based on the pi-calculus. In preparation, 1996.

[PW96] A. Philippou and D. Walker. On transformations of concurrent object programs. In *Proc. CONCUR '96*, Lecture Notes in Computer Science, pages 131–147. Springer Verlag, 1996.

[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.

[San96a] D. Sangiorgi. The name discipline of receptiveness. To appear as an INRIA-Sophia Antipolis TR. Submitted to publication, 1996.

[San96b] D. Sangiorgi. $\pi$-calculus, internal mobility and agent-passing calculi. *Theoretical Computer Science*, 167(2):235–274, 1996.

[SM92] D. Sangiorgi and R. Milner. The problem of "Weak Bisimulation up to". In W.R. Cleveland, editor, *Proc. CONCUR '92*, volume 630 of *Lecture Notes in Computer Science*, pages 32–46. Springer Verlag, 1992.

[Wal95a] D. Walker. Objects in the $\pi$-calculus. *Information and Computation*, 116(2):253–271, 1995.

[Wal95b] D. Walker. Process calculus and parallel object-oriented programming languages. In P. Tvrdik T. Casavant and F. Plasil, editors, *Parallel Computers: Theory and Practice*, pages 369–390. Computer Society Press, 1995.